LUND UNIVERSITY

On Offensive and Defensive Methods in Software Security

Jämthagen, Christopher

2016

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Jämthagen, C. (2016). *On Offensive and Defensive Methods in Software Security*. [Doctoral Thesis (compilation), Department of Electrical and Information Technology]. The Department of Electrical and Information Technology.

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

On Offensive and Defensive Methods in Software Security

Christopher Jämthagen

LUND UNIVERSITY

Doctoral Dissertation
Electrical Engineering
Lund, November 2016

Christopher Jämthagen
Department of Electrical and Information Technology
Electrical Engineering
Lund University
P.O. Box 118, 221 00 Lund, Sweden

Series of licentiate and doctoral dissertations
ISSN 1654-790X; No. 89
ISBN 978-91-7623-942-1

© 2016 Christopher Jämthagen
Typeset in Palatino and Helvetica using LaTeX 2_ε.
Printed in Sweden by Tryckeriet i E-huset, Lund University, Lund.

No part of this dissertation may be reproduced or transmitted in any form or by any means, electronically or mechanical, including photocopy, recording, or any information storage and retrieval system, without written permission from the author.

Abstract

This thesis presents new methods contributing to the area of software security. Both offensive and defensive methods are proposed, where the offensive methods presented in this thesis mostly deal with how an attacker can embed malicious code in a stealthy manner, and the defensive methods aims at detecting some form of attack.

The first approach deals with how a virtual machine can be detected and we discuss its use as both an offensive as well as a defensive method. We develop a proof-of-concept that aims to demonstrate how the technique works in practice.

Next we implement a GlobalPlatform compatible RPC mechanism utilizing both a hypervisor and SELinux and provide some benchmarks for both solutions.

Following this we will look at timestamping of data on a massive scale and how by utilizing the blockchain of the Bitcoin network, we can gain Byzantine fault tolerance for the Keyless Signing Infrastructure.

Then we will look at methods for obfuscating code by overlapping assembly instructions in machine code. We do this both by crafting custom no-operation instructions in the binary, but also provide a method to accomplish this in the source code when that source code will be compiled via a deterministic building process to produce the expected binaries.

Finally this thesis will conclude with a method for detecting Return Oriented Programming attacks by analyzing the raw data of a network stream.

Contents

Contents v

Preface xi

Popular Scientific Summary in Swedish xiii

Acknowledgments xvii

1 Introduction 1

1.1 Complexity 2

1.2 Offense and Defense 3

1.3 Trust 4

2 Security through Compartmentalization 7

2.1 Virtualization 7

2.1.1 Hypervisors 8

2.1.2 Paravirtualization 9

2.1.3 Full Virtualization 10

2.1.4 Virtualization on x86 10

2.2 Mandatory Access Control Schemes 11

2.2.1 SELinux 11

2.3 Trusted Execution Environments 13

2.4 Other Solutions 14

3 Blockchain 15

3.1 Background 15
3.1.1 Cryptographic Hash Functions 15
3.1.2 Merkle Trees 16
3.1.3 Byzantine Fault Tolerance 16
3.2 Blockchain and Proof-of-Work 16
3.2.1 Transactions 19
3.3 Use Cases of Blockchain Technology 23
3.3.1 Decentralized DNS 23
3.3.2 Smart Contract Expressiveness and Sidechains 24
3.4 Security of Blockchains 25

4 Software Security 27

4.1 The x86 Architecture 27
4.1.1 Anatomy of an x86 Instruction 28
4.1.2 No-Operation Instructions 28
4.1.3 Overlapping Instructions 30
4.2 Reverse Code Engineering 31
4.2.1 Disassembly 31
4.2.2 Anti-Disassembly 32
4.3 State of Software Security 34
4.3.1 Backdoors 34
4.3.2 Deterministic Builds 35
4.3.3 Offensive Techniques 36
4.3.4 Defensive Techniques 39
4.4 Conclusions 41

5 Virtual Machine Detection 43

5.1 Introduction 43
5.2 Virtual Machine Detection Techniques 44
5.2.1 Timing Analysis 45
5.3 A New Technique for Remote and Passive VMM Detection . . 45
5.3.1 Network Address Translation (NAT) and Network Protocols 46
5.3.2 Prerequisites 47
5.3.3 Time-To-Live (TTL) 47
5.3.4 IP Identification (IP ID) 48

5.3.5 TCP Control Flags 49
5.4 Proof of Concept 49
5.5 A Note on IPv6 50
5.6 Conclusions 50

6 Secure RPC Mechanisms for Embedded Systems 53
6.1 GlobalPlatform 54
6.1.1 The Client API 54
6.2 Hypervisors 55
6.3 Implementation 56
6.3.1 Linux Shared Memories 57
6.3.2 Implementing GlobalPlatform with SELinux 57
6.3.3 The Hypervisor as a GlobalPlatform TEE 58
6.4 Evaluation 60
6.4.1 Evaluated Implementations 61
6.4.2 Performance Results 61
6.4.3 Implementation Effort 62
6.4.4 Security Considerations 64
6.5 Conclusions 65

7 Blockchain for the Keyless Signing Infrastructure 67
7.1 Keyless Signing Infrastructure 68
7.1.1 Limitations of KSI 69
7.2 Design 70
7.2.1 Broadcasting CRHs 72
7.2.2 Complex Spending Conditions 72
7.3 Further Considerations 75
7.3.1 Merkleized Abstract Syntax Trees 75
7.3.2 Relative Locktime 76
7.4 Discussion 76
7.5 Conclusion 77

8 Creating Hidden Code Using NOP Instructions 79
8.1 A New Technique for Overlapping Instructions 80
8.1.1 Requirements 80
8.1.2 Overview of the Main Idea 80

8.2 Suitable MEP Instructions 82
8.3 Assembling the Hidden Execution Path 83
8.3.1 Hiding Code in a Linear Stream of NOPs 83
8.4 Additional Practical Considerations 86
8.4.1 Hiding Code in Scattered NOPs 86
8.4.2 Normalization of MEP instructions 88
8.5 Detection 89
8.5.1 Anti-Analysis 89
8.5.2 Detection Algorithm 92
8.6 Conclusions 93

9 Exploiting Trust in Deterministic Builds 95
9.1 Hiding Instructions in Binary Code 95
9.1.1 Main and Hidden Execution Paths 96
9.1.2 Basic Design 96
9.1.3 MEP-to-HEP Mappings 97
9.2 Constructing the HEP from Source Code 98
9.2.1 Structs Allocated on the Stack 102
9.2.2 Control Flow 103
9.3 Evading Analysis 105
9.3.1 In the Source Code 105
9.3.2 In the MEP 106
9.3.3 In the HEP 106
9.4 A Proof of Concept 107
9.5 Discussion 109
9.5.1 Maintainability 109
9.5.2 Stealthiness 110
9.5.3 Strengths and Weaknesses 110
9.5.4 Prevention and Detection 111
9.6 Conclusion 112

10 Detecting ROP Payloads in Data Streams 113
10.1 Overview of Approach 114
10.2 A More Detailed Description 116
10.2.1 Optional Data Pre-Filter 116

10.2.2 Cluster Detection 117
10.2.3 Pattern Matching 119
10.2.4 Statistical Test 123
10.3 Performance 128
10.3.1 Detecting Exploits 129
10.4 Strengths and Limitations 130
10.4.1 Strengths 130
10.4.2 Limitations 131
10.5 Conclusions 132

References **133**

Preface

This thesis contains results from research performed by the author at the Department of Electrical and Information Technology at Lund University. Parts of the material have been presented at international conferences. Publications and reports have appeared as follows.

- C. Jämthagen, M. Hell, B. Smeets, »A technique for remote detection of certain virtual machine monitors,« *International Conference on Trusted Systems*, pp 129-137, Springer, 2011.
- A. Vahidi, C. Jämthagen, »Secure RPC in embedded systems: evaluation of some GlobalPlatform implementation alternatives,«, *Proceedings of the Workshop on Embedded Systems Security*, ACM, 2013.
- C. Jämthagen, P. Lantz, M. Hell, »A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries,«, *Anti-malware Testing Research (WATeR)*, pp 1-9, IEEE, 2013.
- C. Jämthagen, L. Karlsson, P. Stankovski, M. Hell, »eavesROP: Listening for ROP Payloads in Data Streams,«, *International Conference on Information Security*, pp 413-424, Springer, 2014.
- C. Jämthagen, M. Hell, »Blockchain-based publishing layer for the Keyless Signing Infrastructure,«, *The 13th IEEE International Conference on Advanced and Trusted Computing*, IEEE, 2016.
- C. Jämthagen, P. Lantz, M. Hell, »Exploiting trust in Deterministic Builds,«, *International conference on computer safety, reliability and security*, Springer, 2016.

CONTRIBUTION STATEMENT

The author of this dissertation is the main author of all but one of the listed publications.

The idea behind the publication titled »A technique for remote detection of certain virtual machine monitors« come from results found in the author's master's thesis. The author of this thesis was responsible for the proof-of-concept code that demonstrated the technique in question. All authors contributed equally to the writing of the final publication.

The publication titled »Secure RPC in embedded systems: evaluation of some GlobalPlatform implementation alternatives« arose from collaboration with the SICS institute, whom the main author's affiliation was with. SICS provided implementation and analysis of the hypervisor implementation of the publication. This thesis's author contributed with implementation and analysis of the SELinux part of the research. Both authors contributed equally to the writing of the final publication.

In »A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries« the author of this thesis provided an implementation of the proof-of-concept code and accompanying scripts related to the detection of the technique. All author's contributed equally to the discussion of the ideas and analyses of the idea and the writings of the final publication.

The implementation in »eavesROP: Listening for ROP Payloads in Data Streams« was a collaborative effort of the PhD students. All authors participated in discussions, design, analysis and writing of the paper.

The author of this thesis provided the design presented in the publication titled »Blockchain-based publishing layer for the Keyless Signing Infrastructure«. Both authors contributed to the discussion, analysis and writing of the final publication.

The basic idea and design principles presented in »Exploiting trust in Deterministic Builds« was conceived by the main author. The main author also provided the proof-of-concept code that was published. All of the authors contributed to the discussion, analysis and writing of the final publication.

Popular Scientific Summary in Swedish

Avhandlingen tar upp flera aspekter som relaterar till säkerhet i mjukvara. Fokus ligger på både offensiva och defensiva metoder, där de offensiva bidragen handlar om hur exekverbar kod kan gömmas på ett sätt som gör det svårt att upptäcka. De defensiva bidragen relaterar till tidsstämpling av data och detektering av skadlig kod.

DOLDA EXEKVERINGSVÄGAR

När vi skriver texter inkluderar vi ofta ord vi aldrig menat ska vara del av texten. Ord som döljer sig bakom andra ord eller som är en kombination av flera ord. Ta exempelvis ordet "premiss". I detta ord kan vi hitta flera andra ord som "rem", "remiss", "miss" och "is". Ord som inte nödvändigtvis har någonting med sammanhanget i texten att göra. På samma sätt jobbar processorn i en dator med ord i maskinkod, och som den läser och exekverar en efter en. Ett ord i maskinkod kallas för en instruktion och varje instruktion manipulerar värden på ett sätt som programmeraren ämnade den att göra. En instruktion kan också innehålla andra instruktioner beroende på vilken bokstav, eller byte, i instruktionen vi startar exekvering från.

När vi läser en text så vet vi att den ska läsas sekventiellt, ord efter ord, och inte avbrytas mitt i en mening för att hoppa till ett annat stycke och sedan återvända. En dator däremot lyder vår minsta vink och kommer följa alla invecklade instruktioner på ett deterministiskt vis. Denna absoluta lydnad kan utnyttjas för att få den att exekvera instruktioner som programmet aldrig ämnade finnas tillgängligt, men som ändå finns där implicit. På detta viset kan vi skapa dolda meningar, eller exekveringsvägar, som innehåller skadlig kod ämnad att ta kontroll över datorn som koden exekveras på. Detta är en

form av steganografi, vilket är konsten att dölja meddelanden så att det inte är uppenbart att det finns något meddelande över huvud taget. Istället för meddelanden avsedda för en mottagare så kan vi skapa dolda exekveringsvägar avsedda för datorn att exekvera, givet inkörsporten.

Vår forskning visar att det är möjligt att introducera dolda exekveringsvägar direkt via källkoden. Källkoden är den kod som skrivs i läsbar form och som sedan omvandlas, eller kompileras, till maskinkod. Problemet är att den här omvandlingen oftast resulterar i exekverbara filer som skiljer sig jämfört med om någon annan kompilerar samma källkod. Detta är ett problem som resulterar i att användare måste lita på att den som kompilerade och distribuerade den exekverbara filen ej har ändrat i den innan kompileringen skedde. Deterministisk kompilering gör att kompileringsprocessen blir deterministisk och att oavsett vem som kompilerar källkoden så kommer det resultera i identiska exekverbara filer. Genom att en stor mängd användare som själva kompilerat via den här processen intygar att inget har gått snett, kan övriga användare känna sig tryggare i att ingen har manipulerat med källkoden innan kompilering. Det skulle krävas att samtliga användare som kompilerade källkoden var i maskopi för att producera samma skadliga variant av den exekverbara filen och gå i god för den. Eftersom processen är öppen för alla är det liten risk att någon skulle lyckas smyga in skadlig kod i källkoden.

Om deterministisk kompilering används för ett projekt så kan vi utnyttja det för att skriva källkod som efter kompilering inkluderar dolda exekveringsvägar. Om vi kan bekräfta att en dold exekveringsväg finns när vi kompilerar den, så kan vi vara säkra på att den finns där om någon annan kompilerar samma kod.

Resultaten visar att det är fullt möjligt att introducera dessa dolda exekveringsvägar på ett relativt enkelt sätt, men att de samtidigt är ömtåliga då minsta lilla ändring i källkoden till programmet skulle kunna förstöra hela den dolda exekveringsvägen. Genom att arrangera om instruktioner i programmet på ett sätt som bevarar semantiken, på samma sätt som vi kan bygga om meningar och bevara innebörden av den. Alternativt kan vi ersätta instruktioner med synonyma motsvarigheter. Då skulle man kunna göra den här typen av attack harmlös.

TIDSSTÄMPLING

Att bevisa att viss information fanns tillgänglig vid en viss tidpunkt var tidigare svårt utan att blanda in en notarius publicus. Tidsstämpling har ofta varit förknippat med uppvisande av informationen för en tredje part som agerat som vittne när vi vill bevisa det för någon annan vid ett senare tillfälle.

Detta kräver att den vi vill bevisa tidsstämpeln för litar på att vittnet faktiskt talar sanning.

Sedan 2009 har ett decentraliserat nätverk existerat, kallat Bitcoin, vars främsta funktion har varit att tillhandahålla en form av digitala kontanter. Det fungerar genom att alla transaktioner registreras i en öppen databas, kallad blockkedjan, som var och en själv kan verifiera att de transaktioner som berör dom faktiskt stämmer. Samma databas kan användas som ett vittne till information som ska tidsstämplas. Genom att ta en signatur av datan som ska tidsstämplas och lägga in den i blockkedjan får den en tidsstämpel med ett par timmars precision, samt att du inte behöver avslöja den data du lägger in i blockkedjan. Signaturen av datan är unik för din information, och när du väljer att offentliggöra din tidsstämplade data kan andra parter skapa samma signatur och verifiera att den finns i blockkedjan.

Anledningen till att blockkedjan fungerar som ett vittne för tidsstämpling är den sammanlagda mängden energi som krävs för att kunna lägga in information i den. Blockkedjan består av länkade block, och när ett nytt block med data ska läggas till måste ett kryptografiskt pussel lösas för att blocket ska ses som giltigt av resten av nätverket. Det enda sättet att lösa detta pussel är med rå datorkraft. Blocket kommer även att innehålla den ungefärliga tiden då det skapades. Om skaparen av blocket skulle sätta en helt felaktig tidsstämpel så skulle blocket avvisas av resten av nätverket, och inte bli en del av blockkedjan. Eftersom det kostar att skapa block finns incitament att inte avvika från de regler systemet måste följa för att block ska ses som giltiga. Varje block betalar ut en viss summa av dess interna valuta, även dessa kallade bitcoin, vilket gör incitamenten ännu starkare att inte avvika från reglerna då ett förlorat block innebär ett inkomstbortfall för dess skapare.

På grund av dessa incitament kan vi få goda garantier på att på att blocken som skapades för länge sedan faktiskt skapades omkring den tidpunkt som deras tidsstämpel vittnar om. Detta innebär också att informationen som finns i det blocket också måste ha existerat vid den tidpunkten.

Dessvärre så kan vi inte få bättre finkornighet på tidsstämpeln än ett par timmars precision. Anledningen till detta är att distribuerad konsensus är en långsam process. För att få tidsstämplar med en sekunds precision krävs betrodda tredje parter. Vår forskning visar att centraliserade system som kan tidsstämpla data per sekund, kan utnyttja blockkedjan för att tidsstämpla sina tidsstämplar. Genom att kombinera betrodda tredje parters tidsstämplar med en tidsstämpel i den decentraliserade blockkedjan så kan vi få det bästa av båda världar. Vi kan få starka garantier på att informationen som finns i blockkedjan stämmer och att den mer exakta tidsstämpeln som tillhandahålls av den betrodda tredje parten ligger i närheten av den tidsstämpeln i blocket.

DETEKTERING AV VIRTUELLA MASKINER

När angripare ska ta över en dator vill de helst komma åt icke ont anande användare som sällan har koll på säkerheten på sina system. På det viset kan angriparen ha tillgång till systemet en längre tid. Forskare försöker imitera osäkra system för att locka till sig angripare i syfte om att studera deras metoder. För att göra detta skapar de ofta vad som kallas virtuella maskiner, ett operativsystem som körs som en applikation i ett annat operativsystem. Denna virtuella maskin ska vara omedveten om sin omgivning, och från det yttre operativsystemet kan forskarna se i detalj vad angriparen sysslar med. Genom isolering kan forskarna hålla angriparen ute från deras system.

Detta har fått angripare att utnyttja metoder som ämnar att detektera om systemet de är inne på är en virtuell maskin eller inte. Om de upptäcker att de är inne i en virtuell maskin kan de bestämma sig för att inte gå längre då det finns en risk att de kan vara övervakade.

Vår forskning presenterar en teknik som gör det möjligt för angripare att upptäcka om en maskin är virtuell eller inte. Metoden kräver att angriparen ansluter till en speciell webbserver som analyserar paketen för att avgöra om de härstammar från en virtuell maskin.

Andra användsområden för tekniken kan vara för angripare att rikta in sig på virtuella maskiner som är sårbara på något vis. Om de försöker angripa en maskin som inte är virtuell riskerar de att avslöja sina hemligheter. Genom detektering kan deras metoder användas med minimal risk för att avslöjas.

Acknowledgments

I owe a debt of gratitude to so many people whom have helped me throughout my time as a PhD student. First and foremost I am very grateful for the support received from my supervisors Martin Hell and Ben Smeets. Without your help and encouragement, this work would not have been possible.

Many thanks goes out to my colleagues in the crypto group, with whom I have had many insightful discussions with, and which have led to better results in my research.

My family, whom have stood by me my entire life - My mother Anna, my father Constantine, my brothers Alexander, Kevin and Andre and my grandmother Rut - thank you for all the love and support throughout all these years.

The love of my life, Maria; Thank you for encouraging me and standing by me, even when I had to work through weekends and evenings. I promise that I will make it up to you sweetheart.

Finally, the little ones I think about whenever I am about to do something, and whom I want to make the world a better place for; my children, Isabella and Vincent. Thank you for always putting a smile on my face. I love you.

Christopher
Lund, November 2016

1 Introduction

Security solutions for IT systems have evolved immensely as we have come to rely on those systems in our daily lives. From financial institutions keeping track of stock trades to the home user buying something online, everyone is reliant on the confidentiality, integrity and availability of their data and services.

One example of security solutions is Anti-Virus (AV) software. AV software was long the norm, and still is, to protect users against malicious code. Traditional AV software is not optimal though, as it is based on static signatures of malicious code. Malware authors would simply slightly rewrite their code, and the AV software would be unable to detect it. AV software is most effective against malware that spreads on a larger scale and not so effective against directed attacks.

For enterprise users there is a plethora of tools to secure their IT perimeter, i.e., the boundary between private and public parts of a target system. Some of the available tools include stateful firewalls and Intrusion Detection/Prevention Systems. Despite such tools and the defense in depth applied, the number of breaches are staggering [sta16]. Defense in depth is a strategy where multiple defenses are deployed in order to maximize cost and effort for an attacker to be successful, while simultaneously increasing our chances of detecting the attacker due to the increased time the attacker must spend for a successful breach.

This thesis will discuss some new ideas on both the defensive as well as offensive aspects of computer security practices. We discuss how complexity in software can hurt the security of systems. Our efforts of exploiting some of these complexities can be found in Chapter 8 and 9 and are based on the publications »A new instruction overlapping technique for anti-disassembly and

obfuscation of x86 binaries« and »Exploiting trust in Deterministic Builds«. These publications look at the general complexities found in some processor architectures and how they can be abused for nefarious purposes. The publication titled »Blockchain-based publishing layer for the Keyless Signing Infrastructure« relates in general to the area of trust. It considers how timestamping of data can be done even with minimal trust requirements. Publications »Secure RPC in embedded systems: evaluation of some GlobalPlatform implementation alternatives« and »eavesROP: Listening for ROP Payloads in Data Streams« relates to defensive research and covers how execution can be safely contained from some other part of the system and how a specific type of attack can be detected respectively. The publication »A technique for remote detection of certain virtual machine monitors« describes a technique that we show can have both offensive and defensive uses.

1.1 COMPLEXITY

Unnecessary complexity can often be detrimental to the security of an application. More features, more intertwined code and generally a larger Trusted Computing Base (TCB), the more difficult it will be to secure and properly test the intended functionality of a compute system. The more complex an implementation is, the more likely it is that a malicious third party can subvert the intended functionality of that implementation to do something it was never intended to do. The attack surface of an implementation describes how exposed the implementation is to attacks. In software, more functionality, more lines of code and more complexity increases the attack surface by giving an attacker greater opportunities of successfully finding vulnerabilities, i.e., weaknesses that can be exploited to compromise a system.

One such complexity that we exploit in this thesis is that of the instruction set of Intel's x86 processor architecture, which we will refer to as x86. x86 is a so called Complex Instruction Set Computing (CISC) architecture. CISC provides a large set of complicated instructions that pack functionality in individual instructions. This comes at the cost of added complexity to implement, and because CISC provides many different instructions and configurations, a complicated scheme is necessary to encode and decode such instructions. As a consequence, this makes the size of instructions variable in order to minimize space requirements, which is a property we exploit in Chapters 8 and 9.

The other dominant type of architecture is the Reduced Instruction Set Computing (RISC) architecture. In RISC, an instruction will only do one thing and it is up to the programmer (or rather the compiler) to use the necessary instructions to accomplish the desired functionality. This is in contrast to CISC architectures that aims to provide the programmer with one instruction

to do much of the desired functionality. Since RISC is relatively simple, the encoding of instructions are fixed, and in that way the techniques presented in this thesis that exploits the property of executing instructions from unaligned offsets are not applicable, due to RISC not allowing that to happen.

1.2 OFFENSE AND DEFENSE

John Lambert, General Manager at Microsoft's Threat Intelligence Center, once said "If you shame attack research, you misjudge its contribution. Offense and defense are not peers. Defense is offense's child." [Lam14]. Research in offensive techniques is sometimes misconstrued as something that only benefits malicious players without realizing that those malicious players may already be aware of the newly described offensive technique. By looking for new ways to break things, researchers can simultaneously study how to defend against any new potential offensive techniques. By ignoring the offensive part of research and leaving it up to malicious actors, while only focusing on the defensive aspects of already known vulnerabilities, will limit the security community's ability to respond to new threats.

When an attacker discovers a new attack vector, i.e., a specific way an attack can be executed, it will often be met with a patchwork defense. A defense which adds an additional layer of complexity on top of the vulnerable component. This added complexity may open up further attack vectors, or not completely solving the problem which may end up giving users a false sense of security. Rarely is the underlying problem solved, probably because that would incur a greater initial cost and a longer time to deploy. As an example, an early defense against buffer overflow attacks was the Address Space Layout Randomization feature (ASLR) which made it more difficult for the attacker to know at what address his shellcode, or malicious code, will be located. Nowadays ASLR is routinely bypassed by attackers [Ada15] as it did not solve the underlying problem of buffer overflow attacks, in which a program fails to check the boundary of a buffer which may lead to overwriting adjacent memory locations.

The cat-and-mouse game between attackers and defenders is not going to stop anytime soon, and new ways to defend against unknown attacks need to be considered. One interesting technology here is the Qubes operating system [RW10] that allows the user to seamlessly create new virtual machines on-the-fly and contain applications that may be more vulnerable from other, more security-sensitive, applications. Compromise of one virtual machine would leave the others secure, barring some compromise of the underlying virtualization technology. Functionality to automatically open new and unknown pdf-files can, as an example, be done in a so-called "disposable"

VM [RW13], that is created for the only purpose of opening that file, after which it is disposed. This new way of thinking is going to be necessary to minimize damages of attacks, as it is unlikely that one is able to counter all attacks in general.

1.3 TRUST

We must trust that the software we run on our devices is free from bugs that could enable an attacker to gain access to our machine. We must trust that the applications running on our devices are not backdoored. We must trust that a Certificate Authority (CA) that attests to the identity of our bank has done its job, and the list goes on. In security, much boils down to trust. Even hardware must be trusted where the manufacturing process has become so complex due to the huge amount of transistors, an attacker located at the factory that produces the integrated circuits can sneak in one gate in order to create a stealthy backdoor which is only triggered with a specific sequence of events which is unlikely to be triggered accidentally [YHD+16]. It would not matter much if all the software on the machine was 100% bug-free if the hardware it is running on is compromised.

The main issue with trust is often that we need to apply it to a third party, and often we do not have any choice (or at least a very limited choice) in who to trust for a particular use case. One approach to create trustworthiness starts with that we should aim to distribute trust over as many entities as possible. If we retrieve some data from a single entity, it is much easier for that entity to play tricks on us, whereas if we retrieve data from multiple entities, a majority of them must have been corrupted or be colluding and provide us with the same faulty data for us to be cheated.

Open source software is an excellent area where trust is widely distributed. From developing to testing to using it, everyone can see the code that is running and independently verify it. Of course not everyone is savvy enough to parse computer code, but those that do can provide guarantees to others that no irregularities were found. This provides users with a more dynamic set of witnesses of the code than that of a company's proprietary software, which likely only is tested by their own engineers. Of course, there is another intermediary we must trust in open source, namely the compilation process. What turns the human-readable code into machine-readable code. A lot of things can happen during this transformation, and is another point we must trust to do its thing. We can compile it ourselves, but again, most users are not tech-savoy enough to do this and will more likely rely on binaries pre-built by third parties that will easily be double-clicked and installed. To solve this problem a technique called "deterministic building" (see more in 4.3.2) was

developed which aims to make the compiled binary identical no matter who built it. This enables a set of builders to attest to the hash digest of a binary and let users gain greater confidence that if all these builders produced the same binary, then it is more likely to be the product of the open source code. With deterministic building we have distributed the trust of the compilation process from one builder, to many builders. In this thesis we show that the trust in even deterministically built binaries can be subverted. See Chapter 9.

At the end of the day we want the entities we rely on to be trustworthy. By extension this requires these entities to build a reputation of being trustworthy through their course of action. Either by developing code with minimal flaws, issuing certificates in a correct manner or allow extensive security reviews of their products and development processes. The reputation of an entity that fails to operate according to users expectations can find their reputation quickly deteriorating. Whether its failure is due to malice, incompetence or inaction does not matter when the end result makes the trustworthiness of that entity questionable.

Better yet, we can make the level of trust in third parties to an absolute minimum for some use cases by utilizing Bitcoin's blockchain. Bitcoin solves the Byzantine generals problem [LSP82] in which a system can continue to operate normally even under the influence of some malicious actors. We explore Bitcoin's blockchain in Chapter 3 and utilize it for timestamping in conjunction with the Keyless Signing Infrastructure in Chapter 7.

2

Security through Compartmentalization

Isolating running programs makes it harder for a compromised program to affect the correct functioning of other programs. A basic type of isolation has existed in many operating systems for a long time in that the memory allocated to one process is not reachable by another. The concept of isolation has been expanded upon ever since and in this background chapter we will look at a couple of common technologies used to achieve security through isolation.

2.1 VIRTUALIZATION

Virtualization is, in essence, a process of abstracting away the physical characteristics of the underlying hardware from the OS and having an intermediate layer, called the Virtual Machine Monitor (VMM) or hypervisor, which distributes resources to the virtual machines in some manner. This will isolate entire operating systems, not only processes, from each other. The technology has been around since the 1960's [Ros04], initially used to divide system resources between different applications on a mainframe.

Isolating through virtualization is a common practice today for a wide range of use cases including analyzing malware, which give the analyst the possibility to take a snapshot of the system before infection, and ability to revert back to that state when analysis is completed. In this way the analyst will not have to re-install the system from scratch, which saves a lot of time. Normal users can benefit from the technology as well, for example by utilizing the Qubes operating system [RW10] which allows an arbitrary number of virtual machines to be run with the goal of allowing the user to partition their tasks into different domains in an isolated manner.

Virtualization can also provide benefits such as load balancing, by taking full advantage of the available resources, and hardware multiplexing. Another useful feature of virtualization is easy migration. If a system must upgrade its hardware, the virtual machine could retain compatibility if the hypervisor handles the communication between OS and hardware, making such a move much more frictionless.

2.1.1 HYPERVISORS

The hypervisor is one way to enable virtual machines. The computer on which a hypervisor is running is called the host, and any virtual machine running under this hypervisor is called a guest. The notion of a hypervisor was first described by Popek and Goldberg in [PG74], and the authors classified two different types of hypervisors.

Type-1 This type of hypervisor runs directly on the host's hardware in order to control resource management to guests. Another name for this type of hypervisor is bare-metal hypervisor. Examples of type-1 hypervisors are Xen, Microsoft's Hyper-V and VMware ESX/ESXi.

Type-2 This type of hypervisor is executed as any other program on the main operating system of the host. Examples of type-2 hypervisors are VMware Workstation/Player, Oracle's VirtualBox and QEMU.

The responsibilities of hypervisors range from memory management to CPU scheduling for all VMs. In order for a CPU to be virtualizable, Popek and Goldberg set forth a set of requirements that should be met. They divided instructions into three categories.

Privileged instructions are defined as instructions that can be executed in a privileged mode only. If a privileged instruction is executed in non-privileged mode, it will fail.

Control sensitive instructions are defined as instructions that attempt to change the configuration of different resources.

Behavior sensitive instructions are defined as instructions that have different behavior depending on the configuration of different resources.

With these definitions they said that for a CPU to be virtualizable, the sensitive instructions must be a subset of the set of privileged instructions, i.e., the sensitive instructions must also be privileged. This is necessary for the hypervisor to be able to intercept those instructions that change the state of the CPU, otherwise the isolation between guests could become compromised.

As an example, the x86 architecture does not fulfill these requirements for virtualization. x86 has a set of 17 instructions which are sensitive, but not privileged. Thus, since these instructions do not cause faults when executed in user mode, they cannot be intercepted by the hypervisor. The hypervisor cannot manage virtual machines because inconsistencies can occur when one guest modifies configurations and does not notify the hypervisor.

One example of these virtualization-incompatible instructions is the `SIDT` instruction, which stands for Store Interrupt Descriptor Table. This instruction can be executed in user mode and will copy the contents of the Interrupt Descriptor Table Register (IDTR) into the destination operand of the instruction. If the system is virtualized, the hypervisor cannot intercept the SIDT when made on a guest, and leave the hypervisor unable to serve the guest with the correct information.

Popek and Goldberg also listed three properties that should be fulfilled for virtualization

The Equivalence property says that an application executed inside a virtual machine should be indistinguishable from the same application being run in a real machine. This property is also known as the fidelity property.

The Resource control property says that the hypervisor must have complete control over the virtualized resources. This is also known as the safety property.

The Efficiency property says that most of the instructions should be executed without interference of the hypervisor. This is also known as the performance property.

2.1.2 PARAVIRTUALIZATION

In paravirtualization, the process of virtualization is greatly simplified by having strategic changes made to the guest operating system to maximize its performance. An API is provided for guests to access certain hardware resources. Obviously this requires the guest OS to be modified and aware of the hypervisor in order to use the interface provided.This often brings performance gains for paravirtualized guests compared to other virtualization techniques.

One popular product that provides paravirtualization is Xen [BDF+03]. It is their hypervisor that is utilized in the Qubes OS mentioned earlier. Xen allows one VM (dom0) to have full access to the hardware, from which management of all the other VMs (domains) happens. A paravirtualized operating system is loaded in dom0 at boot time.

Xen has suffered from many vulnerabilities over the years [xen16], some of which could have allowed an attacker to escape a guest OS and enter the

host [Rut16]. A so called 'VM escape' attack is the ultimate break from the isolation that virtualization provides [SML10].

2.1.3 FULL VIRTUALIZATION

With Full virtualization, the aim is to completely abstract away all hardware capabilities and provide them via the hypervisor such that the guest OS can stay unmodified, unlike the case with paravirtualization. Modern CPUs usually support hardware assisted full virtualization modes. This type of virtualization is especially suitable when running proprietary systems which have no paravirtualized version available to them, such as Windows operating systems. It is also the best choice for when you do not want the user of the system to know he or she is running in a virtual machine. This is important when creating honeypots, which is a type of isolated environment aimed at luring in attackers in order to study their tools and techniques. In such an isolated system you do not want the attacker to be able to tell that he is in an isolated and potentially surveilled environment. The subject of detecting whether or not a process is running in a virtualized environment is explored in Chapter 5.

2.1.4 VIRTUALIZATION ON X86

Even though x86 is not virtualization-compatible according to Popek and Goldberg [PG74], some CPU models include hardware support to achieve virtualization anyway. Intel call their technology VT-x (Virtualization Technology for x86), while AMD call their technology SVM (Secure Virtual Machine). They work similarly, but we will focus on the details of VT-x.

Some of the difficulties that x86 virtualization faces, and that VT-x aims to solve, are listed below.

Address-space compression: Even though the hypervisor resides in its own address space for the most time, it must allocate a small part of the guest's memory for itself. It is paramount that the guest cannot write to this memory allocated by the hypervisor. To solve this, VT-x allows the virtual address space to be changed at every transition via a new data structure called the Virtual Machine Control Structure (VMCS) which resides entirely in physical memory. By having segment and control registers which handle logical to virtual and virtual to physical memory translations to be saved and restored in a proper way, the problem of address-space compression can be solved.

Non-faulting access to privileged state: When instructions that access registers with CPU state information does not cause a fault to allow the

hypervisor to intervene, the guest may be handed incorrect information. VT-x will cause a VM exit transition to occur when a guest tries to access a privileged state, making sure the correct data is handed to the guest.

Interrupt virtualization: Guests must not be allowed to mask and unmask external interrupts. The hypervisor needs full control of this to, for example, make sure that a keyboard action is not delivered to the guest until it is its turn to run. VT-x includes a special external-interrupt which if enabled hands over execution to the hypervisor when a guest tries to mask or unmask interrupts.

Ring compression: Since the x86 architecture's paging functionality makes no distinction of rings 0, 1 and 2, the guest must be run in ring 3 together with user mode applications. To solve this VT-x makes sure that the kernel can be executed in ring 0 again, while the hypervisor is run in ring -1.

Frequent access to privileged resources: When a guest tries to access privileged resources, a fault must be generated and control handed over to the hypervisor. Since this is a regular occurrence, faulting every access will cause severe performance degradation. By using the VMCS structure provided by VT-x to only invoke the hypervisor when necessary, we can minimize performance degradation.

2.2 MANDATORY ACCESS CONTROL SCHEMES

The most common access control mechanism for operating systems are discretionary in nature, meaning that access permissions/restrictions on an object is defined by the owner of that object. This type of access control is called *Discretionary Access Control* (DAC) [Li11].

Mandatory Access Control (MAC) is another means of access control, which follows a pre-defined policy on how objects and subjects are allowed to interact. The policy commonly applies a rule which denies access to everything for everyone and then have rules for specific cases that must be possible. This makes MACs difficult to manage.

2.2.1 SELINUX

The Mandatory Access Control provided by SELinux is the foundation to one of the implementations in this thesis. SELinux uses the *Linux Security Module* (LSM) in the Linux kernel to achieve mandatory access control.

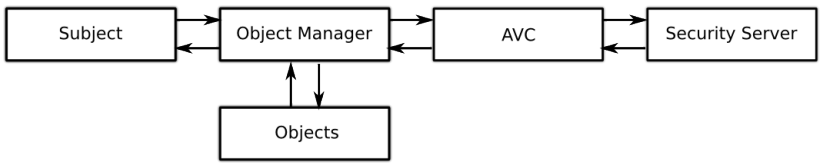

Figure 2.1: The main SELinux components

It should be mentioned that SELinux is not the only solution utilizing LSM. There are several other solutions including AppArmor [VC09], SMACK [Sch08] and Tomoyo Linux [THT05]. We chose to work with SELinux since it is a widely used solution for Mandatory Access Control for Linux systems.

SELinux supports three different forms of MAC; type enforcement, role-based access control and Multi-Level Security. For our purposes in Chapter 6 we will only use type enforcement. With type enforcement we assign security labels to all objects and subjects on the system and create rules based on these.

The main components of SELinux are subjects, objects, object managers, the security server, the Access Vector Cache (AVC) and the security policy. Figure 2.1 shows how these components interact with each other.

When an access request from a subject, e.g., a process requests permission to write to a file, the access request is sent to the object manager dealing with that particular resource. The object manager then queries the *Access Vector Cache* (AVC) to see if the same request was recently made. If that is the case, the object manager receives an answer from the AVC. Otherwise the access request is forwarded to the security server which checks the security policy for a decision. The answer is cached in the AVC for future requests and then forwarded to the object manager which grants or denies access to the subject accordingly.

Before SELinux decides whether an access request should be granted or denied, the DAC is consulted and only if it passes the DAC check will SELinux proceed checking with the security server.

SELinux with type enforcement allows us to confine applications within domains and limit their privileges to the bare minimum necessary to do their job. If the application should later require additional permissions, these have to be specified explicitly in the policy and then loaded in the system.

In a Role-Based Access Control (RBAC) setting, users in a system are assigned roles according to which they have certain access rights. This type of access control enables an organization to efficiently enforce security policies specific for their enterprise. As an example, in a hospital setting, doctors can assume the role with access to patients journals while nurses have no such access. By assigning new employees with their appropriate roles with pre-specified access rights, the overhead of setting specific access rights for new

employees is minimized.

With Multi-Level Security (MLS) individuals are granted clearance for some specific level in order to access objects classified at that level. The levels are 'Top Secret', 'Secret', 'Classified' and 'Unclassified', from most privileged to least privileged. A subject with clearance to 'Top Secret' information will have access to all the other levels as well, while someone with clearance for the 'Secret' level will have access to 'Secret', 'Classified' and 'Unclassified' information, they will not have access to those labeled 'Top Secret'.

2.3 TRUSTED EXECUTION ENVIRONMENTS

The *Trusted Execution Environment* (TEE) is a small part of a system which is considered to be *trustworthy*. The TEE integrity is guaranteed by a subset of the system (possibly the same) that is *trusted* (more formally, a *Trusted Computing Base* [Rus81]). Some notable modern hardware approaches to *Trusted Execution Environment* (TEE) are the Intel Trusted Execution Technology (TXT) (which itself builds upon the Trusted Platform Module) and the ARM TrustZone technology [Gre12] [Gro11a] [td09]. TrustZone is a security extension to some of ARMs processors that provides a cheaper alternative to adding a second physical CPU to gain security through isolation from an untrusted environment.

Intel's Software Guard eXtension (SGX) [AGJS13] provides an extension of the instruction set to allow users to create so called enclaves, which provides private regions of memory that is protected from other processes, even those that run at higher privileges than the target process. Even if a system is compromised, including the kernel, a process running within an enclave would still be isolated from the attacker, in that the data in that enclave would be protected. SGX can be seen as an extension of TrustZone, which provides only one enclave, namely the secure world, and SGX can provide an arbitrary amount of enclaves.

SierraTEE [Sie13a] is a GlobalPlatform compliant TEE implementation that utilizes ARM TruztZone technology to achieve isolation between trusted and untrusted components. SierraTEE borrows a number of design elements from the SierraVisor Hypervisor by the same company [Sie13b].

The ST-Ericsson NovaThor platforms [SE11] make use of various security technologies such as TrustZone to provide a trusted execution environment which communicates with a rich OS using the GlobalPlatform API. Other similar products are MobiCore by Giesecke & Devrient, Trusted Foundations by Trusted Logic and OP-TEE by Linaro.

In [WHS12] Weiss *et al.* discuss a system inspired by the GlobalPlatform TEE which is build upon the Fiasco.OC microkernel and its L4Re runtime

environment [Gro11b].

2.4 OTHER SOLUTIONS

Other interesting solutions that relate to security through compartmentalization is Docker [Mer14] which provides isolation for Linux applications by containing everything that application needs inside a software container, and Wine (Wine Is Not an Emulator) [AJ94] which is a compatibility layer to allow Windows applications to run in a Linux Environment. Wine achieves this by implementing alternatives to the Dynamically Linked Libraries (DLLs) which exist in Windows operating systems in order to run Portable Executable (PE) files under Linux.

3 Blockchain

The blockchain is a recent innovation, dating back to the invention of the cryptographic currency Bitcoin [Nak08]. The blockchain can be seen as an append-only database where entries are costly to modify due the nature of the mechanism of how they are added. This mechanism, called proof-of-work (PoW), and other features provided by the blockchain will be explored in-depth in this chapter. We focus on Bitcoin's blockchain, since it is the most secure blockchain to date given the amount of computational resources utilized to create blocks [btc16]. In Chapter 7 we will utilize Bitcoin's blockchain to achieve Byzantine fault tolerance for the Keyless Signing Infrastructure.

3.1 BACKGROUND

This section aims to provide some basics on cryptographic hash functions and Merkle Trees, since these are basic building blocks used in blockchains. We also cover Byzantine fault tolerance here.

3.1.1 CRYPTOGRAPHIC HASH FUNCTIONS

A cryptographic hash function is a one-way function that takes data of arbitrary length as input, also referred as a message, and outputs a string of fixed length, referred to as hash digest. Pragmatically speaking one should not be able to find two different messages that have the same hash digest as their output. It should also not be possible to retrieve the message given the hash digest, hence a one-way function.

In order for a cryptographic hash function to be deemed secure, it will need

to have the following three properties:

Pre-image resistance Given hash digest `x`, it must be difficult to find a message, `m`, such that `h(m) = x`.

Second pre-image resistance Given message `m`, it must be difficult to find a message, `m'` where `m' != m`, such that `h(m) = h(m')`.

Collision resistance It must be difficult to find any two messages, `m` and `m'`, where `m' != m`, such that `h(m) = h(m')`.

3.1.2 MERKLE TREES

In a Merkle tree [Mer80] every node is a hash digest of its concatenated children. This structure allows for efficient verification of data inclusion given only the root hash and the intermediate hash values along the path to the data item to be proven to be a part of the tree. Proving that a leaf node is part of the tree has logarithmic complexity in both time and space, providing high scalability. Figure 3.1 illustrates a Merkle tree and the necessary data to verify inclusion of data.

3.1.3 BYZANTINE FAULT TOLERANCE

A Byzantine fault tolerant system can properly function even when Byzantine failures are happening. A byzantine failure [DHP+04] [DHSZ03] occurs when the system fails in arbitrary ways, e.g., when some part of the system is processing requests incorrectly and corrupts the local state. A Byzantine fault tolerant system will continue to function given a certain threshold of byzantine failures is not breached.

Byzantine fault tolerance is a sub-field of error tolerance based on the two generals problem, or the more general Byzantine generals problem [LSP82], in which a number of generals plan to attack a city and must reach consensus with everyone else on the time to attack. If too few generals attack at the same time, the attack will fail. The problem is provably unsolvable [AEH75] and solutions aim at making the consensus-making process as difficult and as costly as possible to disrupt. In the next section we will see how proof-of-work for blockchain systems provides Byzantine fault tolerance.

3.2 BLOCKCHAIN AND PROOF-OF-WORK

A blockchain can be viewed as a database where data entered into it has strong guarantees of immutability and where these guarantees grow stronger as more data is appended. These guarantees are what makes a blockchain,

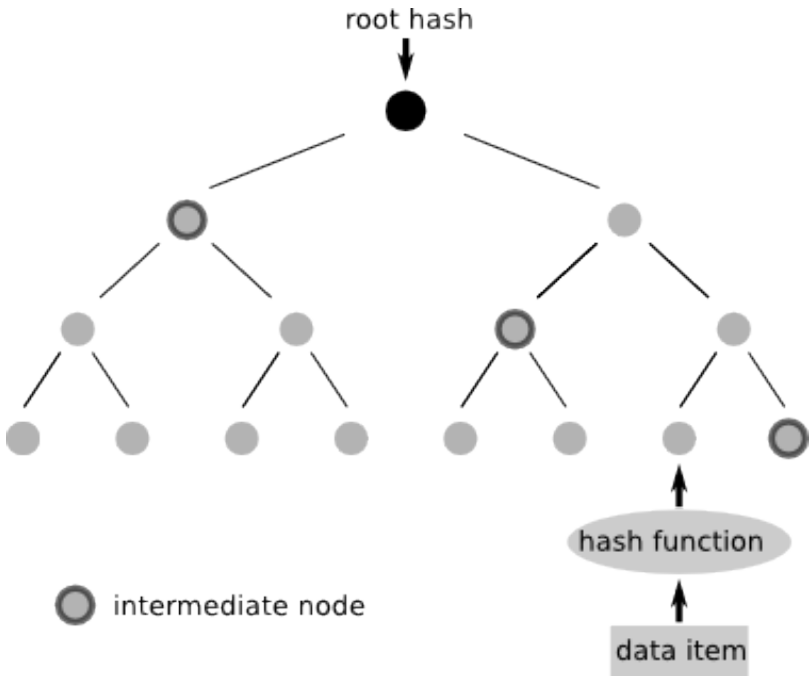

Figure 3.1: Given the root hash and intermediate node hashes, anyone can verify the particular data item was included in the tree. By computing the hash digest of the data item and pairing that digest with its sibling, intermediate hash and continuing in this fashion until pairing the final intermediate hash with the currently held hash digest and verifying that the hash digest of those two nodes equals the root hash, we have effectively verified that the data item was included in the Merkle tree.

among other things, ideal for timestamping applications, because they provide assurances that any attempt to change the data will be detected.

A blockchain consists of linked blocks, which in turn contain transactions. A block contains a block header and a set of transactions as its payload. The set of transactions are commited in the block header via the Merkle tree hash root of all transactions as leaves. Additionally the header contains a Unix timestamp, defined as the number of seconds elapsed since 00:00:00 UTC, 1 January 1970, and the hash of the previous blocks' header. This makes sure that each new block commits to all previous ones as well, thus forming a chain of blocks, where the order of them cannot be manipulated without being detected.

The blockchain is secured with proof-of-work, where the block header is repeatedly hashed until the hash value fulfills some condition. The condition is that the value of the hash digest, interpreted as an integer, must be below some target value, also specified in the block header. This form of block signing allows anyone to try to create a block without discrimination, because all that is needed is access to resources that can perform these hashing operations. The target value is regularly adjusted such that the average time between the issuance of two blocks is fixed. The security guarantees are derived from the assumption that if a majority of the computational power is honest and always builds on the chain with most work, then an attacker trying to modify data in older blocks will not be able to accumulate enough work to become the chain everyone else recognizes as the correct one.

Proof-of-work makes modification of data in existing blocks expensive since the hash digest of the block header must be recalculated to satisfy the condition. Additionally, every subsequent block header will need to be recalculated since they depend on the previous block header, going all the way back to the block that was modified. Unless the attacker possesses more than half of the computational power of the entire network, he is unlikely to be able to catch up and create a longer chain. The deeper the data to be modified is, the more expensive it will be. It is the chain of proof-of-work that helps overcome Byzantine failures by allowing participants of the system to share an identical state of the database.

Since bitcoin tokens are issued to the creator of a block, also called a miner, the incentives are aligned with extending the chain with most work attached to it. Otherwise there is a risk that the block will never be part of the chain, and the block creator cannot claim the tokens rewarded in that block. There are some exceptions to this rule where a single entity possessing a large enough share of the hashing power of the network can benefit from withholding valid blocks in an attack called selfish mining [ES14] [SSZ15].

A fork occurs when one block in the blockchain has two parent nodes. This is a common occurrence where two miners find a valid block simultaneously. In such a case, when another block is found the rest of the network will start working on top of the block they hear about first. When another block is found, all other miners working on the alternative block should abandon it and continue working on top of the block in the longest chain. The valid chain is the one with most work attached to it. When a different fork overtakes another one and replaces it, blocks no longer part of the chain will be orphaned. The process of rearranging the blockchain, undoing transactions in orphaned blocks and so on, is referred to as a chain reorganization and is performed by nodes that followed the particular fork. There are no guarantees that a transaction included in an orphaned block is included in the new fork. As such, relying on important data in blocks with only a few blocks on top of it

have lower guarantees of staying in the chain.

Proof-of-work incentivises miners to act honestly since it costs them resources to create blocks. Proof-of-work is not the only thing necessary to create valid blocks though. Apart from the proof-of-work, several other consensus rules exist which must be followed for blocks to be considered valid. The most important consensus rule is arguably that a specific token must not be spent more than once. This rule against double-spending allows deterministic tracking of transaction outputs in a transaction chain, which will be useful to track hash commitments embedded by the Keyless Signing Infrastructure, described in more detail in Chapter 7.

Another important consensus rule is that the timestamp in the block header must not be ahead more than 2 hours of the time of the local machine that receives it, nor must it be earlier than the median timestamp of the previous 11 blocks. Should the timestamp not be within these boundaries, the block is deemed invalid by the receiving peer. This provides rough assurances of what time the data in the block were commited at. The reason for having such wide ranges acceptable is that it is difficult to synchronize time with unknown, potentially malicious peers.

Other consensus rules include a limit of 1MB of transaction data and halving the reward in each block each 210,000 blocks. There are many more rules, many of which are bugs that need to be followed in order to not drop out of consensus with the other peers. These bugs were introduced when Bitcoin bootstrapped, and if they are fixed it would cause old software to become incompatible. One example of such a bug can be seen in Section 3.2.1, where an extra, unnecessary, element on the stack have to be added. Fixing this bug would make older clients incompatible which would cause a permanent split of the blockchain.

3.2.1 TRANSACTIONS

A transaction specifies how tokens in the blockchain get reallocated. A transaction must include one or more inputs and outputs. Each input will reference an output with unspent tokens and each output will specify conditions for how to spend the amount of tokens locked in it. The amount in the outputs referenced by the inputs must be equal to or more than the amount set in the outputs of the new transaction. If the outputs spend less than what is available, the difference can be collected as a fee by the miner that includes the transaction in a block. Fees are essential to get transactions included in a block. Since any entity can begin mining, i.e., attempt to create a new valid block, it is also up to them what transactions they include in a block. If we assume there will be more transactions available than can fit into a block at any given time, the rational miner would look to maximize profits by including

the transactions that pays the highest fees.

To decide whether a transaction is eligible for inclusion in a block, inputs and outputs include scripts which are executed and must return `TRUE` in order to be valid. When a new transaction is created, each input script will be paired with the referenced outputs script. Only transactions that can provide a valid input script will be able to spend that specific output. Additionally, there is a locktime field in the transaction, the value of which must be below the median timestamp of the previous eleven blocks in order to be valid. If the value of the locktime field is below 500,000,000 it is interpreted as a block height, i.e., the number of blocks created in total, and the transaction must not be included in any block prior to the height specified. If the value is above 500,000,000, it is regarded as a Unix timestamp and the transaction must not be included in a block unless the median timestamp of the previous eleven blocks are above the locktime value.

The language Bitcoin transactions utilizes is called Script. Script is stack-based with a list of instructions and data items processed from left to right. If at any point during execution a verify operation sees a `FALSE` value at the top of the stack, the execution stops and the transaction is marked as invalid. When the transaction script executes, the top stack value must not be `FALSE` for the transaction itself to be valid. Below are some examples of some standardized ways scripts are currently being used in Bitcoin to give the reader, both an idea of how scripts are executed, and what constructs we will use later on.

PAY-TO-PUBLIC-KEY-HASH (P2PKH)

In a P2PKH type of transaction, the script in the output to be spent is formed to force the spender to supply the raw public key in the input script to be hashed and matched to the hash digest supplied in the output script. Should the hashes match, the script goes on to validate a signature of the transaction with that public key. The scripts look like this:

```
output script:  OP_DUP
                OP_HASH160
                <hash_of_public_key>
                OP_EQUALVERIFY
                OP_CHECKSIG

input script:   <signature>
                <public_key>
```

The following happens when this script executes.

- The `<signature>` and `<public_key>` are pushed to the stack from the input script
- The `OP_DUP` instruction duplicates the top stack value, i.e., `public_key`, so there are now two instances of `public_key` on the top of the stack
- `OP_HASH` pops the top stack value, hashes it and pushes the hash to the stack
- `<hash_of_public_key>` is pushed to the stack
- The two top values of the stack are compared to make sure they are equal via the `OP_EQUALVERIFY` instruction
- If they are not equal, execution stops and the transaction is deemed invalid
- If they are equal, execution continues and `OP_CHECKSIG` will try to verify that `<public_key>` validates `<signature>` correctly
- If it is valid, the transaction is valid and it may be included in a block

PAY-TO-SCRIPT-HASH (P2SH)

In the same way that a P2PKH requires a public key to hash to a specific hash digest, P2SH requires an entire serialized script to be hashed to match the specific digest in the output. This provides much more flexibility for the receiver to specify the conditions he wants in his script without burdening the sender with the details. The serialized script that must be provided to spend a P2SH output is called a redeem script and is executed if the hashes match. It also has the advantage of script confidentiality until it is spent.

Scripts presented later will be embedded in P2SH transactions.

MULTISIGNATURE TRANSACTIONS

Multisignature encumbered outputs forces the spender to supply M valid signatures from N given public keys, where $M \leqslant N$. This allows the security of that output to be distributed among many keys, where loss or theft of one of the keys does not necessarily mean that the output is compromised, i.e., the output cannot be spent with only one key. Below is an example multisignature script.

```
output script:  OP_2
                <public_key1>
```

```
                <public_key2>
                <public_key3>
                OP_3
                OP_CHECKMULTISIG

input script:   OP_0
                <signature1>
                <signature3>
```

The following happens when this script executes

- The data items from the input script and the output script are pushed to the stack
- `OP_CHECKMULTISIG` is then executed which takes the top stack value, `OP_3`, to know how many public keys to pop
- After the public keys are popped off the stack, the value `OP_2` is popped and tells the instruction how many signatures it will need
- `OP_CHECKMULTISIG` tries all combinations of the public keys to verify all signatures and will push `OP_TRUE` to the stack if it finds enough valid signatures, otherwise it will push `OP_FALSE`
- `OP_0` is there due to a bug in `OP_CHECKMULTISIG`, where it always pops one extra value from the stack

OP_CHECKLOCKTIMEVERIFY

`OP_CHECKLOCKTIMEVERIFY` can be used to make the locktime field of transactions much more powerful. Instead of just pre-signing transactions with the locktime requirements, an output can include a locktime requirement on the stack together with the instruction `OP_CHECKLOCKTIMEVERIFY`, which will only mark a transaction as valid if the locktime field of the transaction trying to spend the output is above the locktime requirement specified in the output script. We will utilize this instruction and multisignature scripts to create spending conditions that prioritizes some groups in Section 7.2.2.

OP_RETURN

`OP_RETURN` is an opcode that fails the script execution immediately and makes any attempt to spend such an output invalidate the transaction. In practice it makes the specific output unspendable, and this is used when you want a hash value embedded in the blockchain, or for some reason burn tokens. The

reason for using `OP_RETURN` is that it marks that output as unspendable and allows clients to stop tracking it, thus not requiring unnecessary resources for running a client. These types of outputs can also be pruned and deleted from permanent storage in some modes of operation to save disk space. Since it is unspendable, the associated value with the output should be zero.

`OP_RETURN` is the way we will embed CRHs in the blockchain.

3.3 USE CASES OF BLOCKCHAIN TECHNOLOGY

With the invention of Bitcoin and the blockchain [Nak08], blockchains have been proposed as a solution for many of today's problems relying on trust. From reforming the financial system by taking advantage of the immutability and transparency provided by blockchains, to tracking ownership of digital assets [Gro16] as well as physical property, so called smart property [sza97]. There have been proposals for protocols to operate directly on top of existing blockchains for entirely new purposes than what they were originally intended for, such as Counterparty [cou], which is a platform for P2P financial applications on top of the Bitcoin blockchain, and Colored Coins [Ros12] which is a technique for associating assets with addresses, also on the Bitcoin blockchain.

Brand new blockchains have also been bootstrapped to fill the need of these new solutions, like Namecoin [nam] as a censorship-resistant alternative to the DNS system, and Ethereum [Woo14] which provides feature-rich scripting capabilities to do smart contracting.

The blockchain is finding new use cases every day, which shows the versatility of the system. Below are some other uses for the blockchain in a little more detail.

3.3.1 DECENTRALIZED DNS

Namecoin is an interesting example, and one of the first alternative chains created next to Bitcoin, that still utilizes and receives security from the computational power available to the Bitcoin network. So called merged-mining allows Bitcoin-miners to simultaneously mine blocks on the Namecoinchain by committing to the state of it in a Bitcoin block. This proved to have some serious security issues when it showed that not all Bitcoin miners were mining Namecoin blocks, and the centralization of computational power was even worse with one entity having control of more than half of the computational power in Namecoin, theoretically allowing them to perform 51% attacks, where rewriting the history of blocks is possible in order to reverse or censor a specific transactions. It was especially bad with merge-mining since the pool doing merge-mining would not lose any income on Bitcoin's blockchain,

making the attack essentially free. Attacking merge-mined blockchains have occurred previously where a pool attacked a cryptocurrency called Coiled-Coin [att12], because the pool operator felt it was a scam.

Alternatives to Namecoin have popped up, the most notable being Blockstack [ANSF16]. Blockstack provides secure and decentralized naming and identity services. They also utilize the Bitcoin blockchain, but in a more resilient way compared to Namecoin. Name operations on Blockstack are embedded in underlying bitcoin transactions that are marked as Blockstack transactions so that they can be easily identified by other Blockstack nodes. Unlike Namecoin, which stores all records directly on the blockchain, Blockstack will store such data in other ways and only provide commitments to that data in the blockchain, making that approach more scalable.

3.3.2 SMART CONTRACT EXPRESSIVENESS AND SIDECHAINS

Ethereum raised the question on whether the scripting language for doing smart contracts can be too expressive, as the Decentralized Autonomous Organization (DAO) [Jen16] was released as a very complex piece of smart contract, and even after multiple security reviews was exploited [dC16] due to a missed bug in the code allowing an attacker to siphon of tokens worth tens of millions of dollars. The DAO was supposed to allow a diverse set of people to pool their capital and fund various projects through voting proportional to their stake.

Decentralizing prediction markets with the help of blockchains have also been a topic of research [CBF+14]. Two of the most interesting projects in this area is Augur [PK15] which operates on top of the Ethereum blockchain, and Hivemind [Szt15] which aims to launch a prediction market as a so called Bitcoin sidechain. The concept of a sidechain [BCD+14] was presented by a company called Blockstream and essentially allows feature extensions to be added to a different blockchain, but will still utilize the underlying token of a different blockchain. In that way bitcoins can be locked on Bitcoin's blockchain and be redeemed/created on the sidechain and will only be unlocked on the main chain if the corresponding coins on the sidechain are locked/destroyed with proof of that provided in the main chain. Experimental features can be implemented on the sidechain with hard guarantees that if something goes wrong, only the sidechain will suffer and those that do not opt to use that sidechain will stay unaffected.

Another interesting sidechain project is the Rootstock [DL15] sidechain, which plans to create a sidechain with the same functionality as Ethereum, but which uses the bitcoin currency instead of a newly created one. In this way it can utilize the network effects of the bitcoin token, provide a more expressive smart contracting language while simultaneously not having to

expose the underlying Bitcoin blockchain to new experimental features.

SECURITY OF SIDECHAINS

Initially the idea was to secure sidechains by utilizing merge-mining, where Bitcoin miners can add their computational power to secure the sidechain at no extra cost to them. As we discussed in Section 3.3.1, merge-mining could provide less security to the chain if not a significant majority of the existing miners perform the merge-mining, leaving the sidechain open to 51%-type attacks.

On the opposite side, there are sidechains that are secured by a federation of signers. In this configuration blocks are valid if a certain threshold of predefined signers have signed the block, making this type of sidechain inherently more centralized because of the fixed set of signers.

In between merged-mined and federated sidechains are a combination of the two, where a federation secures the chain during bootstrap and as more miners starts securing the sidechain, the influence of the federation decreases. When, or rather if, all Bitcoin miners support the sidechain, the federation will have lost its power of the sidechain. Similarly if the hashing power on the sidechain decreases, the federation will regain power.

3.4 SECURITY OF BLOCKCHAINS

A blockchain makes sense if it is widely distributed and used among a diverse set of people. By having a large set utilizing the blockchain, they are contributing to the redundancy of where the data is stored. It also makes that particular platform more valuable because of its usage, as its network effects are larger. From this we can say that fragmentation, i.e., multiple blockchains, will reduce the effective security of all of them. If it costs 100 million dollars to attack a blockchain in an environment where there is only one, it would only cost one million dollars to attack a blockchain in an environment where there are 100 blockchains each with the same amount of computational power. It is the computational power behind a blockchain that provides it with its security, and fragmenting that to multiple blockchains makes as much sense as dividing a password into substrings and hashing them separately [Mic].

Bitcoin is the single most secure blockchain to date, with an overwhelming [bit16b] 80% of the market capitalization of all blockchains at the time of writing. Market capitalization and security are closely linked due to the mining game being fiercely competitive and the cost of creating a bitcoin is roughly its market value. Thus, the amount of energy expelled to create a block will, on average, equal the amount rewarded. Basic economics tells

us that should the price of one token increase by 100%, then more hashing power would enter the network to reap some of those profits until the network reaches an equilibrium where the cost to create that token is equal to its value.

The computational power in cryptocurrencies that rely on proof-of-work can come from many different types of hardware. CPUs, GPUs, FPGAs and ASICs. When Bitcoin first bootstrapped, the reference client included mining functionality that utilized the CPU. However, the CPU was very limited in the amount of security it added to the network compared to how much energy it used. This was not very efficient, and soon software that utilized GPUs made its entrance, giving the security of the network a real boost. The end of the line is of course Application Specific Integrated Circuits (ASIC) where hardware was specifically designed for the sole purpose of mining for bitcoins. The use of ASICs has received a lot of criticism, with arguments being that it is not something everyone has in their own home and thus hurts the decentralization of the network. Any reasonable proof-of-work scheme is however always subject to be implemented in specialized hardware at some point. Thus, rather than rejecting it, it should be embraced. Any manufacturer of such hardware has an incentive not to give itself too much computational power, as the confidence in the network would decrease if a single entity had too much power. It could be difficult to properly identify the creator of blocks since no identity is required to start mining and a brand new identity is created for each new block.

Another interesting point in favor of ASICs is that botnets cannot affect the network as much. If ASICs are not available, then a botnet with hundreds of thousands of machines could effectively attack the network at no cost to the attacker [PGP12]. The most common approach to restricting ASICs on alternative blockchains is to make the proof-of-work algorithm memory hard, thus making ASICs less effective. The problem with this approach is that making the algorithm more complex increases the likelihood of there being potential for optimizations. If there is room for optimization, then the ones with most resources are likely to be first to discover them, and keep that optimization to themselves. This would make others who do not apply that optimization to run at a loss, because the optimized miner will be more profitable and eventually price out their competitors. Bitcoin is using double sha256 hashes as its proof-of-work algorithm. Hardware specs for it is freely available online and the algorithm is simple, making any attempt to implement it into ASICs a relatively low-cost affair compared to more complicated alternatives. Even a simple hashing algorithm such as sha256 is not free from optimizations though, as AsicBoost [Han16] was proposed and patented to provide miners with optimization techniques with up to 10% improvement.

4

Software Security

Software security encompasses measures taken to secure code from flaws, both intentional and with malicious intent in mind. It is a broad field dealing with, among other things, validation of user input, authentication, cryptography and much more.

In this chapter we will discuss some topics to aid in understanding Chapters 8,9 and 10 which are based on the publications titled »A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries,« »Exploiting trust in Deterministic Builds« and »eavesROP: Listening for ROP Payloads in Data Streams,« respectively.

4.1 THE X86 ARCHITECTURE

The x86 architecture pioneered by Intel remains one of the most widely used Instruction Set Architectures (ISA) today. It is a Complex Instruction Set Computing (CISC) architecture meaning that it provides instructions that themselves may execute several low-level operations as one. The alternative to CISC is Reduced Instruction Set Computing (RISC), where instructions normally only perform one low-level operation. Most implementations of the RISC architecture have fixed-length instructions whereas in the x86 architecture instructions can be between 1 and 15 bytes [int]. A consequence of this is that instructions in a RISC architecture are executed on a fixed alignment, whereas x86 instructions can be executed from any byte alignment, with the possibility of executing an instruction from a byte never meant to be the starting byte of an instruction, also denoted as unintended or overlapping instructions.

4.1.1 ANATOMY OF AN X86 INSTRUCTION

An x86 instruction, as illustrated in Figure 4.1, is divided into six fields where the opcode is the only mandatory field for all instructions. Below follows a short description of these fields.

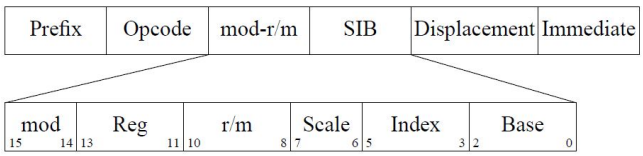

Figure 4.1: Illustration of a x86 instruction

An instruction can have up to four prefixes where each **Prefix** is one byte in length. A prefix changes the behavior of the instruction it is applied to, for example changing or overriding the operand or address size.

The **Opcode** is the instruction code that defines the main behavior of the instruction. The most common opcodes have a length of one byte, but using the expansion code, `0x0f`, the opcode can become two bytes. There are additional extensions that can make the length three bytes in total. For one-byte instructions the opcode includes 3 bits that specifies which register to perform the specific operation on.

The **mod-r/m** byte defines the addressing mode and operands of the instruction. It is divided into the 2-bit `mod` field and two 3-bit fields called `Reg` and `r/m`. The `mod` field specifies direct or indirect addressing, i.e., directly register to register or if one of the registers are to be dereferenced. The `mod` field also specifies if there is a displacement field or `SIB`-byte for this instruction. Table 4.1 shows how the `reg` and `r/m` fields are decoded and Table 4.2 shows how the `mod` field is decoded.

The **Scale-Index-Base** byte (`SIB`-byte) is used for indexed addressing, for example in arrays. It is illustrated as
`[Base + Index*Scale]` where `Base` and `Index` are registers and `Scale` can assume the value 1, 2, 4 or 8.

The **Displacement** field specifies an offset for a memory dereferencing instruction. It can also specify an absolute address to be used by the instruction. The length of the address displacement field can be 0, 1, 2 or 4 bytes.

The **Immediate** field contains any constants used in instructions and can be of 0, 1, 2 or 4 bytes.

4.1.2 NO-OPERATION INSTRUCTIONS

No-operation (`NOP`) instructions can have many uses, despite the fact that the only effect it has on the CPU state is the update of the program counter. The

most common use of the NOP instruction is for memory alignment of machine code, for the purpose of more efficient instruction handling by the CPU. Modern x86 instructions, for instance, fetches instructions at DWORD boundaries. If the target instruction of a branch would be in the middle of a DWORD, then the CPU would fetch the preceding WORD as well in addition to the target instruction. Another use is to prevent hazards in the CPU pipeline.

The most commonly used NOP instruction for the x86 architecture is encoded within a single byte, 0x90, and it is an alias for the instruction XCHG EAX,EAX, which simply switches the values between the registers in the two operands.

Compiled binaries often have multiple single-byte NOPs after each other to achieve memory alignment. If these NOPs are executed, they will take one clock cycle each to execute. An alternative solution would be to replace multiple single-byte NOPs with one multi-byte NOP instruction.

NOP instructions on the x86 architecture should, according to Intel's x86 manual [Int13], vary between one and nine bytes. In reality though, we can construct valid NOP instructions up to the maximum instruction size, 15 bytes, by using multiple instruction prefixes. Below, the recommended encoding of each multi-byte NOP instruction can be seen.

```
Instruction                          Encoding
NOP                                  66 90
NOP DWORD PTR [EAX]                  0F 1F 00
NOP DWORD PTR [EAX+00]               0F 1F 40 00
NOP DWORD PTR [EAX+EAX + 00]         0F 1F 44 00 00
NOP WORD PTR [EAX+EAX + 00]          66 0F 1F 44 00 00
NOP DWORD PTR [EAX+00000000]         0F 1F 80 00 00 00 00
NOP DWORD PTR [EAX+EAX + 00000000]   0F 1F 84 00 00 00 00 00
```

Table 4.1: Decoding what register is used in the reg and r/m fields for various data-sizes used. Also applicable to some other instructions that encodes this into the opcode.

Value	32-bit	16-bit	8-bit
000	EAX	AX	AL
010	ECX	CX	CL
011	EDX	DX	DL
100	ESP	SP	AH
101	EBP	BP	CH
110	ESI	SI	DH
111	EDI	DI	BH

Table 4.2: Interpretation of the mod field

00	Indirect addressing mode, meaning that the register specified in `r/m` is interpreted as an address and the contents in that address is fetched. There are some exceptions here: 1. When `r/m` is 100, the processor will switch to SIB mode, and 2. When `r/m` is 101 the CPU switches to 32-bit displacement mode where the displacement is interpreted as an absolute address, and not only an offset.
01	Same as with 00, but with 8-bit displacement added to the value before dereferencing
10	Same as with 01, but with 32-bit displacement added to the value before dereferencing
11	Direct addressing mode where the instruction operates on two registers directly without any memory dereferencing

```
NOP WORD PTR [EAX+EAX + 00000000]   66 0F 1F 84 00 00 00 00 00
```

For the nine-byte `NOP`, the first byte (`66`) is an instruction prefix for overriding the operand-size. The following two bytes (`0F 1F`) is the opcode. The fourth byte (`84`) is the mod `r/m` byte. The last five bytes (`00 00 00 00 00`) describe the memory operand. Note that even though the `NOP` has a memory operand, when executed it does not access that memory in any way. This is simply how the `NOP` is represented in assembly code.

Since no memory is accessed, the last five bytes can be set however we want and this would have no effect on how the instruction is executed compared to its recommended configuration. In Chapter 8 we use this behavior to create custom overlapping instructions for malicious purposes. Overlapping instructions are described next.

The use of `NOP` instructions for unconventional purposes has been shown to have many applications as well. The most famous example may be the `NOP` sled used to ease the exploitation of buffer overflow vulnerabilities [One96]. Another example is when the insertion of `NOP` instructions in strategic places in a malicious program's executable code could prevent AV-software from detecting it [CJ03].

4.1.3 OVERLAPPING INSTRUCTIONS

Overlapping instructions exist in all compiled x86 code. The variable-length instructions accommodates different interpretations of the code depending on what byte decoding starts from. An example of this phenomenon is shown

below where the `jmp` instruction will jump to the latter half (`0xff`) of its own instruction and execute from that point.

```
eb ff       jmp -1
c0 c3 00    rol bl, 0x0
;The jump will execute
ff c0       inc eax
c3          retn
```

This can be used in application exploitation attacks based on return-oriented programming (ROP) discussed in Section 4.3.3. In order to find more useful instructions for the gadgets used in this attack, unintended instructions can be discovered by looking for appropriate overlapping instructions by trying to decode instructions at different byte offsets.

4.2 REVERSE CODE ENGINEERING

Reverse engineering software is the process of analyzing a binary to deduce what functionality it provides. Analysis can occur through observation of the execution of the program through a debugger, disassembly of the raw machine code or even decompilation of the machine code which attempts to translate the assembly code into a high-level language that is easier for humans to analyze.

4.2.1 DISASSEMBLY

Disassembly is the process of taking machine code as input, and outputting human-readable assembly code. There are two main ways of approaching the problem of disassembly, namely *static* and *dynamic disassembly*. Static disassembly consists of analyzing and examining the machine code in order to construct the most probable sequence of assembly instructions. This is done without actually executing the code, which is in contrast to dynamic disassembly, where the instructions are executed and identified upon execution. Both methods have their advantages and drawbacks. Static disassembly must guess which instructions are executed, but will on the other hand be able to cover all code. Dynamic disassembly, on the other hand, will know which instructions are executed, but it will only identify those instructions that are actually executed using the current input and environment.

Static disassembly can in turn be performed using a *linear sweep algorithm* or *recursive traversal*. A linear sweep algorithm starts with the first executable byte and then proceeds through the machine code, disassembling instruction by instruction. One drawback of this method is that the disassembler can not easily distinguish data from instructions in the code section of the executable.

This can lead to errors if data is located in a stream of instructions. Another limitation is when one instruction ends, the disassembler assumes that the next instruction follows it immediately. In recursive traversal, the actual control flow of the program is followed. If it encounters an unconditional jump instruction, the disassembly proceeds at the target address. This will allow the algorithm to avoid errors based on data that is embedded in the code section. On the other hand, it may not always be easy, or even possible, to compute the correct target of a jump instruction.

4.2.2 ANTI-DISASSEMBLY

Anti-disassembly is the process of deliberately trying to make it more difficult to disassemble machine code. Several techniques have been proposed in the literature [AdJ06] [SH12] [ACdC09] and the most basic ideas are typically to take advantage of the limitations in linear sweep algorithms and/or recursive traversal. The fact that the linear sweep algorithm disassembles each instruction in a sequence, regardless of the control flow, can be exploited by adding junk bytes after an unconditional jump. These junk bytes will never be executed, but the algorithm will assume that they are part of the next instruction, resulting in a misalignment between the executed code and the disassembly listing. As long as the junk bytes constitute the beginning of an instruction and the following instruction starts after the end of the junk byte instruction, the disassembler will start producing the wrong code.

The following example will insert a junk byte with a value `0x9A` in order to confuse a linear sweep disassembler.

```
JMP foo         EB 03
DB 0x9A         9A   #This is the junk byte
foo:
XOR EAX,EAX     31 C0
XOR EBX,EBX     31 C3
INC EAX         40
INC EBX         43
INT 0x80        CD 80
```

This piece of code would be disassembled in the following way.

```
JMP foo+1                       EB 03
foo:
CALL DWORD 0x4340:0xC331C031    9A 31 C0 31 C3 40 43
INT 0x80                        CD 80
```

One limitation of using junk bytes as an anti-disassembly technique, is that the disassembly will resynchronize with the real code after a small number of

steps since x86 assembly instructions do not have fixed size instructions. This can be seen in the above example where the instruction `INT 0x80` is found in both disassembly listings.

One common technique to fool recursive disassemblers is to use *opaque predicates*. By having a conditional jump instruction where the condition is the same every time a program is executed, the conditional jump works effectively like an unconditional jump if the jump is taken, or a `NOP` instruction if the jump is not taken. Recursive disassemblers will often be fooled to follow the incorrect execution flow while disassembling. Below is an example of an opaque predicate.

```
MOV EAX, 0x1
TEST EAX, EAX
JZ foo+1
foo:
CALL something
```

Register EAX will always contain the value 1, and the conditional jump instruction will never be taken, but a recursive traversal disassembler will first evaluate the target of the branch instruction, starting with the second byte of the `CALL` instruction, and when evaluating the fallthrough instruction it will notice that the bytes are already disassembled and will not include this in its output.

To fool a dynamic disassembler the jump instruction must be able to evaluate to both true and false, depending on the circumstances. For instance if the executable finds itself running within a virtual machine, it may decide to jump to a piece of code with benign functionality, otherwise the malicious content will be executed.

There have been much work involving anti-disassembly techniques. Some of the more innovative ideas involves generating instructions during run-time based on system information so that different instructions are generated depending on the system it is executed on. These instructions are generated with cryptographic hash functions in [AdJ06] and pseudo-random number generators in [ACdC09]. The idea of embedding hidden instructions within a larger instruction is described in [LSPM12]. The first mention of overlapping instructions as a means to complicate disassembly was first described in [Coh92]. Special cases of overlapping instructions has been used as a way to increase tamper-resistance of binaries [JJV07]. It was also mentioned in a dissertation [Kin10] as a problem when developing binary-analysis techniques and tools. Overlapping instructions also have a use in return-oriented [Sha07b] and jump-oriented [BJFL11c] programming scenarios, where a greater amount of gadgets (small snippets of instructions) can be found using the technique.

Especially in jump-oriented programming is overlapping instructions beneficial, because the necessary types of instructions needed are scarce, but the op-code byte for it is rich.

Improvement of disassembly methods, like the differentiation of data or junk-bytes from executable code, has also been considered in [WZH+11].

4.3 STATE OF SOFTWARE SECURITY

There is a perpetual cat-and-mouse game going on between attackers and defenders. As soon as a new attack is discovered, defenders will rush to propose countermeasures to stop that attack, and when an effective countermeasure is deployed, attackers will find new ways to circumvent those defenses.

4.3.1 BACKDOORS

Throughout the years there have been numerous attempts by adversaries to plant backdoors in software projects [Edg10] [bc03] [Eva11] [Sec10] [wel13]. The main goal of backdoors is to gain unauthorized access by circumventing the authentication step or simply by gaining access to a system remotely. These are often very subtle modifications of the source code that can easily go unnoticed by code reviewers or static analysis tools. Developers often rely on static tools that inspect the source code for any programming flaws [Cov] [Fla] [Spl]. These tools have a difficulty of identifying logical flaws and manual review must be conducted to identify potential backdoors. Reviewers rely on manual reading of code, supported by checklists and coding standards. This consists of identifying certain unsafe functions that can be the cause of security vulnerabilities or if input is being sanitized [How06]. This type of scan for security vulnerabilities may not be sufficient for identifying potential backdoors, instead the reviewer may have to read and understand each line of code in a project which is time-consuming and costly. The more mature open source projects have adopted peer reviewing as an important quality assurance [AJ07] [P.C11]. In some cases a code commit must first be reviewed before it is accepted. Depending on how trusted the developer is, i.e., the better reputation he/she has, it is more likely that the developer will get code changes accepted [BC14]. There is also a lack of extensive results and research on how peer reviewing reduces security vulnerabilities and backdoors in open source community. Still there is some research initiatives addressing these questions that have been initiated [BC13].

A backdoor can be seen as trigger-based code which is executed when specific inputs are received, denoted as trigger conditions. Discovering the trigger conditions can be difficult [ZW11] [SLGL08] to automated analysis. The actual problem of identifying trigger-based code has made recent advances,

introducing tools for automatic detection of backdoors [SH13] for detecting rarely exercised code paths. As a response to this, Andriesse et al. [AB14] show how one can hide the backdoor via instruction-level steganography by modifying the final binary.

In order to thwart tampering of binaries, the code can be signed by the compiler or the person in hand of publishing the binary. Lately some initiatives have been taken to introduce secure software distribution that would enable trusted binaries to be downloaded and verified by multiple users. This is denoted as deterministic, reproducible or verifiable builds [Git] [Deb16] [Tor15] in which the build process will generate identical binaries. Whenever users download the source code or binaries from different sources, they will be able to verify that the same binary is used by others. This binary transparency could protect against malicious distribution of binaries which could include malicious code, either by planting it in third-party distributions or tampering with the binary. The problem with this approach is based on the fact that a trusted developer could potentially inject malicious code in the official code repositories before the source code is actually released, therefore circumventing the trust model of deterministic builds.

4.3.2 DETERMINISTIC BUILDS

Distributing compiled binaries of open source software does not guarantee that the binaries are in fact compiled from the referenced source code. There is nothing stopping an attacker from adding malicious code, producing the binaries from the modified source code and claim that it was the result of compilation of the unmodified source code. This poses security issues for people who do not have the knowledge or do not want to compile the source code themselves. They basically have to put their faith into the single entity who compiled the binary that they did not inject malicious code into it.

Deterministically built binaries allow multiple builders to produce the same byte-by-byte binary such that a hash value of the binary is the same for all builders. This removes the single point of failure whereby trust needs to be put into a single builder. Instead, trust is now distributed between multiple builders, whereby each builder individually publishes a signature of the binary. This allows anyone who wishes to install the binary directly to verify that multiple builders have compiled the binaries from the referenced source code and that there are no discrepancies between the hashes of the different builders. Unless all builders are conspiring or are all controlled by another attacker, the binary can be deemed safe from manipulation before compilation.

Examples of security critical applications that use deterministic building for their binaries are the Tor projects Tor Browser Bundle [Tor15] and the reference implementation of the cryptocurrency Bitcoin [bit16a]. The operating

system Debian is also aiming for a full deterministic build and the alternative software repository for Android applications, F-Droid, also has plans to make their packages deterministically built [Fdr15].

It is important to note that just because some application has been deterministically built, it does not guarantee that there is no malicious code. It merely solves the need of trusting a single builder. The source code would still need to be appropriately reviewed and tested, as is the norm for secure software.

Deterministic compilation is a reasonable security feature that removes the need to trust a single entity. The way deterministic building achieves this minimized trust model opens up for other attacks which we explore further in Chapter 9.

4.3.3 OFFENSIVE TECHNIQUES

Buffer overruns [Ale96] have for a long time been a common source of software vulnerabilities. The buffer overrun vulnerability may be exploited to perform a code injection attack, where the goal is to inject arbitrary code and replacing the return address with the address of the injected data. There are several well-known and widely used mitigations against this approach. Since the injected code should not be executable, but rather considered as data, the memory pages corresponding to this data can be marked as non-executable. Data Execution Prevention (DEP) [Hen09] is a hardware supported OS feature implementing this idea. A similar approach is the $W \oplus X$ security feature [pro00] in which memory pages are either writable or executable, but not both. While this will prevent the classic code injection attacks, it will not prevent code reuse attacks. In these attacks, the adversary will not inject the code to be executed, but instead direct the program flow to code that is already loaded by the process, typically a shared library. One example of a code reuse attack is the return-to-libc attack [c0n] in which the attack points to existing code in libc, a library used by many programs.

A more powerful code reuse attack is Return-Oriented Programming (ROP), in which the attacker identifies small pieces of usable code segments, called *gadgets*, and chains them together using a `ret` instruction. A `ret` instruction will pop an address from the stack and continue execution at that address.

RETURN-ORIENTED PROGRAMMING

Return-oriented programming [Sha07a] is an exploitation technique that allows for arbitrary code execution without having to inject code into the vulnerable process. To achieve this, an attacker constructs a payload of addresses, each pointing to a small sequence of instructions reachable and executable by

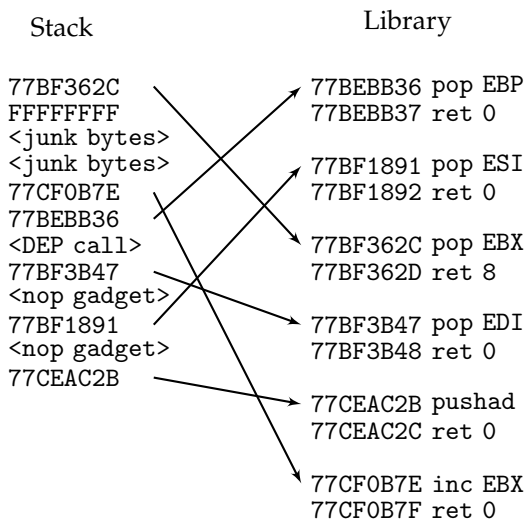

Figure 4.2: An example of a ROP exploit. The addresses to gadgets are located on the stack, together with necessary integers and junk. The junk bytes are used when the `ret` instruction pops several values from the stack before continuing execution.

the affected process. These instruction sequences are called gadgets and typically consist of very few instructions, ending with a return instruction (`ret`). This return instruction will pop the next dword from the stack, put it into the instruction pointer register (EIP) and continue execution at the next gadget. Gadgets do not have to be aligned with the intended instructions. Any byte that represents the opcode of a `ret` can potentially be used as a gadget.

Not only `ret` instructions can be used in these types of attacks. It is also possible to use jump-based instructions as in [BJFL11a] [CDD+10].

As an example [neg13], one frequently used ROP exploit pattern disables DEP, which makes normal code injection possible. Figure 4.2 shows the layout of the payload, residing on the stack, and the corresponding gadgets found in a library. `DEP call` represents the address of the function to disable DEP for the current process.

The details of the exploit can be found at [neg13]. The `pushad` instruction, executed as a last step, will push all registers to the stack in the following layout

```
EDI
ESI
EBP
```

```
|-<pointer>
| EBX
| EDX
| ECX
| EAX
->
```

The <pointer> is the address to the memory location immediately after the location of the registers on the stack. EDI and ESI are set with a nop gadget address as execution will have to begin at the address specified by EDI and then move through both EDI and ESI and get to EBP which will execute SetProcessDEPPolicy with the flag value of zero found in EBX. When this is finished execution will continue on the stack at <pointer> and succeed as DEP has been disabled. The shellcode to be injected would be placed at the <pointer> address.

This exploit includes six instructions and if used as in the referenced exploit, where each instruction is included in its own gadget, we will have six gadgets each consisting of one instruction.

In [WLL+13], the inside threat of an evil developer is described. Wang et al. present an attack where iOS applications are released with a deliberately injected vulnerability by the developer that can be remotely exploited. The vulnerability is placed in such way that it should pass the review and vetting process of Apple. The application is introduced with a bug and when triggered, a ROP attack is performed. In the attack, the attacker still have to send the attack payload, i.e., the gadget addresses/offsets to the vulnerable application while the techniques presented in Chapters 8 and 9 triggers the backdoor directly in the code without any information required on how to locate the attack payload.

In the dissertation [PRA], the author proposes a method to hide ROP gadgets in open source projects in order to evade code reviewing. The author argues that injecting backdoor code in a project is difficult and focus is instead on introducing a vulnerability in the code. This may be true for a new developer committing code that will be peer reviewed by others and will likely detect an attempt to plant a backdoor. However, in the scenario of a trusted developer that introduces a backdoor it may have a different outcome. Yet another potential scenario can arise if the trusted developer colludes with one or multiple code committers to review their contribution but intentionally miss out on the backdoor.

In a first attempt to utilize return-oriented programming in a non-malicious context, RopSteg [LXG14] shows how it can be used for software obfuscation by hiding code in gadgets, a property that is introduced as program steganography. A different approach with overlapping instructions used for tamper-

resistance in software is shown in [CVC+02] where the hidden execution path contain code that is used for hash computations. Later this is used for verification of the integrity of the main execution path using oblivious hashing [Y. 03]. The idea is that when the code is tampered with, the hidden code will also be modified and the resulting verification will fail.

4.3.4 DEFENSIVE TECHNIQUES

One widely available countermeasure against code injection and code reuse attacks is Address Space Layout Randomization (ASLR) [PaX03]. ASLR will randomize the base address of the program's text, stack, and heap segments and the adversary will not know at which address a library starts, or where the gadgets will be located. However, it has been shown that ASLR can sometimes be bypassed [SPP+04]. The addresses of certain instructions can be leaked through other vulnerabilities and in some cases it is feasible to brute force the start address of a library, thereby succeeding with a ROP attack even in the presence of ASLR.

Following the introduction of ASLR in Windows XP SP2 (2004) and the Linux Kernel (since version 2.6.12, 2005), writing exploits has become much more difficult. Since then, the number of base addresses that are randomized has increased, and so has the entropy of the randomization.

However, the efficiency of ASLR is limited. First, some small amount of code is not randomized, leaving the possibility to still use gadgets in the code where the location is predictable. Even though this code is rather small, it has been shown that it is possible to find usable gadgets in it [SAB11]. Randomizing the application code is one kind of protection against these attacks [WMHL12]. Another aspect of ASLR, as was shown in [SPP+04], is the limited entropy in the address space, which makes it possible to brute-force absolute locations.

Even if the address space layout is re-randomized between each attempt, the number of required attempt is only slightly increased compared to the case when a brute force through a fixed number of addresses is used. The number of attempts needed in the re-randomization case is geometrically distributed with an expected value of 2^n, where n is the entropy of the randomization. Several other techniques for bypassing ASLR has been described in [MÂÍ08].

In addition to brute-forcing the ASLR, it has been shown that information leakage can occur through e.g., buffer and heap overrun bugs [Dur02] [Vre10] and other types of vulnerabilities [Ser09] [SMD+13]. This could give an attacker at least partial information about the location of ASLR-affected code. Being able to read a return address on the stack is enough to deduce information about the base address of a DLL.

The exact means by which an attacker bypasses ASLR, may it be through

brute force or information leakage, are independent of our payload detection algorithm presented in Chapter 10.

There have been several proposed defenses against ROP attacks, all taking slightly different approaches and using different assumptions. A typical mitigation is to identify some specific features in the attack that distinguishes it from benign code execution and then build a mitigation technique based on those distinguishing features [CXS+09a] [CZM+14] [DSW11] [Fra12] [PPK13] [PK11]. Two examples of well known defenses are kBouncer [PPK13] and ROPecker [CZM+14], which are based on heuristics in order to detect ROP attacks. They e.g., consider the number of instructions between consecutive indirect branches. The heuristic rule is that ROP attacks typically use short gadgets chained together using indirect branches. It has recently been shown that such defenses are inadequate and that it is possible to craft attacks that bypass these, and similar, defenses [CW14] [DSLM14] [STP+14]. Another approach is to rewrite libraries or targeted code such that it is not usable in an attack [LWJ+10] [OBL+10a] or to randomize addresses which are needed by an attacker [GKKB12] [HNTC+12] [PPK12a] [WMHL12].

Instead of detecting the attacks on the target systems, another goal may be to detect ROP attacks in data. In [PK11] data was scanned and possible exploits were speculatively executed in order to determine if they were exploits. This requires a snapshot of the virtual memory of the process that is protected. In [SSO+13] the authors consider a detection approach where documents are analyzed to find ROP attacks. Documents are collected and sent to a separate virtual machine, where they are opened in their native application, and a memory dump is then analyzed for ROP payloads. This approach has the advantage that they do not need to make any changes to, or run any programs on, the target computers. This allows for fast deployment of the defense.

Control Flow Integrity (CFI) [ABEL05] [ABEL09a] can also be used to stop code-reuse attacks. While this is a robust counter-measure for many attacks, properties such as overhead and complexity has limited its adoption. Still, it is a promising mitigation approach which has been given much attention and several aspects of CFI have been considered recently [BJF11] [ZWC+13] [ZS13].

As more and more utilization of code-reuse attacks like ROP happens, research into defenses based on CFI intensified and one recent advancement on this front is from a team called PaX, most famous for their security-hardened Linux kernel. The PaX team proposed Return Address Protection (RAP) [Tea16] which is a CFI scheme that has a low impact on performance and scales well with the CFG of an application.

There has also been much work on how to mitigate the source of the exploit, namely the presence of a buffer overrun vulnerability. Well known defenses include StackGuard [CPM+98], which is a compiler extension that places a

canary value on the stack. Before returning from a function, the integrity of the canary is verified. Stack Shield [Ven00] makes a copy of the return address and saves it in the beginning of the data segment. Before returning from the function, the return address is compared with the saved value. These mitigations have been followed by other compiler based solutions, e.g., PointGuard [CBJW03]. An overview and comparison of several buffer overflow prevention implementation and their weaknesses can be found in [SJ04].

Malicious code scanners [KK04] [KC04] [NKS05] have been used to detect malicious code using pattern matching. However, since the ROP payload does not include the malicious data, just addresses to it, they can not be used to detect ROP attacks [WPLZ10]. Our detection mechanism, presented in Chapter 10, looks for the addresses to gadgets and can be seen as an ad hoc mitigation against ROP attacks, based on ideas from malicious code scanners.

4.4 CONCLUSIONS

ROP was once the state-of-the-art of exploitation techniques. It has now become so common, and new more advanced techniques developed, that ROP can hardly qualify as state-of-the-art any longer. The first defense proposed against ROP-attacks was DROP [CXS+09b] (Detecting ROP) which is based on dynamic runtime instruction instrumentation in order to detect patterns commonly found in ROP attacks. This defense had several weaknesses including a large performance overhead and detection of gadgets only possible in a single library. As more defenses were proposed, researchers started to find enhancements to the ROP attack, like JOP (Jump-Oriented Programming) [BJFL11b].

Attackers can afford to have complicated exploitation techniques, because they only have to work once. Once executed, system compromise happens, the exploit has fulfilled its function. Attackers can even afford having their exploits work probabilistically where sometimes failure may happen. An exploit is not something that need to run day and night like production system software. The level of sophistication and ingenuity in the attacker's toolkit continues to increase [Ada15] with recent advancements like the DRAM row hammering [SD15] which abuses the physical properties of RAM to flip bits in some memory cells by "hammering" adjacent cells. Row hammer is an interesting technique in that the vulnerability is in the physical nature of how memory works, with no possibility to protect currently vulnerable systems with a simple software upgrade.

5

Virtual Machine Detection

Fingerprinting services and operating systems have always been an integral part of an attacker's methodologies before attempting compromise of a system. Fingerprinting is possible because different systems implement protocols differently. By noticing discrepancies in the behavior of the different implementations, one can gain valuable information about the target system.

As virtualization technologies become more and more utilized, it is interesting to study how the presence of these can be detected. Detecting the presence of a virtualized execution environment can benefit both malicious and non-malicious use, and in this chapter a technique for passive detection of some virtualization products is described. This chapter is based on the publication titled »A technique for remote detection of certain virtual machine monitors«.

5.1 INTRODUCTION

Virtualization is a technology that has many applications. The use of virtualization can improve utilization of server farm hardware resources and simultaneously lower energy costs.It can also be useful from a security point of view. For more detailed information on virtualization, see Chapter 2. One particular use case in this aspect relates to malware analysis [LS10] where virtualization makes it easy for an analyst to restore the state of the operating system in which the analysis took place, to an earlier non-infected state. With virtualization the analyst can avoid the hassle of reinstalling the operating system between each sample analysis and become more efficient. A malware sample that detects it is in a virtualized environment may draw the conclusion that it is being analyzed, and choose to stop execution of its malicious pay-

load. By refusing to decrypt itself, the analyst is forced to spend more time on the sample to study it. The malware could also decrypt itself to an alternative, benign, piece of code which could potentially fool the analyst into thinking that this sample is not malicious at all. There are known malwares that utilizes virtualization detection, the most notable being the conficker worm [Con09]. In this case, the malware will not execute malicious code if it finds out it is running within a virtual machine.

Similarly, if the attacker has an exploit that only works on some specific virtualization product, detecting the presence of such before executing the exploit could be the difference between detection of the exploit and longer lifetime of it.

Honeypots and honeynets constitute a technology that provides security researchers with an environment aimed to lure attackers in to, in order to study their modus operandi, tools and techniques. Honeypots may be confined in a virtualized environment where a potential attacker can be properly contained, while all tools used for monitoring exist outside of the virtual machine. Virtualizing the honeypot can lead to the attacker being able to detect it and halt activities that may divulge information about his techniques. Of course, as more virtualization is utilized the more difficult it is for an attacker to deduce whether they are in a honeypot or a legitimate target. These are some cases where VM detection benefits the attacker.

Some malwares are based on virtualization technology, such as the rootkit dubbed Blue Pill [Rut06] which utilizes x86 virtualization. The idea is to install a hypervisor and make the currently running operating system run as a virtual machine under it. Detecting virtualization, where there should be none, would allow an infected user to take action. Another example of malware implemented as virtual machines is the subvirt rootkit [KC06]. In this case VM detection technology is a defensive method.

5.2 VIRTUAL MACHINE DETECTION TECHNIQUES

Several methods to detect virtual machines have been proposed. Redpill [Rut04] is a detection technique for VMware Workstation and Microsoft's Virtual PC, that reads the location of the Interrupt Descriptor Table (IDT). If the location is one of those known to be used by the virtualization product, they are detected as being in use. Since this address is returned by the hypervisor itself, it is easy to protect against this technique. Red pill is also interesting due to the fact that unprivileged users could retrieve the IDT location.

Detection is possible when a discrepancy in execution in, or communication from, a virtual machine can be quantified. Because of the hypervisor's involvement, discrepancies are a natural consequence of virtualization and

it can often be difficult, or even impossible, to avoid them. Several Virtual Machine Monitor (VMM) detection techniques have been proposed, one notable being the previously mentioned red pill [Rut04] which, even if it is not practically useful anymore, is a good example of how discrepancies facilitates detection.

Remote detection of VMMs were proposed in [FLM+08]. However, root access to at least one VM was required in order to execute the benchmarking code with highest privilege level and interrupts turned off.

5.2.1 TIMING ANALYSIS

According to Popek and Goldberg, a virtual machine should fulfill the equivalence, or fidelity, property, which states that an application should be indistinguishable whether it is executed in a virtual machine or a real machine. However, because the hypervisor must intervene with certain instructions, this property is difficult to achieve in full. When the hypervisor intervenes with some instructions it will incur an overhead in execution time, and the more an application utilizes such instructions, the easier it becomes to notice discrepancies in execution time compared to when that application is executed in a non-virtual machine.

Timing data could also be retrieved from TCP and ICMP headers to measure discrepancies in the round trip time.

5.3 A NEW TECHNIQUE FOR REMOTE AND PASSIVE VMM DETECTION

In this chapter details will be provided to show how certain behavior of the NAT implementation in the popular client virtualization tools VMware Workstation/Player and Oracle's VirtualBox will allow us to tell if a client is running a web browser in a virtualized environment. Our proposed technique works remotely and in a passive manner. It does not require the external verifier to run benchmarking code on the target machine, nor probe the target for open ports. It is solely based on network traffic initiated by and sent from the target itself. A proof-of-concept implementation is provided where the only requirement is for the target to connect to a web server that is under the control of an attacker. This will, under some reasonable circumstances, allow us to determine if the client is visiting from a guest in VMware Workstation/Player or VirtualBox. In the sequel, for simplicity, we only refer to VMware/VirtualBox, when we mean all three products.

Many times, active probing is necessary when fingerprinting a system to gain any valuable information, because it is how the edge cases behaves that will differ between different systems. The technique proposed here will evaluate information that is available in all packets, namely the Time-To-Live (TTL)

and IP identification (IP ID) values contained in the IP Header. This makes passive fingerprinting easier. If the target is being used as a server, with ports open for probing, the technique can easily be modified to perform active detection. It would then be possible to check any computer if a service is running in a virtual machine under VMware/VirtualBox. However, server virtualization typically does not use the mentioned virtual machine monitors targeted in this chapter, as they are primarily client-side virtualization products.

The detection technique has several malicious use cases. An attacker that want to infect users with malicious content on a web server and at the same time avoid detection by malware analysts could use the detection technique to serve benign content to users connecting from a virtual machine and malicious content to everyone else. This works under the assumption that the malware analyst connects from one of the vulnerable virtualization products with the appropriate configurations. Another case is where the attacker is in possession of an exploit that is specifically targeting said virtualization product. In this case malicious content would be served only when the presence of the virtualization product can be assumed.

Virtualization detection will in this case allow an attacker to make more informed decisions on who to try to attack, in the end making the web server go unreported for longer and the potential vulnerabilities and exploits utilized in the attack to stay undisclosed.

5.3.1 NETWORK ADDRESS TRANSLATION (NAT) AND NETWORK PROTOCOLS

The proposed technique is based on the fact that the NAT device used in VMware/VirtualBox mangle network traffic. A NAT device is used to allow a user with a limited amount of global IP addresses (usually just one) to connect several internal machines to the Internet by having the NAT supply a number of local IP addresses for the internal network. Typically, a NAT only changes the source IP address and source TCP/UDP port in outgoing packets and destination IP address and destination TCP/UDP port in incoming packets. Additional changes are made by the VMware/VirtualBox virtual NAT device as these NAT engines receive the network traffic and resend it using their own TCP/IP stack. This behavior is documented in VirtualBox user manual [Vir, Ch. 6], but we have not found it in any VMware documentation.

We have identified three additional changes, which give rise to anomalies, allowing us to distinguish these NAT implementations from other NATs. Each anomaly is presented in detail in this section. Other NATs that have been considered, and which do not have any of these anomalies, include the NATs in the competing virtualization products Xen and Virtual PC, IPtables and the dd-wrt firmware used in many home routers. Note that we consider default

installations as e.g., in IPtables it is possible to configure the IP header values in outgoing packets. While our list of tested NATs is not exhaustive, and it cannot be due to the large number of proprietary implementations, the fact that only the VMware/VirtualBox NATs show this behavior indicates that the probability for false positives are low.

5.3.2 PREREQUISITES

The proposed technique for detecting VMware/VirtualBox is based on anomalies in the NAT implementations. The detection is partly facilitated by differences between how Windows and Linux sets the TTL and IP ID fields in the IP header and thus, we need the following prerequisites in order to successfully detect a target:

- The target must have the VMware/Virtualbox NAT device enabled.
- The target's guest and host operating system must provide different initial TTL values and/or different IP ID generation methods. Examples of operating systems that differ in these aspects are Windows and Linux, which are both supported by VMware and VirtualBox.

Using bridged networking and/or the same operating system family for both guest and host would result in a false negative.

5.3.3 TIME-TO-LIVE (TTL)

The purpose of the 8-bit TTL value in the IP header is to avoid having a packet circulating indefinitely on the Internet in e.g., a routing loop. It is decremented by one for each router hop. The initial value of the TTL in outgoing packets is implementation specific. In Windows, the initial value is 128, while a typical Linux system sets it to 64. While a NAT can modify the TTL before sending a packet out on the network, to the best of our knowledge it has no particular reason to do so. However, the value of the TTL is changed when using the VMware/VirtualBox virtual NAT device. Due to the fact that the host operating systems TCP/IP stack is used to rebuild outgoing packets, the TTL is always changed to the default value used by the host. A Windows guest running on a Linux host will create IP packets with a TTL of 128, but it is changed to 64 before sending the packet out on the network. A corresponding behavior can be seen for a Linux guest on a Windows host, i.e., the TTL is changed from 64 to 128 when passing the NAT. This modification of the TTL value has an effect on tools that rely on the TTL. As an example, `traceroute` relies on incrementing the TTL for each new packet in order to determine the route taken for a packet to a given destination. Using traceroute in a guest running on VMware/VirtualBox does not give the expected behavior as the TTL is rewritten in the NAT to the initial value set by the host OS.

5.3.4 IP IDENTIFICATION (IP ID)

The main purpose of the IP ID value is to reassemble fragmented packets. The exact value of the IP ID value is irrelevant, instead it is important that all packets from one host that is currently on the network have different IP ID values. Otherwise, the reassembling of fragmented packets would be ambiguous. The generation algorithm of IP ID values is implementation specific. RFC 4413 [MW06] specifies three distinct ways for generating the IP ID value:

- **Sequential jump:** One *global counter* is used. All outgoing packets receives an IP ID value from this counter and the counter is incremented by one for each outgoing packet. This generation method is used by e.g., the Windows operating systems.
- **Sequential:** Each outgoing packet stream has its *own counter* which is incremented by one for each outgoing packet in the stream. This generation method is e.g., used by Linux operating systems.
- **Random:** Each outgoing packet is assigned a random value generated from a Pseudo Random Number Generator (PRNG). This method is e.g., used in the OpenBSD operating system.

Similar to the initial TTL value, the Windows and Linux operating systems differ in the way that the IP ID in the IP header is treated. Again, similar to the initial TTL value, the VMware/VirtualBox virtual NAT device changes the IP ID value, by using the algorithm which is default for the host operating system. While we have not found any other NAT implementation that changes the IP ID for outgoing packets using a different algorithm than that of the originating OS, there are apparent benefits of having the NAT control the IP ID. With several guests running on one host, or alternatively, several computers behind one NAT, one guest (or computer) has no information about the IP ID values generated by other guests (or computers). Thus, the probability of collisions in IP ID values leaving the NAT increases. If the NAT is allowed to control this value, collisions can be avoided. The fact that this control is implemented by the VMware/VirtualBox NAT is clear when examining packets originating from a Linux guest on a Windows host. The IP ID of these packets are generated by the same sequential jump algorithm as packets originating from the host itself.

It should be noted that packets from two different connections are needed in order to reliably distinguish the generation algorithms used by Windows and Linux operating systems. It is possible to use only one connection, but that assumes that the guest is also using other connections and that packets are examined and compared before and after packets are sent on other connections.

The IP ID has been used in many ways to extract information about a target, such as the number of hosts behind a NAT [Bel02]. The IP ID field of packets leaving the NAT was used to count the number of hosts. A fictional story based on incrementing IP ID values is given in [Fyo04]. In this story the character abuses the fact that the IP ID is incremented by one for each outgoing packet. Over time he can see patterns in the traffic to different organizations and correlate the increment in the IP ID and how much the stock of that company has gone up or down. Based on these observations, it was possible to get a better idea of whether or not to invest. Even though this is only a fictional story, it is an interesting idea, similar to the idea in [Jor09], where tracking spam could give leverage on the stock market. Finally, steganography software exists which hides data within the IP ID field [KA03].

5.3.5 TCP CONTROL FLAGS

While the TTL and IP ID fields are located in the IP header, the third anomaly we have found is in the TCP header. More specifically, in the TCP control flags. A connection can be terminated either by sending a FIN packet or a RST packet, i.e., a packet with either the FIN or the RST control flag set. When a guest terminates a connection by sending an RST packet, VMware/VirtualBox translates this, in some cases, to a FIN packet. This can be seen as creating a graceful shutdown of the connection instead of tearing it down with a connection reset. An interesting fact is that this behavior is more prominent when the guest and host use different operating system families, i.e., Linux host and Windows guest or Windows host and Linux guest. It does however happen in some cases when the host and guest are using the same OS.

5.4 PROOF OF CONCEPT

A small detection daemon has been implemented as a proof of concept. It looks for the anomalies in the TTL and IP ID fields in the IP header to detect if HTTP packets originates from a VMware/VirtualBox guest. The implementation uses a web server, in our case an Apache server [Apa] on a Linux system. On the server, a small custom packet sniffing daemon is run, collecting IP ID and TTL values from connecting clients. The daemon utilizes the libpcap packet capture library. The daemon also analyze and writes the IP address of a detected virtual machine to a file *detected.txt*, which the web server can query to decide what content to serve a particular user.

The TTL gives information about the client operating system or, in the case of VMware/VirtualBox, the host operating system. In order to also use the information provided by the IP ID generation algorithm, two connections are needed. When a user connects to the default HTTP port (80) on the web

server, it is immediately redirected to another port, 8080 in our case. Comparing the IP ID values in packets to different ports will allow us to make an informed guess about the generation algorithm. We compare the IP ID value of the last packet from the connection to port 80 and the IP ID value of the first packet from the second connection. If the difference between those two values are below a certain threshold value, we conclude that the sequential jump generation method, i.e., Windows is used. The server queries the file *detected.txt* and can determine the contents of the returned web page based on the result.

However, the information given by the TTL and IP ID is not enough. It can tell us that the client uses e.g., Linux, but it can not distinguish between a plain Linux computer, Xen or Virtual PC with Linux guest or VMware/VirtualBox with Linux host and Windows guest. However, looking at the *user-agent* string found in the HTTP request header, we can get information about the operating system used for the original IP packet. The user-agent string will often contain a substring of the base operating system, and this will not be changed by the NAT. As the VMware/VirtualBox NAT is the only NAT that translate TTL and IP ID, as far as we are concerned, we can distinguish this from other NAT implementations and also packets not passing through a NAT.

In Fig. 5.1 a flow diagram is given, showing the communication between server and client when the server attempts to gain information about the clients usage of virtualization.

5.5 A NOTE ON IPV6

The technique presented in this chapter have been based on IPv4, but it could be utilized when IPv6 is used as well. The TTL still exists in an IPv6 header, although it is called the hop limit. There is no IPID field in IPv6 headers though. The same initial values seems to be in use by Windows and Linux OSes, as they were for the TTL value in IPv4 packets.

Since IPv6 routers do not perform fragmentation of IP packets, hosts are required to do path Maximum Transmission Unit (MTU) discovery, use the default MMU to guarantee proper propagation of the packet, or perform fragmentation end-to-end. So for IPv6, we only have the hop limit to rely on under an IPv6 setting.

5.6 CONCLUSIONS

A new passive remote detection technique for VMware Workstation, VMware Player and VirtualBox was detailed in this chapter. It is based on the fact that the NAT used in these VMMs rewrites information before it is sent out on

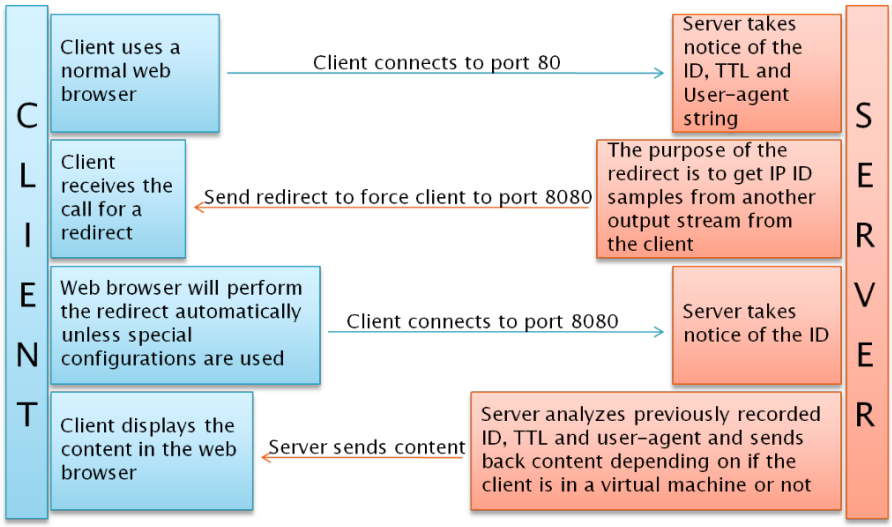

Figure 5.1: Flow diagram of communication between client and server.

the network. The fact that this behavior has not been found in other common NAT implementations, together with the assumption that different TTL default values and IP ID generation algorithms are used in the guest and host machines, will allow us with high certainty to determine that one of these VMMs is used and that the traffic originates from a guest in this VMM. If the TTL and IP ID generation algorithm is changed on the network and the TCP RST control flag is replaced by a FIN, then our detection will receive false positives. It is an open problem to determine under which other circumstances, if any, that false positives occur. Possible applications could be e.g., anonymity solutions such as Tor [DMS04], web proxies and VPNs.

Users who consider this possibility for remote detection to be a threat should use bridged networking for their guests or mangle their packets accordingly to avoid detection. If VMware inc. or Oracle consider this a security problem, they could consider a redesign of their NAT implementation such that this information leakage is prevented.

6

Secure RPC Mechanisms for Embedded Systems

Computing devices of various kinds are gradually becoming a significant part of our lives. Today, we can use devices such as smartphones to perform many of our job duties, perform financial transactions with stores, banks or even stock markets, and interact with almost any part of society. Given our dependency on such devices, one could ask what can be done to make them more secure. Securing computer systems is a very broad area and can involve anything from legislation to education to secure software and hardware design. In this work, we will concentrate on two aspects that can with relative ease be added to existing systems: isolation and secure communication. In particular, we will consider a software architecture where components at different security levels coexist and our objective is to protect some components from others by using some type of isolation mechanism. However, we would also like to introduce a secure communication channel as the only method to bypass this absolute isolation.

As an example, consider a scenario where a smartphone contains a banking application and a web browser. For obvious reasons, we do not want the browser to have free access to our banking data. At the same time, we may want to use the banking application to make payments when browsing some trusted online stores. Another example is the emerging *bring-you-own-device* (BYOD) culture in organizations. In order to not leak company secrets, it would be a good idea to separate personal and company data and functionality on the phone. This can be achieved with a secure and trusted subsystem where sensitive data and operations are stored.

In this work, we consider isolation and secure communication as an *add-on* to an existing system. More specifically, we will examine the Client API specification within the GlobalPlatform architecture which defines a secure RPC

mechanism for embedded systems [Glo11b] [Glo10]. This API will be implemented in conjunction with various security and isolation mechanisms such as virtualization. We will then discuss key attributes of each implementation such as performance, security and complexity.

For more background on isolation technologies and SELinux, see Chapter 2.

6.1 GLOBALPLATFORM

GlobalPlatform's specification for a Trusted Execution Environment defines a secure RPC mechanism for communication between two components. This can be implemented between two applications communicating over a traditional RPC channel such as pipes, sockets and shared memories. For better control one may also move the communication mechanism into the OS layer or even a hardened operative system or a hypervisor. These subjects will be explained briefly in this chapter.

GlobalPlatform is a non-profit standardization organization involved mainly in the smart card segment. The GlobalPlatform specifications are divided into three groups: smart card, device and system.

Device specifications include a Trusted Execution Environment (TEE) at a fairly system and platform independent level. Before this specification was introduced, each secure device had its own API and technology for realizing a TEE, which made development of secure applications difficult and time consuming. This resulted in less developers utilizing TEE technologies and in the end less secure software. It is important to note that GlobalPlatform has not implemented any TEE solutions themselves, but only published guidelines on how it should be done.

The TEE specification is divided in three parts: **System Architecture** explains the overall software and hardware architecture of the TEE and the basic concepts [Glo11b], **Internal API** defines the API used by the trusted applications [Glo11a] and **Client API** specifies a mechanism for secure communication between client (normal) applications and the trusted applications [Glo10]. The latter specification is the subject of investigation in this work.

6.1.1 THE CLIENT API

The Client API defines a mechanism for secure communication between *client applications* (CA) within the rich world OS and *trusted applications* (TA) running inside the TEE.

The specification defines a handful of operations at source code level for remote procedure calls across the security domains. These operations can be divided into the following groups (in the order of use):

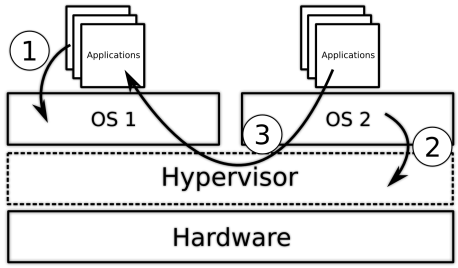

Figure 6.1: Hypervisor running two guests. The illustrated communication channels are (1) system call, (2) hypercall and (3) RPC between guests

1. Initialize
Set up a context and initialize a communication session with the callee.

2. Prepare shared memories
Allocate or register shared memories that will be used as parameters in remote calls.

3. Invoke command
Performs the remote call with the given parameters.

4. Release & finalize
Release memories. Free session and context and close connection to TA.

6.2 HYPERVISORS

Slightly simplified, we define a type 1 hypervisor as a lowest-level software that regulates an operating systems access to resources. In the context of device security, a hypervisor can be used to separate running software into two domains: a secure guest running sensitive software (TAs) and a normal (rich) guest running everything else (CAs) [1].

For this setup to be usable, one must also provide a well-defined and secure communication mechanism between the guests. Normally there are communication channels between user-space applications and the OS (system calls) and similar channels between the OS and the hypervisor (hypercalls). With a careful combination of these two one may create a third channel for communication between applications/OS's across guests, as illustrated in Figure 6.1, which in this work is used to perform GlobalPlatform RPCs.

[1] This can of course extend to more guests and more security domains

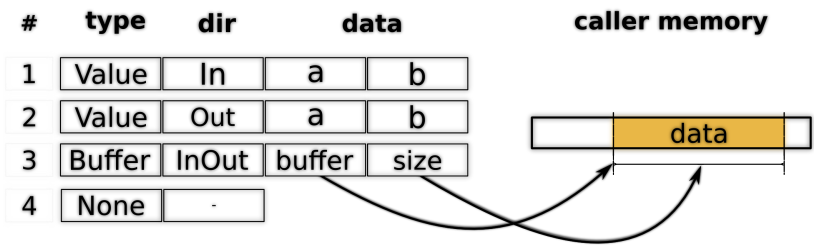

Figure 6.2: InvokeCommand parameter example

6.3 IMPLEMENTATION

Once a GlobalPlatform communication session has been established, the InvokeCommand is used to perform a remote procedure call to the other end:

```
TEEC_Result TEEC_InvokeCommand(
 TEEC_Session*            session,
 uint32_t                 commandID,
 TEEC_Operation*          operation,
 uint32_t*                returnOrigin);
```

This function is given a remote procedure identifier (*commandID*) and up to four parameters (in *operation*) with a type (None, Value, Buffer) and a direction (In, Out, InOut). An example is shown in Figure 6.2 where three parameters are used.

The Value type is used for small amounts of data (up to 64 bits), while Buffer is used to pass a pointer to a memory region containing larger structures. A Buffer is partially represented by a SharedMemory structure:

```
typedef struct {
 void* buffer;
 size_t size;
 uint32_t flags;
 ...
} TEEC_SharedMemory;
```

Users can use the API to allocate a new buffer:

```
TEEC_Result TEEC_AllocateSharedMemory(
 TEEC_Context*            context,
 TEEC_SharedMemory*       sharedMem);
```

In some situations, it is instead desirable to register a previously allocated memory using the following API function. Unfortunately there are some security implications with this, which we will discuss later in this work.

```
TEEC_Result TEEC_RegisterSharedMemory(
 TEEC_Context*          context,
 TEEC_SharedMemory*     sharedMem);
```

It is trivial to move a Value between caller and callee, for example by using the same mechanism normal function calls utilize. Buffer parameters are more complicated as callers memory is normally not visible in the address space of callee. This can be handled in a number of different ways, which are described in the remaining of this chapter.

6.3.1 LINUX SHARED MEMORIES

One method for sharing large amounts of data is to utilize shared memories, which are memory regions that may be accessed by multiple processes simultaneously. A shared memory can be achieved in many ways, but for our Linux implementation we will use the concept of memory mapped files using the POSIX functions shm_open and mmap.

shm_open will create or open a POSIX shared memory object, while mmap will map it to a memory region and allow other processes to do the same. These can be incorporated into the AllocateSharedMemory function in GlobalPlatform as the basis for a RPC mechanism.

6.3.2 IMPLEMENTING GLOBALPLATFORM WITH SELINUX

For background on SELinux, please visit Section 2.2.1. The vanilla shared memory implementation suffers from some security problems. For example, we cannot guarantee that the receiving end really is the trusted callee. To solve such issues, we will utilize SELinux.

To achieve a functional and secure TEE using SELinux we must first define a set of subjects and objects which are essential for the solution to work properly. We must establish how shared memory should be handled and how the transfer of execution from a CA to a TA should be done in a secure fashion. Figure 6.3 illustrates this idea.

We assume that all CAs are executed under the unconfined_t domain.

The TA executables are labeled with the type tee_exec_t. Whenever an operation within a TA is requested, it will be executing under the tee_t domain and the transition to this domain is handled by SELinux once the CA has called the TEE client API through the function InvokeCommand. In the SELinux policy we have the following rules:

```
allow unconfined_t tee_t : process transition

allow unconfined_t tee_exec_t :
```

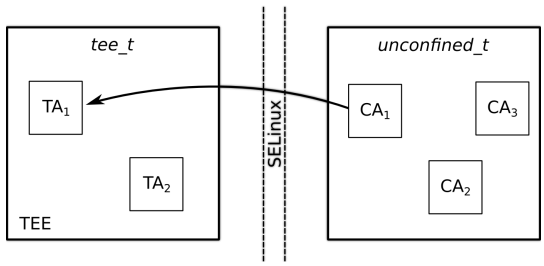

Figure 6.3: The SELinux setup

```
        file { execute read getattr open }

allow tee_t tee_exec_t : file entrypoint

type_transition unconfined_t tee_exec_t :
        process tee_t
```

The first rule specifies that the domain unconfined_t is allowed to transition into the tee_t domain. Next rule specifies that subjects in the unconfined_t domain are allowed to execute files labeled tee_exec_t, and thus also need the read privilege. The third rule specifies that files labeled with tee_exec_t have an entrypoint in the tee_t domain, i.e. only these files may make the transition to the tee_t domain. The last item specifies that when a subject within the unconfined_t domain executes a file labeled tee_exec_t, the newly created process should be confined to the tee_t domain, given that the subject which invoked the process has the proper permissions.

In our implementation, InvokeCommand calls the execl function, which executes the desired TA. SELinux handles the transition to the tee_t domain where it can run isolated.

6.3.3 THE HYPERVISOR AS A GLOBALPLATFORM TEE

We have also chosen to implement a minimal TEE with the help of the *SICS Thin Hypervisor* (STH), which is a virtualization software specifically created to enhance security in resource constrained embedded systems [GDN11] [2]. The STH has the ability to run multiple operating systems on the same CPU

[2] In practice, any thin hypervisor could be used here. We choose to use STH since (1) we are familiar with it and (2) it can function even in the absence of hardware security extensions such as ARM TrustZone and ARM virtualization extensions. The latter seemed to be a reasonable assumption when adding secure communication to an existing system with no previous security mechanisms.

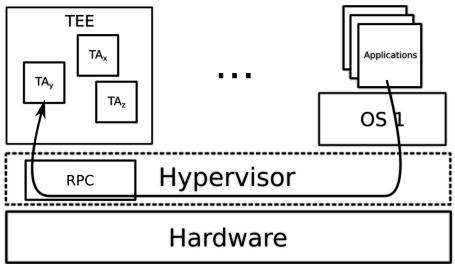

Figure 6.4: Executing trusted application as a guest on top of the STH hypervisor. Note that all communication with this guest is supervised by the hypervisor.

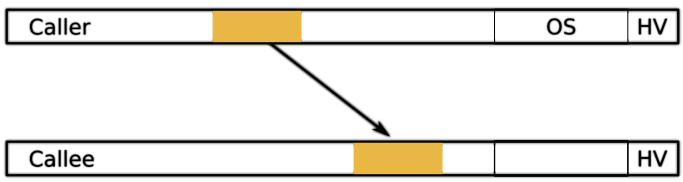

Figure 6.5: Temporary mapping of caller memory into that of callee during a RPC.

while guaranteeing guest OS isolation. As illustrated in Figure 6.4, STH can also host a trusted guest that provides security services to other guests.

In the hypervisor implementation the caller and callee normally reside inside different guests and have completely different address spaces. We handle this by automatically mapping Buffer parameters into the address space of the callee, as illustrated in Figure 6.5. This is similar to the temporary mapping mechanism used in some IPC implementations such as that of L4 [Lie93], but without using flexpages and pager hierarchies. Notice that unlike the Linux shared memory implementation discussed earlier, the hypervisor does not require prior definition of used memories.

This method works quite nicely as long as the shared Buffer is aligned to page boundaries, which we could enforce in our implementation of AllocateSharedMemory. However, if the shared memory is not fully aligned with page boundaries (which may happen if RegisterSharedMemory is used instead), the virtual memory mapping created by the hypervisor will reveal adjacent data within the same page to the callee. If this is unacceptable, the unaligned sections must be copied into new empty pages before they are presented to callee, as illustrated in Figure 6.6.

The maximum number of pages that are replicated is eight in either direction (i.e. all parameters are InOut buffers with first and last page unaligned).

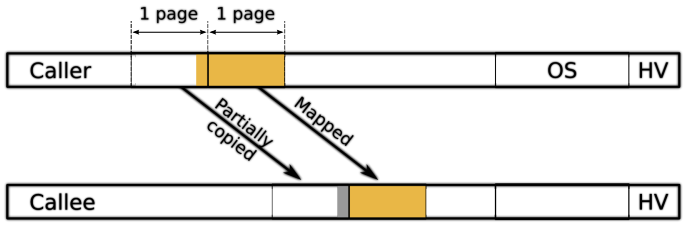

Figure 6.6: Memory re-mapping and copying for unaligned regions.

These pages must also be cleared from previous content before use. The performance penalty for this depends on the system page size, as illustrated in Table 6.1.

Table 6.1: Unaligned page usage penalty overhead per RPC.

Page size	Bytes copied (worst case)	Bytes cleared (worst case)
1K	16368	8184
4K (ARM & x86)	65520	32760
16K	262128	131064
64K	1048560	524280

Fortunately, the larger page sizes are quite uncommon. Furthermore, under the assumption that not all memory is always accessed by the callee, we may use *lazy copying* to reduce the overhead of memory copying: If the hypervisor makes no effort to set up the unaligned pages, an access violation will be reported if the callee attempts to access these pages. Only then will the hypervisor set up the offending page and copy the corresponding data to it. The performance impact of this will be demonstrated in Section 6.4.2

6.4 EVALUATION

For performance evaluation, we will use a simple synthetic benchmark consisting of the calls shown below

```
NULL():
    do nothing

SUM(OUT Value v1; IN Buffer b1):
    v1 := sum of all bytes in B1

COPY(IN Buffer b1; OUT Buffer b2):
    bytewise copy b1 to b2
```

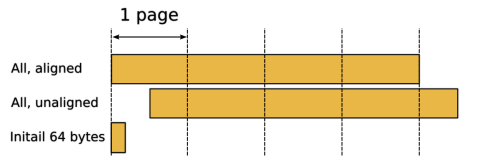

Figure 6.7: Different buffer configurations used in the benchmark.

```
ADD(IN Buffer b1; INOUT Buffer b2):
    bytewise add b1 to b2

MUX(In Value v1; IN Buffer b1, b2; OUT Buffer b3):
   if v1 = 0: copy b1 to b3, otherwise copy b2 to b3
```

These calls have been chosen to highlight different characteristics of the implementation. For example the *NULL* call will help us measure the remote call overhead while *MUX*, which uses only one out of two input buffers, should highlight the benefits of lazy mapping.

As illustrated in Figure 6.7, we consider three buffer types: 16*KB* buffers that are fully *aligned* to page boundaries, 16*KB* *unaligned* buffers that start/end in the middle of a page and *small* 64 byte buffers (to simulate situations where the transmitted data is much smaller than a page).

6.4.1 EVALUATED IMPLEMENTATIONS

We consider the implementations shown in figure 6.8. Among these, shared memory is tested with SELinux enabled or disabled (*SE-Shmem* and *Shmem*). Hypervisor is tested in three variations with regard to unaligned pages: not handled (*HV-plain*), always copied (*HV-copy*) and lazily copied (*HV-lazy*). The native and pipe implementations are included for reference.

6.4.2 PERFORMANCE RESULTS

For measurement, we have used two Linux systems running on ARM Cortex-A8 and Intel Core 2. All numbers have been normalized to that of an ARM running at 200 MHz. For reference, execution time for the native case is shown Table 6.2.

Table 6.2: Execution time of the native implementation on ARM (in μs).

Buffer	NULL	SUM	ADD	COPY	MUX
aligned	≈ 1	411	412	575	576
unaligned	≈ 1	412	411	575	576
small	≈ 1	3	2.50	4.00	3.25

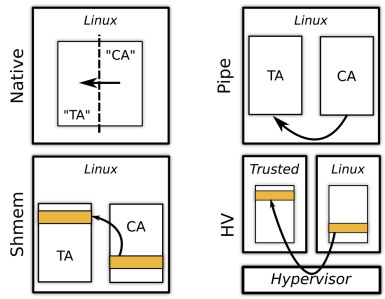

Figure 6.8: The evaluated implementations are: *Native:* native call implementation (local, no protection), *Pipe:* two processes communicating over a pipe or a socket, *Shmem:* two processes communicating using shared memories and *HV:* hypervisor implementation.

Table 6.3 contains the normalized execution time for 16*KB* aligned buffers. As expected, any form of RPC adds a significant overhead to the NULL call.

It should be mentioned that *Native* and *HV* have negligible session setup time. In contrast, the setup time of *Shmem* and *Pipe* dominate the total execution time. This is very visible if we are measuring a single RPC call (denoted $N = 1$) per RPC session. This may however not be how ones embedded application works. If we instead assume that $N = 1000$ calls are made per session, the setup time is no longer significant.

Another issue that must be mentioned is the running time fluctuation (jitter) in *Shmem* and *SE-Shmem*. This was very visible for $N = 1$ and may possibly be attributed to internal kernel mechanisms such caches and pools of various kinds.

Table 6.4 highlights the cost of using unaligned buffers. We initially feared that copying buffers would significantly decrease performance but our measurement showed that this was not the case.

As we already have seen, the remote procedure call adds a significant overhead to the execution time. This is even more visible if we use small buffers, as shown in Table 6.5.

In this case, the effect of unaligned buffers and lazy mapping is more visible.

6.4.3 IMPLEMENTATION EFFORT

It can be argued that complex code is insecure code [NBZ06] [SW11]. It is therefore also important to evaluate the complexity of our implementations. Furthermore, we would like to investigate how critical each implementation

Table 6.3: Aligned buffers, normalized against native. The NULL column is approximate.

Impl.	NULL	SUM	ADD	COPY	MUX
	N = 1				
Native	1.00	1.00	1.00	1.00	1.00
HV-plain	54.75	1.14	1.14	1.10	1.10
Shmem	52.99	1.85	3.94	3.06	3.99
SE-Shmem	80.59	2.14	4.50	3.47	4.48
	N = 1000				
Native	1.00	1.00	1.00	1.00	1.00
Shmem	27.25	1.31	1.22	1.16	1.18
SE-Shmem	27.28	1.31	1.22	1.16	1.18
Pipe	182.00	2.75	3.04	3.21	3.37

Table 6.4: Unaligned buffers, normalized against native ($N = 1000$).

Impl.	NULL	SUM	ADD	COPY	MUX
Native	1.00	1.00	1.00	1.00	1.00
HV-plain	54.75	1.14	1.14	1.10	1.10
HV-copy	54.75	1.15	1.18	1.13	1.13
HV-lazy	54.75	1.15	1.17	1.12	1.13

Table 6.5: Small buffers, normalized against native ($N = 1000$).

Impl.	NULL	SUM	ADD	COPY	MUX
Native	1.00	1.00	1.00	1.00	1.00
HV-plain	54.75	23.30	24.40	15.19	19.23
HV-copy	54.75	25.40	29.10	18.25	24.69
HV-lazy	54.75	24.80	27.10	17.25	22.23
Shmem	58.00	37.00	45.50	18.80	35.00
Pipe	184.0	291.8	339.9	321.3	426.8

is, i.e. how much damage a vulnerability can cause.

In Table 6.6, lines of code (including empty lines, comments, *etc.*), complexity and criticality are shown. The last two are subjective estimations based on the class of technology each component belongs to. Complexity scales from 1 (trivial) to 5 (extremely complex) and criticality ranges from 1 to 5, where 4 and 5 denote ability to bypass the OS and hypervisor security mechanisms.

Table 6.6: Implementation effort and complexity, excluding common parts. Estimated size of related software projects is also shown for reference.

Impl.	SLOC	Complexity	Criticality
Native	245	1	1
Pipe	1426	3	3
Shmem	1265	2	1
HV CA	241	1	1
HV TA	67	1	2
Shared code	1505	-	-
SELinux policies	74	3	3
HV (in hypervisor)	826	4	5
STH	1-4K	-	-
Linux kernel	12-15M	-	-
SELinux	23K	-	-

As a side-note, the SELinux policy required around 20 rules.

6.4.4 SECURITY CONSIDERATIONS

From a security point-of-view, SELinux depends on the Linux kernel and is susceptible to vulnerabilities inside the kernel [Tin09]. For example, an attacker may be able to disable SELinux and by extension the TEE, rendering the effective security of the TEE useless.

The hypervisor on the other hand does not depend on the security of the guest OS, and has a much smaller code base which makes development and evaluation easier. On the other hand our RPC implementation resides inside the hypervisor, directly manipulating the virtual memory map and overriding the OS security. It also makes the hypervisor noticeably larger, which by some definitions automatically decreases the security of the hypervisor.

6.5 CONCLUSIONS

Along with the traditional pipes/sockets and shared memory RPC implementations, two secure implementations of GlobalPlatform's Client API have been demonstrated in this chapter: one assisted by a hypervisor and the other supervised by SELinux.

In terms of security, we note that the SELinux implementation is susceptible to kernel bugs. On the other hand, the hypervisor implementation adds a relatively significant amount of code to the hypervisor itself, which may translate to decreased security.

In terms of performance, the hypervisor implementation (even in presence of unaligned buffers) was the winner. The shared memory implementation was slightly slower (or much slower if setup time was included) and use of SELinux made this implementation only slightly slower (or significantly slower if setup time was included). Both implementations were however faster than the traditional pipe/socket implementation. We also noted that while the measured hypervisor execution times were very stable, the shared memory and SELinux numbers fluctuated. This could make the hypervisor implementation a better candidate for applications where real-time characteristics is of importance.

What SELinux loses in performance it gains in flexibility and availability, hypervisors are hardware dependent and often require changes to the Linux kernel to achieve acceptable performance. If you are currently using a Linux system, chances are you already have access to SELinux. These days this even applies to small embedded systems, with the availability of SEAndroid on Android smartphones as the prime example [SC13].

7

Blockchain for the Keyless Signing Infrastructure

Blockchains have emerged as widely used databases of linked blocks. While the original goal of blockchains was to secure and link transactions in Bitcoin [Nak08] and other cryptocurrencies, blockchains have developed into more general platforms, suitable for, among other things, timestamping and signing information. Blocks are issued frequently, once per 10 minutes on average in Bitcoin, and by linking blocks together it is practically infeasible to rearrange them or change information in them without having access to a majority of the computational resources that secures the blockchain. Even then there is a large cost to rewrite history, where we refer history as data in older blocks. Blockchains are explained in more depth in Chapter 3.

It will become clear that the goals of KSI [BKL13] can be combined with the functionality of blockchains. Specifically, in this chapter we show how to use a blockchain in order to mitigate some of the limitations in the KSI construction. In particular, we add a publishing layer that utilizes a blockchain which allows more frequent publication of root hashes, as well as simplified methods for verifying these hashes. Using the scripting functionality of blockchain transactions, we also show how the KSI operator can strengthen its operational security by requiring multiple signatures to actually publish hashes. While the Bitcoin blockchain is the most widely used blockchain, and will be the targeted blockchain in this chapter, our proposed solution is not necessarily limited to this blockchain.

In Chapter 3 we discussed the details of how the blockchain works.

7.1 KEYLESS SIGNING INFRASTRUCTURE

The Keyless Signing Infrastructure (KSI) system was proposed in [BKL13]. KSI provides a means to timestamp data on a large scale. The timestamps can be verified by anyone and the supporting infrastructure ensures that modifications of documents or timestamps are infeasible. In that sense, the timestamps can be seen as server-based signatures without non-repudiation. Different from ordinary signatures, where the user can sign a document offline, here a server must participate in the computation and publication of the signature. Keyless, here, means that the signatures can be verified without relying on the secrecy of a private key. A reliable and unforgeable timestamp will ensure detection of any attempt to modify the documents.

The security of timestamps and the simplicity of implementation and verification relies on the periodic publication of root hashes in widely witnessed media. The widely witnessed media is proposed to be, among others, newspapers, public forums and micro-blogging platforms [BS14]. To avoid too frequent publication, for example in newspapers, a frequency of once per month has been proposed [BKL13].

The KSI is based an a set of servers, ultimately producing an unforgeable timestamp, i.e., a proof of existence. The timestamps are constructed through a globally distributed Merkle hash tree [Mer88], or more specifically, a hierarchy of hash trees with roots propagating through several layers. Clients provide data, or hashes of data, that they want timestamped to a *gateway*. The data from multiple clients are aggregated in a hash tree and the root is sent to the next layer, an *aggregation layer*. Several servers, typically geographically distributed, serve as aggregation layer in order to minimize delays. The root hash in one layer serves as a leaf of a Merkle tree in the next layer, ultimately reaching a globally unique root hash in the *core layer* of the hierarchy. Such a root hash is generated regularly, e.g, once per second in the current implementation [BKL13], building a calendar of root hashes.

When the root hash has been computed in the core layer, the hash path needed to compute the root hash from the submitted value is propagated down through the network. A calendar hash is the root hash of all documents submitted for timestamping by clients for a particular second. The calendar hash together with the hash path needed to compute the calendar hash from the given document can be used to verify that the document has been correctly timestamped. The hash path can be stored with the document and anyone can query a particular calendar hash in order to verify the integrity and timestamp of a given document. By distributing the calendar hashes to all participants, a successful forgery by reordering or manipulation of calendar hashes will require cooperation between all participants (or at least those that are used when verifying timestamps). In order to obtain a trusted calendar hash, or

sequence of hashes, this has to be obtained from several sources.

To simplify this procedure, periodically, a calendar root hash (CRH), which is the root of the Merkle tree consisting of calendar hashes, is published in widely witnessed media. This can be newspapers, public forums or micro blogs. The hash path from the calendar hash to the published CRH can be added to the document hash path. Now, assuming that enough participating servers verify the published CRH, a document can be verified only using the published value. Thus, the published value adds important aspects to KSI.

- The KSI infrastructure is no longer needed in order to verify a document
- The need to query several participating servers in order to obtain a trusted calendar hash is removed.

7.1.1 LIMITATIONS OF KSI

The hash paths used by KSI to verify timestamps rely on the root hashes being trusted. This is achieved by distributing them to all participants. Storing all in a central database would require this database to be fully trusted. Publishing CRHs in newspapers, forums or micro-blogging platforms once per month will simplify the procedure of verifying root hashes and it will also simplify the trust model since anyone can verify the published value and object if it looks manipulated.

However, publishing CRHs in newspapers is rather inefficient. First, the frequency of publication is at best once per day. Second, verifying a hash value printed on paper requires some form of manual process to digitize it. Using centralized micro-blogging platforms or public forums is prone to denial-of-service attacks [vB09] and could be the target of malicious attackers. With infrequent publications, a verifier must trust the subset of servers that are queried before a calendar hash path and CRH can be used for verification. An additional disadvantage of infrequent publishing is that the timestamp on the publishing medium, e.g., date in the newspaper, only guarantees that the data was available before that date, but will not give more granular guarantees about it. For more granular information, trust in a third party will be needed.

These limitations can be solved by publishing the CRHs in a blockchain. A blockchain provides a possibility for much more frequent publication as new blocks are typically added several times per hour. They also provide a standardized interface to allow simple verification and retrieval of values. Moreover, their integrity, once enough blocks have been added, is guaranteed by a distributed consensus involving a large amount of servers and miners.

Utilizing an open-access blockchain will reduce the trust requirements of the KSI operator and aggregators, as clients are able to independently verify the CRHs as they are included in the blockchain.

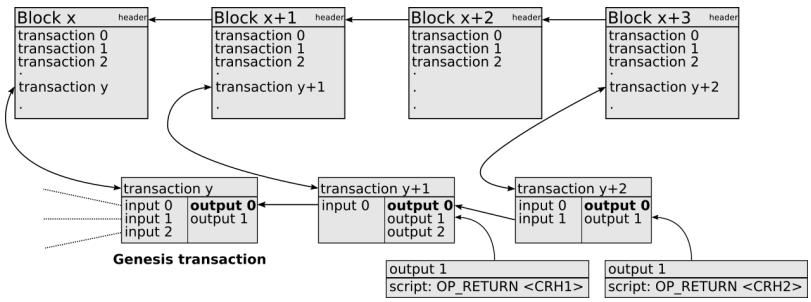

Figure 7.1: Clients track the first output in each transaction to retrieve a deterministic and ordered chain of transactions with CRHs in the second output of each transaction.

We aim to solve the following limitations in this thesis:

1. No authentication of published CRHs.
2. Infrequent publication of CRHs.
3. Inconvenient to independently verify CRHs.
4. No possibility to verify CRH before publication.

7.2 DESIGN

The naive approach of embedding hashes in the blockchain is to simply create a transaction with the commitment value included in an `OP_RETURN` output and have that transaction included in a block. KSI clients are then notified of which block and transaction the CRH is included in to allow them to independently verify that their data has been included. This approach has two notable drawbacks. First, nothing stops the KSI operator from including multiple variants of CRHs and reporting different values to different clients. This would break consistency of having all hashes collected under the same tree and may open up attack vectors. Secondly it forces clients to always keep references to the specific block and transaction that include the hash for any future verification needs of their data. Additionally there is no way to authenticate that the KSI operator actually was the entity that included the hash in the blockchain. Each CRH would have to be accompanied by authentication data, which would be cumbersome to maintain. Authentication is important for many reasons. In the case of a chain reorganization we want to make sure that the KSI operator does not fraudulently change any of the values. If only

they can sign transactions with the embedded CRHs, we can provide proofs of fraud in an event where they try to alter the CRHs.

A better solution is for the KSI operator, at the start of its operations, to signal one transaction that it has control over. This transaction will be dubbed the genesis transaction. This transaction will mark the beginning of a transaction chain that the KSI operator will have sole control over. When a CRH should be embedded in the blockchain, the KSI operator creates a new transaction that spends the first output of the genesis transaction, adds a new output, which will be the next output to track, and one additional output with the CRH itself. For each new CRH to be added, the KSI operator creates a new transaction that spends the first output from the last transaction in the transaction chain that begins with the genesis transaction.

Figure 7.1 illustrates these concepts. Transaction y is the genesis transaction. Transaction y+1 spends the first output via input 0 and includes the first CRH in output 1. When output 0 of transaction y+1 is spent, a new transaction with a new CRH and a new tracking output is created. Note how it is not necessary for a transaction to be included in every block. The KSI operator can make as many or as few transactions as they deem necessary. The tracking of the transaction chain is unaffected. If there is a need to include more funds in the transaction for fees, it is possible to include more inputs. It does not matter which input spends the tracking output, as long as one of them does. It is also possible to include more outputs after the first two. The first output in each transaction of the transaction chain is used for tracking the chain, while all other outputs are ignored.

With this design clients in a KSI infrastructure can deterministically track CRHs with the only prerequisite knowledge being the genesis transaction. Timestamps in the block headers provide a rough idea of the time when the CRH was included, and clients can be assured, with strong guarantees, that if a chain reorganization occurs and the KSI operator attempts to tamper with orphaned CRHs, they can also be held accountable by providing proof of the orphaned chain with differing values of the commited CRH.

If there are a lot of clients utilizing the services provided by the KSI, there will be wide consensus on which transaction is the genesis transaction. Verification of CRHs in a future where the KSI may be out of business could retain the trust in the per-second granularity it provided during its time. It should be easy to know and trust which transaction marked the first in the KSIs transaction chain. If the CRHs were commited and reported individually, outside a transaction chain, the assurances would be weaker because only the clients that commited data in that individual block would keep track of it. There could also be alternative commitments. Commitments in a transaction chain have presumably been validated by all its participants, can be authenticated to originating from the genesis transaction and can thus be more trustworthy.

7.2.1 BROADCASTING CRHS

One question is when to broadcast transactions with CRHs to be added to the blockchain. We discuss two options here. The simplest, cheapest and most manageable seems to be for the publisher to continue to add calendar hashes as leaves to a tree until the previous CRH has been included in a block, and only after that broadcast a transaction with the new CRH. When this transaction has been broadcast to the blockchain network, the publisher continues to work on a new hash tree, the root hash of which will be broadcast when the previous one has been included in a block. This approach has the advantage of only requiring one transaction and transaction fee per CRH and a simple implementation. The disadvantage however is because block creation is a random process, it could take a long time before the current hashes are commited to the blockchain. For some clients it may be important to have the CRH that commits their data included in a block as soon as possible.

To accommodate this, the KSI operator could perform transaction replacement. Transaction replacement is the act of replacing a previously broadcasted transaction, that has not been included in a block yet, with a new version. The new version will need a higher fee attached to it compared to the previous version in order to be reliably forwarded by nodes on the network and eventually reach miners.

Another approach would be to spend the unconfirmed transaction over and over, thus creating a chain of unconfirmed transactions each with their own CRH attached to it. When a block is created the whole chain of transactions could be included. However, this approach bloats the blockchain which is a limited resource, and the replace-by-fee [HT15] approach should be preferred.

Figure 7.2 illustrates how the publishing layer collects calendar hashes to create CRHs for publishing on the blockchain.

7.2.2 COMPLEX SPENDING CONDITIONS

In its most simple form, the condition to spend an output will only require a signature of the transaction from the private key related to the specified public key and the public key itself. These scripts can be arbitrarily complex though to cover various failover modes like key-loss or key-theft. Most common is to make use of multi-signature scripts, as described in Section 3.2.1, to require a threshold of signatures for an output to be spent. This provides the KSI operator with the ability to strengthen its operational security. Should one of the keys be lost or stolen, the compromised key cannot by itself spend the output. The remaining, uncompromised, keys can still spend the output and specify a new spending condition that excludes the compromised key.

Spending conditions can be even more complex. We can assign multiple groups, each with a set of signers, with different conditionals for when and

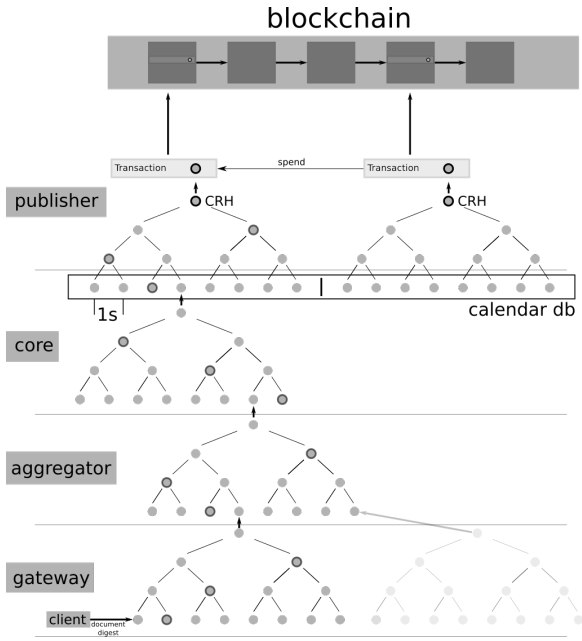

Figure 7.2: Overview of KSI and the proposed publishing layer. Publisher collects calendar hashes and eventually tries to publish root hashes with the calendar hashes as leaves in a Merkle tree. The marked nodes in the trees is everything the client needs to verify that their document digest is part of the CRH

how they may make a spend. Should something happen to the top priority group, the second group of signers can take over operations after some amount of time has passed without the output having been spent. Such a spending condition could be a 2-of-3 multisignature script from the primary group or 3-of-4 multisignature script from a failover group, with the condition that the output has not been spent until a specific time. The failover group would not be able to spend the output unless the primary group fails to make a spend until that time. Any number of groups with different priorities and timeouts can be set to cover multiple failures.

Below follows an example redeem script of how three groups (A,B,C) have spending permissions at different times. This redeem script will be supplied in an input together with appropriate signatures to succeed in spending it. Group A will always be able to spend the output, and will require 2 valid signatures from any 3 specified public keys. Group B will be able to spend the output only after time X, and require 3 valid signatures given from any

4 specified public keys. Group C will be able to spend the output only after time Y, where Y occurs after X, and requires only 1 valid signature from any of the 5 specified public keys.

```
OP_2
<A1.pubkey>
<A2.pubkey>
<A3.pubkey>
OP_3
OP_CHECKMULTISIG
OP_NOTIF
    OP_3
    <B1.pubkey>
    <B2.pubkey>
    <B3.pubkey>
    <B4.pubkey>
    OP_4
    OP_CHECKMULTISIG
    OP_NOTIF
        OP_1
        <C1.pubkey>
        <C2.pubkey>
        <C3.pubkey>
        <C4.pubkey>
        <C5.pubkey>
        OP_5
        OP_CHECKMULTISIG
        <time_Y>
        OP_CHECKLOCKTIMEVERIFY
        OP_DROP
    OP_ELSE
        <time_X>
        OP_CHECKLOCKTIMEVERIFY
        OP_DROP
        OP_TRUE
    OP_ENDIF
OP_ELSE
    OP_TRUE
OP_ENDIF
```

The publishing layer of the KSI could be merged with other stakeholders of the KSI. Group A could, for example, be the core cluster of the KSI, while

group B includes participants from the aggregator layer, ready to step in if there is a failure. Group C could be a set of trusted functionaries that will reboot the transaction chain if and when Groups A and B fail.

An alternative setup is that group A's conditions could require a signature from a core cluster node and a threshold signature from several aggregator nodes. This would increase the trust of the CRH before it is included in a block, given that the core cluster and various aggregator nodes are independent of each other, thus minimizing the risk of collusion. These types of scripts address the limitation of the KSIs centralized trust model to become more decentralized by including independent parties to sign transactions.

7.3 FURTHER CONSIDERATIONS

Blockchain technology is far from mature, and a number of development efforts exist to extend and enhance them. Several proposed features could be adopted to improve the proposed design described in this chapter.

7.3.1 MERKLEIZED ABSTRACT SYNTAX TREES

Improvements to the scripting system, such as Merkleized Abstract Syntax Trees (MAST) [RNS], can be used to enhance our use case, including confidentiality of unused spending conditions and reduced on-chain footprint. A MAST encodes mutually exclusive spending conditions of a script as separate branches of a Merkle tree where only the branch used will be revealed.

Currently the entire redeem script will have to be included in order for the transaction to be verified correctly, exposing unused spending conditions in the process. Confidentiality of unused spending conditions will allow the KSI operator to keep critical operational security information hidden from potential attackers that will try to harvest as much information as possible. It will also reduce the size of large redeem scripts, since only the condition used to make the spend will have to be revealed, including intermediate hash values to correctly verify the Merkle root of the tree. Smaller redeem scripts lowers transaction fees and minimizes blockchain bloat. Furthermore, confidentiality of spending conditions can be kept even between the different groups that have spending permissions, since they only need to be aware of their own condition. One party must obviously be aware of all conditions since someone must generate the script.

One additional improvement that MASTs can provide for our use case is that the CRH commitment can be bundled in the tracking output, hiding the fact that hash commitments are occurring in that transaction for non-participants, as well as enabling additional reduction of the transaction footprint on the blockchain.

7.3.2 RELATIVE LOCKTIME

The `OP_CHECKLOCKTIMEVERIFY` enables spending conditions to be valid at some absolute time in the future. A relative locktime, where an output becomes spendable some specific amount of time after it has entered the blockchain can improve our proposed solution.

With an absolute locktime, such as with `OP_CHECKLOCKTIMEVERIFY`, we have to calculate a new absolute time to use in all newly created transactions. With a relative locktime variant we can select an appropriate relative time for the specific conditions and reuse them without having to recalculate absolute times for each new transaction.

Relative locktime has been implemented [MFk15] [BL15] and deployed on the Bitcoin network.

7.4 DISCUSSION

The functionality provided by the proposed publishing layer of the KSI could be merged with the core cluster and other parts of the KSI. Since the core cluster consists of many servers that run their own agreement protocol to decide what value to commit to, each of them could hold one signing key and sign transactions including these CRHs directly, rather than outsourcing this work to some other set of servers. The publishing layer can also be merged with aggregators and gateways and form complex spending conditions to assure clients further that no single independent organization has control over the values embedded on the blockchain. There are incentives for the KSI operator and other parties to not try tampering with values in circumstances such as chain reorganizations. Since transactions are basically flooded to the blockchain network, and a spend from a tracking output enumerates exactly which spending condition was utilized, the responsible party/parties can be held accountable. Such an attempt at fraud could make people question the integrity of the operators.

There are some disadvantages to using a chain of transactions which includes the CRHs. Since the transaction chain can be uniquely identified, miners could choose to force the KSI operator to pay larger fees, or even outright try to censor transactions that is trying to spend from these outputs. Such a scenario could seriously degrade the reliability of the system. If the computational power of the network is sufficiently decentralized, censorship by some of the miners should not be a problem. There is also a potential bootstrapping attack where the KSI operator could signal multiple genesis transactions to different clients. As time passes and as more clients connect to the system, this should be less of an issue. Clients can verify the genesis transaction with other peers if necessary.

Blockchain technology is still in its infancy, and it is an open question as to how such a system will be secured in the future when block rewards diminish. An idea is that transaction fees will provide the incentive for miners to continue building blocks. Since the KSI wants to make many transactions per day, it could pose an economical issue in the future if transaction fees become more expensive than they are today.

Using the approach of transaction replacement for CRH propagation on the blockchain solves the problem of having to let clients wait unnecessarily long for a block to be found to send out a new transaction. It does however come with some its own set of drawbacks. The cost is greater since more transaction fees need to be added for each transaction replacement. This could be mitigated by allowing clients to pay premiums to have their data embedded earlier. Additionally, at the time of writing the replace-by-fee [HT15] policy is somewhat controversial, and should it not be widely adopted, this approach could prove to be unreliable to use. Should an old version of a transaction be included instead of the latest one, the publisher should start the next tree beginning with the excluded values.

The tracking of CRHs on the blockchain can be implemented utilizing low resources for client-side implementations. If a client is not using the blockchain for something other than verifying CRHs, they only need to download the block headers in order to verify the proof-of-work, and the specific transactions they are interested in. The transactions are provided with Merkle branches to prove that they in fact are commited in a specific block header and all other transactions in that block can be ignored. We can be sure that all transactions have been provided since missing transactions will be evident due to breaks in the tracking outputs in the transaction chain.

Timestamping documents with minimal trust requirements, including security from back-dating and forward-dating, using Merkle trees to prove inclusion of a document was studied in [MAQ99] [HS90] [BHS93]. New signature schemes utilizing quantum-immune primitives for KSI clients is discussed in [BLT14]. It provides extensions where clients can authenticate themselves via one-time keys using hash-chains. Revealing such OTKs could be problematic if the signed data is not sufficiently secure from tampering. Commitcoin [CE12] presents the use of the Bitcoin blockchain as a means to publish proof of a document's existence at a specific point in time.

7.5 CONCLUSION

Several limitations related to the previously proposed publishing channels for calendar root hashes (CRHs) in the keyless signing infrastructure have been identified. We have shown that by using a blockchain as a publication channel

for these root hashes, these limitations have been mitigated and the proposed solution additionally introduces new features to the KSI. The time window in which trust in calendar hashes, and queries of these, are required can be lowered and we can, by using the scripting functionality of transactions, make sure that only one entity is able to publish new CRHs. Moreover, by using multisignatures, it is possible to require explicit verification of a CRH prior to its inclusion in a block. It also adds a fallback if the primary signing group fails.

If KSI clients have no use for the per-second timestamping that KSI provides, they could utilize the blockchain by themselves and timestamp what they need without involving any third parties. If everyone did this, however, it would bloat the blockchain to the point where either fees will be too high for individual pieces of data to be timestamped, or the blockchain grows so large that few people validate its contents and it becomes more dependent on third parties for validation. There is software available [cha] for anyone to run a federated timestamping service on the blockchain which includes document digests from all interested participants and aggregates them into a hash tree, of which the root is the only value included in the blockchain, and proofs are sent back to all clients at that time. This saves blockchain space, as well as transaction fees.

The use of a blockchain has its own limitations, such as transaction fees, and possible attacks on the system in itself, but our proposed publication layer constitutes a viable alternative to newspapers, forums and micro-blogging platforms. It adds simplicity, flexibility and security to KSI and from the perspective of the emerging blockchain technology, it shows yet another application which can take advantage of its properties and power.

8

Creating Hidden Code Using NOP Instructions

The application of analysis evasion techniques has both benign and malicious purposes. License validation code can be protected in order to make it more difficult to analyze the validation procedure. This can delay the development of illegal copies that bypass the license validation. At the other end, there are malware authors that want to prevent analysis of their malware. One way to accomplish this is to confuse the software or the analyst into making wrong conclusions about the code and its behavior. New analysis evasion techniques often also require new, and sometimes ad hoc, ways of testing software for the presence of evasion attempts, e.g., hidden code. For an analyst there is much time to gain from automated detection.

In this chapter we focus on the problem of correct disassembly, and in particular an anti-disassembly technique that aims to trick the disassembler into recovering benign, valid and executable code, which will host the hidden code. The idea is based on using certain instructions in which several bytes can be chosen arbitrarily. We identify several such instructions, but focus on the no-operation (`NOP`) instruction. We show that a large set of instructions can be embedded inside a linear stream of `NOP` instructions. This technique allows a great flexibility in the hidden code and almost any sequence of instructions can be hidden inside the `NOP`s. Furthermore, we discuss how our basic technique can be extended to avoid certain heuristic detection techniques. Some well-known disassemblers, both simple and advanced, are used to test the proposed technique and we show that using our technique, we can successfully hide code from all these tools. Finally, we discuss a testing approach that is tailored to detect the hidden code. We test this detection technique on a proof-of-concept implementation and we successfully detect the hidden instructions with good accuracy.

This chapter is based on the publication titled »A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries«.

8.1 A NEW TECHNIQUE FOR OVERLAPPING INSTRUCTIONS

In this section we give the requirements and the main ideas for our proposed way of overlapping instructions.

8.1.1 REQUIREMENTS

In order to successfully overlap instructions to trick the disassembler into reconstructing the wrong execution path, as described in Section 4.2.2, two requirements must be met.

1. The instructions must overlap each other and must never be aligned such that two instructions end at the same byte.

2. Both execution paths must consist of valid instructions.

Fulfilling both of these requirements is very difficult since an instruction in one path always puts heavy restrictions on the overlapped instruction in the other path. The proposed anti-disassembly technique will meet these two requirements by choosing instructions with certain properties that make it much more manageable to overlap and embed instructions. It will even allow us to choose the instructions in one execution path with much freedom.

In a situation where the program, depending on external or run-time properties, will execute one of the paths, it is crucial that both paths not only produce valid code but that the code is also executable. This will add a third requirement.

3) Both execution paths must consist of executable instructions.

By executable we mean instructions that will, to some extent, guarantee not to crash the program.

8.1.2 OVERVIEW OF THE MAIN IDEA

The goal is to assemble a stream of bytes such that when decoding from two different offsets, two different sets of instructions will emerge, i.e., two different execution paths. The two paths will be denoted Main Execution Path (MEP) and Hidden Execution Path (HEP) respectively. A static disassembler should only recover the MEP up until the point where the two paths converge. A dynamic disassembler, which will decode the actual executed instructions, will in the case when the MEP is executed recover the MEP. Clearly, in the situation when the HEP is executed, for example when the presence of a virtual

machine is not detected, the HEP will be executed and recovered. As malware is often analyzed in a VM, this adds an additional layer of obfuscation for the analyst.

The HEP is the most important execution path, since it should be able to hide arbitrary instructions. Therefore we will put as few restrictions as possible on it and allow it to be as flexible as possible. At the same time, the exact effects the MEP has are not important since its primary function is to hide the HEP. To be able to do this efficiently we identify instructions that have as many bytes that can be arbitrarily chosen as possible. The MEP will consist only of these instructions and they will be coded as `XX YY ZZ`.

`XX` represents instruction prefixes, the opcode and other static bytes part of the instruction that cannot be changed. `YY` includes the dynamic bytes, often describing a memory operand or an immediate value of the instruction. The bytes in `YY` should be large enough to be able to embed a large set of instructions. The `YY` bytes will form the most important part of the HEP. `ZZ` should, just as `YY`, be possible to have any value assigned to it with the only difference that the combination of `ZZ` followed by `XX` must encode to a valid and executable instruction. `ZZ` should preferably consist only of one byte, leaving as many free bytes for `YY` as possible. The combination of `ZZ` and `XX` is denoted the *wrapping instruction*. The wrapping instruction will be executed as part of the HEP and should have little influence over the hidden code. The wrapping instruction is used to glue together the HEP instructions and to separate the MEP and HEP by overlapping two consecutive MEP instructions. Finally, the last HEP instruction should end with `ZZ`, creating a convergence point for the different execution paths.
The MEP will be decoded and executed as

```
Instruction 1: XX YY ZZ
Instruction 2: XX YY ZZ
Instruction 3: XX YY ZZ
```

while the HEP will be executed as

```
Hidden instruction sequence 1:  YY
Wrapping instruction 1:         ZZ XX
Hidden instruction sequence 2:  YY
Wrapping instruction 2:         ZZ XX
Hidden instruction sequence 3:  YY ZZ
```

By starting execution in the first bytes of `XX`, MEP will be executed, while starting execution at `YY` will execute the HEP.

The next section will describe some instructions that could be used to achieve this.

8.2 SUITABLE MEP INSTRUCTIONS

The largest instruction that has the most bytes that can be arbitrarily chosen and still assemble to a valid instruction is a `MOV` instruction where the source operand is a 32-bit immediate value and the destination operand is a memory address specified by a register and a 32-bit immediate value as offset.

```
Encoding: C7 80 10 20 30 40 50 60 70 80
Instruction: MOV DWORD PTR [EAX + 0x40302010], 0x80706050
```

This instruction allows the last eight bytes to be chosen arbitrarily and was used in [LSPM12] to embed a hidden instruction. While this is a valid instruction it is rarely executable since it will typically point to a memory location that is unavailable to the process, resulting in a program crash. Thus, it does not fulfill the third requirement in Section 8.1.1, meaning that it is easily found using dynamic analysis. If VM detection is used by the malware, avoiding HEP execution inside VMs, an analyst using a VM would detect the crash, simplifying the analysis. Allowing executable instructions that do not risk the program to crash will put more restrictions on the possible instructions. We give four other instructions that can be used to this end.

All have several bytes that can be set arbitrarily and have the additional advantage of being executable without failing. These instructions all have four bytes available for the HEP, plus an extra byte for the wrapping instruction.

LEA. Load Effective Address will calculate the memory address in the second operand and store that value in the first operand. Since we can specify a memory operand here without actually doing any memory accesses, it can be used to insert any byte values in the last five bytes.

```
#Example, load address into AX
LEA AX,[EAX+EAX+0x80] 66 66 8D 84 00 80 00 00 00
```

CMOVcc. This instruction performs a MOV operation if a condition is met. For this instruction to be applicable for this technique, the condition must be chosen such that it always fails. Otherwise it may try to access memory that does not exist and result in a segmentation violation.

```
#Example, perform MOV if overflow flag is set
CMOVO AX,[EAX+EAX+0x80] 66 0F 40 84 00 80 00 00 00
```

SETcc. Similar to `CMOV` in that it will set a byte to the value 1 if a condition is met. It has the same problem as `CMOV` as any illegal memory accesses will result in a segmentation violation when the MEP is executed. Caution must be taken to assure the correct conditional is used.

```
#Example, perform SET if overflow flag is set
SETO BYTE [EAX+EAX+0x80] 66 0F 90 84 00 80 00 00 00
```

NOP. A No-Operation instructions can consist of several bytes since it can additionally include e.g., a memory operand. Since no memory is accessed when the instruction is executed, the bytes specifying this operand can be arbitrarily chosen.

```
#Example, 9-byte NOP instruction
NOP WORD PTR [EAX+EAX + 00000000] 66 0F 1F 84 00 00 00 00 00
```

Which instruction to use in the MEP can be situation dependent as they have different properties. Since the HEP instructions will influence the behavior and effects of the MEP instructions, it is most convenient if the MEP has as little side effects as possible. In the remainder of this chapter, we will use the 9-byte `NOP` instruction since it provides features that are not present in the other instructions.

- `NOP` only increments the program counter. The other instructions can affect the CPU state beyond the program counter.
- For `CMOVcc` and `SETcc`, an illegal memory access is likely to arise if the condition is not set to false.
- `LEA` will always update the value of its destination register.

Other instructions that can be used are `PUSH DWORD`, `MOV EAX,DWORD` and so on, but these limits the number of bytes for hidden instructions (length of `YY`) in the HEP to three and will thus not be described any further.

8.3 ASSEMBLING THE HIDDEN EXECUTION PATH

By choosing multi-byte `NOP` instructions in the MEP, we have one valid and executable path. In this section, we show how to properly choose hidden and wrapping instructions such that the HEP is executable and easily manageable.

8.3.1 HIDING CODE IN A LINEAR STREAM OF NOPS

Since the number of bytes at our disposal in a single 9-byte `NOP` instruction is quite limited, we must use multiple `NOP`s to be able to hide any meaningful piece of code. A wrapping instruction between two consecutive `NOP` instructions is needed to assure the correct execution flow of the HEP. The wrapping

Table 8.1: Possible wrapping instructions.

Category	Instruction	ZZ
I	CMP EAX, 0x841F0F66	3D
	TEST EAX, 0x841F0F66	A9
II	PUSH 0x841F0F66	68
	MOV EAX, 0x841F0F66	B8
	MOV ECX, 0x841F0F66	B9
	MOV EDX, 0x841F0F66	BA
	MOV EBX, 0x841F0F66	BB
	MOV ESP, 0x841F0F66	BC
	MOV EBP, 0x841F0F66	BD
	MOV ESI, 0x841F0F66	BE
	MOV EDI, 0x841F0F66	BF
III	ADD EAX,0x841F0F66	05
	OR EAX, 0x841F0F66	0D
	ADC EAX, 0x841F0F66	15
	SBB EAX, 0x841F0F66	1D
	AND EAX, 0x841F0F66	25
	SUB EAX, 0x841F0F66	2D
	XOR EAX, 0x841F0F66	35
IV	MOV AL, BYTE PTR [0x841F0F66]	A0
	MOV EAX, DWORD PTR [0x841F0F66	A1
	MOV BYTE PTR [0x841F0F66], AL	A2
	MOV DWORD PTR [0x841F0F66], EAX	A3
	CALL 0x841F0F66	E8
	JMP 0x841F0F66	E9

instruction will in most cases not be of any use for the HEP and should be chosen to influence the CPU state as little as possible.

To find all possible wrapping instructions, we generated a list of all instructions of the form (`ZZ 66 0F 1F 84`).

```
instruction := ZZ 66 0F 1F 84
for each possible value of ZZ
    disassemble(instruction)
```

In order to have a `NOP` for a wrapping instruction, `ZZ` would have to take up four bytes, leaving only one byte instructions to fit in the HEP. Since we value the flexibility of larger hidden instructions, we will look for alternative wrapping instructions. There are no suitable instructions which have no influence on the state of the machine, when having `ZZ` consist of one byte. However, there are several other instructions that still can be used. Table 8.1 lists the possible wrapping instructions divided into four categories.

Category I includes instructions that change the `EFLAGS` register. These instructions are suitable when the HEP does not use any jumps or other instructions that relies on evaluation of information in the `EFLAGS` register. Only two instructions belong to this category, namely `TEST` and `CMP`.

```
TEST EAX, 0x841F0F66    A9 66 0F 1F 84
CMP EAX, 0x841F0F66     B1 66 0F 1F 84
```

CMP will use subtract to test the operands, while TEST will perform a logical AND operation. It can be noted that the TEST instruction is faster since the logical AND operation is executed faster than subtraction. If efficiency is important this should be taken into consideration. To form the TEST instruction properly for our needs we will have to assign the last byte of the first NOP instruction with the value 0xA9. This byte paired with the first four bytes of the following NOP instruction (66 0F 1F 84) will form the instruction TEST EAX,0x841F0F66. There are four bytes left within each NOP instruction that can include instructions from the HEP. See below for a stream of NOPs and its embedded HEP representation.

```
MEP:
NOP WORD PTR [ESI-0x56FFFE45]       66 0F 1F 84 66 BB 01 00 A9
NOP WORD PTR [ECX+ESI-0x7F32BF40]   66 0F 1F 84 31 C0 40 CD 80
```

The HEP displayed below will be executed if we start executing at the fourth byte of the first NOP. In the example, it is simply a call to the exit system call for any Linux OS with a return value of 1.

```
HEP:
MOV BX,0x0001                       66 BB 01 00
TEST EAX,0x841F0F66                 A9 66 0F 1F 84
XOR EAX,EAX                         31 C0
INC EAX                             40
INT 0x80                            CD 80
```

Category II includes instructions that change the values of general purpose registers or valid memory, like the stack, without updating the EFLAGS register. Using the PUSH instruction or any of the MOV instructions with the source operand being an immediate value does not alter the EFLAGS register. Thus, the wrapping instruction can be placed between a comparison instruction and the instruction evaluating the EFLAGS register, without changing the semantics of that evaluation. Instead, when using these instructions, we must take care to limit the use of the affected register in the rest of the HEP. As an example, MOV EBP,0x841F0F66 can be used as a wrapping instruction. This will limit the use of register EBP in the rest of the HEP. As HEP will mostly be custom assembly code, EBP might not be used as much here as in compiled code, which makes this instruction particularly interesting.

Category III includes instructions that both change the EFLAGS register and updates registers or memory. These instructions have no apparent advantages

over those in categories I and II since they have the limitations of all instructions in those categories. Some of them could be used though, e.g., by using the XOR instruction. Then every other time it is used, EAX will be restored to its original value. ADD and SUB can also be used together to restore the value of EAX if used together.

Category IV includes instructions that cannot be guaranteed to be executable, due to the possibility of illegal memory access.

We have also generated wrapping instructions for the 8-byte and 10-15 byte NOP, but found that the 9-byte NOP gives the best wrapping instructions in terms of maintaining a controllable CPU state.

The main limitation of the (non-wrapping) HEP instructions is the maximum instruction length of four bytes. Except for the last instruction, all instructions are required to be four bytes or less. Still, this requirement can be significantly relaxed as many instructions larger than four bytes can be broken down to multiple instructions of size four or smaller. Below is an example of a MOV instruction that is too large to fit within the HEP.

```
MOV EAX,0x12345678  B8 78 56 34 12  #5 bytes long
```

The five byte MOV instruction can be translated into two 4-byte instructions and one 3-byte instruction.

```
MOV AX,0x5678       66 B8 78 56     #4 bytes long
SHL EAX,0x10        C1 E7 10        #3 bytes long
MOV AX,0x1234       66 B8 34 12     #4 bytes long
```

This reduction in size of instructions significantly increases the flexibility in the choice of HEP instructions, providing e.g., malware authors additional power and possibilities.

8.4 ADDITIONAL PRACTICAL CONSIDERATIONS

As a complement to the approach discussed above, it is possible to take additional measures in order to evade analysis. This section will discuss how the proposed code overlapping technique can be extended in various ways, increasing difficulty of detecting the presence of the HEP.

8.4.1 HIDING CODE IN SCATTERED NOPS

As demonstrated in Section 8.3.1, a large piece of code can be put into the HEP, while at the same time allowing the main execution path to be not only valid assembly code, but also represent code that is executable without crashing the program. Thus, in initial analysis attempts using a linear sweep disassembler,

it is not straight forward to identify the use of a HEP. Still, the potentially long linear NOP stream that is present in the disassembly will probably stand out and raise suspicion. We can improve the stealth of the HEP by scattering NOPs throughout the program. Correct execution flow can be maintained by allocating the last two bytes of each NOP with an unconditional jump instruction to the next hidden instruction embedded within another NOP.

Thus, we can have several short streams of linear NOP instructions, possibly only one at a time, where the last of the instructions contains one instruction with a maximum size of three bytes, followed by the unconditional jump. In this instruction there will only be three bytes available to assign a hidden instruction, limiting the number of instructions that can be used even more than before. It should be noted though that it is still possible to break down some larger instructions to fit this technique. Below is the ADD example from earlier which can be further reduced in size to two and three-byte instructions.

```
#MOV EAX,0x12345678
MOV AL,0x78       B0 78     #2 bytes long
SHL EAX,0x8       C1 E7 08  #3 bytes long
MOV AL,0x56       B0 56     #2 bytes long
SHL EAX,0x8       C1 E7 08  #3 bytes long
MOV AL,0x34       B0 34     #2 bytes long
SHL EAX,0x8       C1 E7 08  #3 bytes long
MOV AL,0x12       B0 12     #2 bytes long
```

By dividing instructions like this, it is possible to use only single NOPs scattered throughout the program. The two-byte JMP instruction can jump forward 127 bytes or backwards 128 bytes in the code, which means that two consecutive NOP instructions, from the HEPs perspective, have a limit on how far away from each other they can be.

This approach has the additional advantage of the possibility to embed hidden instructions in NOPs already existing in compiled binaries. It is not only limited to occurrences of single 9-byte NOP instructions, but it can also be used when there are clusters of single-byte NOP instructions, as these could be converted to multi-byte NOP instructions without changing the semantics of the program. Below is an example of the scattered NOP technique.

```
NOP WORD PTR [ECX+ESI*4+0x4EEB01]   66 0F 1F 84 B1 01 EB 4E 00
...

NOP WORD PTR [ECX+ESI+0x21EB40C0]   66 0F 1F 84 31 C0 40 EB 21
...

NOP WORD PTR [EBP+ECX+0x80]         66 0F 1F 84 CD 80 00 00 00
```

```
.nop1:
MOV BL,0x01                    B1 01
JMP .nop2                      EB 4E
...

.nop2:
XOR EAX,EAX                    31 C0
INC EAX                        40
JMP .nop3                      EB 21
...

.nop3:
INT 0x80                       CD 80
```

Note that all bytes in the first NOP are not filled, but since the last byte is preceded by an unconditional JMP, this byte will never be reached and will not cause a problem.

Using scattered NOPs requires a more detailed analysis than when using a linear stream of NOPs.

8.4.2 NORMALIZATION OF MEP INSTRUCTIONS

So far both a linear NOP stream and the scattered NOPs work very well to hide an alternate path of execution. An analyst will only see the NOPs in a disassembly listing at first. An experienced analyst may however find it odd to see NOP instructions not conforming to Intel's recommendations and might start disassembling from within a NOP instruction and discover the hidden code.

We want some way to make the NOPs look legitimate and there is only one legitimate representation of the 9-byte NOP instruction. To achieve this normalization we will have to use self-modifying code to generate the correct bytes during run-time. The idea is that during static analysis, an analyst will only see the multi-byte NOP instructions in their legitimate representation and when it is time for the hidden instructions to be executed, a decoding routine will be called to generate the HEP and finally jump to it and continue execution.

To be able to decode the HEP correctly we need to store a key for it somewhere. If we store it as it is, it is more likely it will be discovered as they disassemble to valid instructions. XOR, ADD, MOV, OR and similar operations thus cannot be used by the decoder without revealing the instruction bytes in the data section. Instead we propose to use the SUB operation because we can store a value of 0x100 - 0xXX for each key byte where 0xXX is one byte of the hidden instruction sequence.

The addition of self-modifying code allows the NOP instructions to look legitimate and only if the HEP is about to be executed, the HEP will be embedded in the NOP stream. Otherwise the NOPs will look like normal.

To re-create the HEP, we need a key and a decoding routine.

For our proof-of-concept application [Jäm13] the decoding routine which is included in the application looks like this:

```
CALL foo
foo:
POP EAX                       #Retrieve EIP
ADD EAX,offset_to_nop         #Point to first byte of NOP stream
MOV ECX,nop_stream_size       #Number of bytes in NOP stream
LEA ESI,DWORD PTR [EBP+key]   #Address to key in memory
bar:
MOV EDX,BYTE PTR [ESI+ECX]    #Get key-byte
SUB BYTE PTR [EAX + ECX],EDX #Subtract key-byte from code-byte
DEC ECX
CMP ECX,0xFFFFFFFF
JNZ bar
ADD EAX,4                     #Adjust jump target to hit HEP
JMP EAX                       #Jump to HEP
```

To hide the decoding routine, we could embed it within NOPs as well. If the HEP is larger than the decoder routine we would have less NOPs that look suspicious.

8.5 DETECTION

In this section we will present how well the technique holds up against binary analysis tools. We will test it against some different disassemblers and also a binary analysis framework called BAP, which turns the instructions into an intermediate language representation. We will also suggest solutions for how this technique could be detected.

The proof-of-concept application used as an example for testing is a backdoor [Bem13] hidden within a simple "hello world" program. The executable and details on how the backdoor is hidden can be found at [Jäm13].

8.5.1 ANTI-ANALYSIS

The following three analysis tools have been used in the testing.

Objdump [obj] is a utility, part of GNU binutils, and its disassembler employs a very basic linear sweep disassembly algorithm and is a natural starting point

for testing how appropriate this technique is against these types of disassemblers.
IDA pro [hex] is probably the most widely used and advanced disassembler, and for the technique described in this chapter to be a viable option for hiding code, it should hold up against an adversary with IDA pro.
BAP [BAP] is short for Binary Analysis Platform, and one feature of this tool is that it creates an intermediate language representation of the assembly code in the target binary.

The disassembly listings consist of five `NOP` instructions from the example program's MEP.

OBJDUMP

When testing on the objdump utility, the results are very straightforward. Since it uses a linear sweep disassembly algorithm, no branch instruction will be followed and the end result is that the MEP will be shown and the HEP will be hidden.

```
58a:  66 0f 1f 84 6a 01 66  nop  WORD PTR [edx+ebp*2-0x566f99ff]
591:  90 a9
593:  66 0f 1f 84 6a 02 66  nop  WORD PTR [edx+ebp*2-0x566f99fe]
59a:  90 a9
59c:  66 0f 1f 84 89 e1 66  nop  WORD PTR [ecx+ecx*4-0x566f991f]
5a3:  90 a9
5a5:  66 0f 1f 84 cd 80 66  nop  WORD PTR [ebp+ecx*8-0x566f9980]
5ac:  90 a9
5ae:  66 0f 1f 84 89 c2 66  nop  WORD PTR [ecx+ecx*4-0x566f993e]
5b5:  90 a9
```

IDA PRO

With IDA, the adversary has more options to deal with a binary showing a suspicious disassembly output. The following is what is output by default.

```
.text:080485F6 66 0F 1F 84 6A 01 66 90 A9
                    nop     word ptr [edx+ebp*2-566F99FFh]
.text:080485FF 66 0F 1F 84 52 0F 1F 00 A9
                    nop     word ptr [edx+edx*2-56FFE0F1h]
.text:08048608 66 0F 1F 84 89 E1 66 90 A9
                    nop     word ptr [ecx+ecx*4-566F991Fh]
.text:08048611 66 0F 1F 84 CD 80 66 90 A9
                    nop     word ptr [ebp+ecx*8-566F9980h]
```

```
.text:0804861A 66 0F 1F 84 31 C0 66 90 A9
                    nop     word ptr [ecx+esi-566F9940h]
```

The MEP is shown, just as we wanted. The reason the MEP is shown is because we used an opaque predicate to execute HEP. With IDA we can however define some bytes as data and start disassembly from the first byte of the HEP. When doing so the following disassembly output is listed.

```
.text:080485FA 6A 01           push    1
.text:080485FC                 db      66h
.text:080485FC 66 90           nop
.text:080485FE A9 66 0F 1F 84  test    eax, 841F0F66h
.text:08048603 52              push    edx
.text:08048604 0F 1F 00        nop     dword ptr [eax]
.text:08048607 A9 66 0F 1F 84  test    eax, 841F0F66h
.text:0804860C 89 E1           mov     ecx, esp
.text:0804860E                 db      66h
.text:0804860E 66 90           nop
.text:08048610 A9 66 0F 1F 84  test    eax, 841F0F66h
.text:08048615 CD 80           int     80h
.text:08048617                 db      66h
.text:08048617 66 90           nop
.text:08048619 A9 66 0F 1F 84  test    eax, 841F0F66h
.text:0804861E 31 C0           xor     eax, eax
.text:08048620                 db      66h
.text:08048620 66 90           nop
```

If the analyst knows what he is looking for, the HEP will be exposed. To combat this one or more of the techniques listed in Section 8.4 can be used to further confuse the analyst. For instance, by using the scattered NOPs technique the NOPs will not stand out as much as if they are clustered. If the NOPs are normalized, they will not look as suspicious either, even if gathered in a large cluster.

BAP

With BAP the interesting thing is how our MEP and HEP are represented in its intermediate language representation. The following output is from BAP

```
addr 0x8048377 @asm ‘‘nop’’
label pc_0x8048377
addr 0x8048378 @asm ‘‘nop’’
```

```
label pc_0x8048378
addr 0x8048379 @asm ‘‘nop’’
label pc_0x8048379
addr 0x804837a @asm ‘‘nop’’
label pc_0x804837a
addr 0x804837b @asm ‘‘nop’’
label pc_0x804837b
```

Note that each NOP is listed as being one byte. BAP uses the MEP in its intermediate language representation and the HEP is completely removed. Any further analysis in BAP will not reveal anything about the HEP.

8.5.2 DETECTION ALGORITHM

A general method to find hidden code is to look for long sequences of instructions, all of which are unaligned from instructions in the main execution path. The following algorithm can be used to do this.

```
find_HEP(threshold)
    foreach instruction in text-segment
        for i := 1; i < instruction.size; i++
            count := 0
            hidden := disassemble(instruction+i)
            while not in_MEP(hidden) and valid(hidden)
                count++
                hidden := disassemble_next(hidden)
            if count > threshold
                add_to_HEP(instruction+i)
```

This algorithm will try to assemble a stream of bytes into a valid instruction from every possible offset within all instructions of the MEP in the code section of an executable. It will continue to assemble instructions until it has assembled an instruction that is in the MEP or is an invalid instruction. It will look for a certain number of instructions, the threshold, before it considers the instruction stream as a HEP. Choosing this threshold value is tricky since legitimate code can still include a great number of hidden instructions in a row before the disassembly resynchronizes to the MEP. [LD03] shows that the greatest number of instructions found, which weren't part of the MEP, is 27 instructions out of 360,000. This was found in the compiler gcc.

The larger the HEP is, the easier it is to find, as we can set the threshold to a higher number, thus increasing our chances of avoiding false positives. Setting the threshold too high could make the HEP go undetected as well. The best way to avoid false negatives is to start with a high value and then

decrease the threshold by one until something is found. It is when the HEP is small that it becomes difficult to detect, since the smaller the HEP is, more false positives will arise and it is up the analyst to manually go through all instructions and filter out the false positives.

This algorithm works for all kinds of MEP instructions, not just `NOP`s, as it considers all possible offsets in all instructions as a possible starting point for hidden instructions. It even works for a combination of several different MEP instructions.

The scattered `NOP`s variation of our technique poses a problem for the algorithm, as only two instructions, the hidden instruction and the `jmp` exists linearly. To detect this we need to alter the algorithm so that it follows all unconditional jumps. It should also take into consideration that the `NOP`s may not be in order of the execution flow of the MEP. The last `NOP` instruction in the MEP could for instance be the entry point of the HEP. The jump instruction could also be a conditional in which case the algorithm would have to be modified to follow both paths.

The algorithm was implemented as an IDA pro script and tested against our example application. The script can be found at [Jäm13].

We started with an initial threshold of 100 and the backdoor was found and was reportedly 154 instructions long. The largest false positive that was reported was 12 instructions long.

If we apply the algorithm on the code when applying the self-modifying code technique to normalize the `NOP`s, it does not detect the HEP anymore. If we were to hide the decoding routine in `NOP`s the threshold would need to be set to a significantly lower value than 100 to be detected. This means that the use of this additional obfuscation could delay detection further, given that the decoding routine has a smaller number of instructions compared to the HEP.

8.6 CONCLUSIONS

We have presented a new technique for anti-disassembly. By using and extending ideas from code embedding and code overlapping we have shown how to overlap two execution paths that are both executable. This does not only complicate analysis using static disassembly with a linear sweep algorithm, but will also make it more difficult to use dynamic analysis since both paths can be executed. Combining this technique with e.g., opaque predicates, self-modifying code and VM detection mechanisms has potential to significantly delay correct disassembly and analysis of e.g., malware, hidden decryption routines and license validation code. Furthermore, we give an algorithm for discovering the hidden execution path by attempting disassembly of code that is offset a number of bytes from the main execution path. This

algorithm can successfully and automatically discover malware that uses our proposed technique, potentially saving both time and resources for an analyst.

In the next Chapter, we will extend the technique of overlapping instructions to create hidden execution paths via the source code alone, and no tampering of the binary post compilation.

9

Exploiting Trust in Deterministic Builds

In this chapter we investigate the problem of adding hidden machine code via the source code instead of directly modifying the binary. Such hidden code, if undetected, would result in hashes and code signatures that are correctly verified as genuine. We address the problem of keeping the semantics of such instructions hidden in the source code as well as in the binary so that both a code review of the source code and static analysis of the binary will not easily reveal the hidden code. This is accomplished by carefully constructing data structures where the offset to a base address can be used to interpret machine instructions. In the binary, these instructions will be hidden from the main execution path and not visible in a disassembly listing.

9.1 HIDING INSTRUCTIONS IN BINARY CODE

In this section we will explain our approach to inject instructions in a program and have its semantics hidden from an analyst in both the source code and the compiled binaries. The semantics should be hidden against both static and dynamic analysis techniques as well as manual and automatic analysis methodologies. Furthermore, the attacker can implement it entirely within the source code and not have to worry about modifying any compiled binaries.

The requirements to make this work is for the attacker to have complete control over the part of the source code where the injected instructions are implemented. It is also necessary to have consensus among any potential co-developers about what configurations, optimizations and similar compiler settings that should be applied to the building process. Finally, the project should be compiled in a reproducible way to ensure that produced binaries are byte-for-byte identical between builders.

The examples in this section are based on a 32-bit Linux system with the compiler GCC 4.9.1 with no optimizations enabled. Some of the examples may not be applicable to other compilers or other versions of GCC, but should give a basic description of our approach. It should be straightforward to adapt the techniques explained here for other compilers and configurations.

For the remainder of this chapter we also assume that the trigger condition used to execute the hidden instructions is chosen and designed in such way that it is sufficiently hard for an automated analysis to identify it [ZW11] [SLGL08]. Therefore, we regard this topic as out of scope and focus on hiding the trigger-based code.

9.1.1 MAIN AND HIDDEN EXECUTION PATHS

We begin by defining the Hidden Execution Path (HEP) and the Main Execution Path (MEP) in the binary. The HEP is defined as a sequence of assembly instructions hidden implicitly in the source code, i.e., the semantics of those hidden instruction can not be seen in the source code. It is also explicitly hidden in the compiled source code's disassembly listing. This is true for both static and dynamic analysis scenarios, as long as the HEP is not executed. The MEP is defined as a sequence of assembly instructions that is implicitly visible in the source code, i.e., the source code generates the expected assembly instructions. The generated assembly instructions are clearly visible in the disassembly listing of the compiled source code. The MEP shows the functionality that we want to show the user and the HEP is the malicious code.

The HEP can include any instruction that is valid. The MEP is limited to the instructions the utilized compiler and associated configurations can generate from the given source code.

9.1.2 BASIC DESIGN

Our approach of inserting hidden instructions in the source code relies on having the compiler generate instructions in the MEP in such a way that overlapping instructions are formed to a semantically correct, predetermined HEP. One limitation is the self-synchronizing nature of x86 instructions due to the Kruskal Count [LRV09], i.e., beginning decoding instructions from different byte offsets will sooner rather than later result in the different execution paths synchronizing to the same one. Designing the HEP in one continuous set of bytes would result in limited functionality. In [JLH13] this was circumvented with unusually formatted `nop` instructions which are not prevalent in compiled code and thus cannot be used here. In order to get around this issue we need to avoid creating the HEP in this way and instead scatter fragments of the HEP throughout the application, where each fragment ends with a branch instruction pointing to the next fragment or synchronizes with the MEP if the

HEP has finished its execution. The ideal HEP fragment includes one or more relevant instructions for the HEP and one branch instruction that will redirect the HEPs control flow to the next fragment, see Figure 9.1.

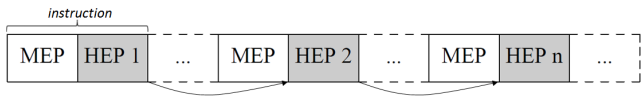

Figure 9.1: Control flow of the HEPs

In Section 9.1.3 we will show how mappings between MEP instruction fields and HEP instructions fields are made with ease of implementation in mind. We will cover how control flow between HEP fragments are handled in Section 9.2.2 and finally in Section 9.3 we discuss what to do to evade potential analysts, both on the source code level and binary level.

9.1.3 MEP-TO-HEP MAPPINGS

MEP instruction fields must be mapped to HEP instruction fields in a way that keep MEP instructions relevant for the program in general and its surrounding instructions. It must also be formed so the relevant HEP instructions are decoded correctly.

To achieve this we must first determine the instruction fields in the MEP that are best suited to encode instructions in the HEP. Looking at a MEP instruction we observe the following.

- Prefixes are very limited in the number of values they can take and will depend heavily on the rest of the fields on whether or not they can be generated. One prefix value can take 11 out of 256 values.

- Opcodes can assume a larger number of bytes, more precisely one-byte opcodes can take 244 out of 256 values (excluding prefixes and the `0x0f` two-byte opcode extension code). The compiler may not be able to generate all of them. Many of the opcodes are redundant, which is why it is likely that compilers are just using a subset of all opcodes. We found this to be true for GCC.

- The `mod-r/m` byte is only one byte and also limited by the compiler in what values it can take. We found that making the compiler use specific registers in the source and destination operands is difficult to achieve.

- The same applies for the `SIB` byte as for the `mod-r/m` byte, in that it is difficult to make the compiler use the specific source and destination operands we want.

- The `displacement` field can be 0, 1, 2 or 4 bytes long and is relatively easy to use to make the compiler generate our desired values. For example, accessing a stack variable will make the compiler generate an instruction that dereferences memory from an offset to the base pointer `ebp`. This offset is the `displacement` field, and depending on how we design the stack layout and what variables we use we can control the values in this field. When using a 32-bit `displacement` field we need to consider a trade-off between the size of a data structure allocated in memory and how much of the `displacement` field we can control. In this work we limit ourselves to using only the three least significant bytes of the 32-bit `displacement` field, making the maximum amount of allocated memory 16,777,216 bytes.

- The `immediate` field is ideal to work with, because we can set it to whatever we want without any practical consequences. It can be 0, 1, 2 or 4 bytes in size by using the types char, short and int respectively. Large constants are not very common in the code section and we will in Section 9.3 discuss how to use the `immediate` field.

Based on this analysis the `displacement` and `immediate` fields of a MEP instruction will be used to create the HEP, see Fig. 9.2. Since the fields are adjacent they can be combined to create HEP instructions.

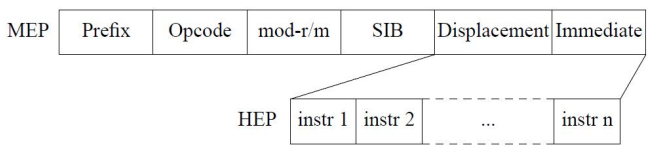

Figure 9.2: Illustration of MEP and HEP layout in an x86 instruction

9.2 CONSTRUCTING THE HEP FROM SOURCE CODE

In this section we describe how to write code that generates the desired hidden code fragments. This is illustrated in Fig. 9.3 where the hidden code fragment is introduced at the source code level to be compiled and later invoked as unintended instructions.

We start by defining two different structs.

```
struct imm8_t {
    char imm8[256][256][256];
};
```

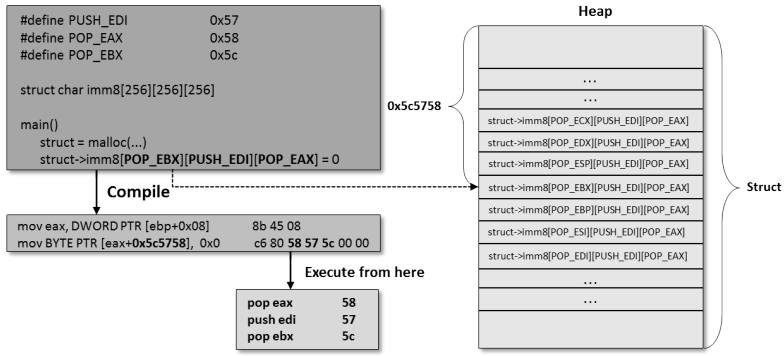

Figure 9.3: Introducing a HEP in the source code. Compilation of the source code will generate unintended instructions which are executed by decoding at certain offsets in the compiled code

```
struct imm32_t {
    int imm32[256][256][64];
};
```

For the rest of this section we will assume the structs are allocated on the heap and when dereferencing any elements in that struct it is done with an offset from the base address of that struct located in register `eax`, see Figure 9.4.

The `imm8_t` struct provides a simple way to assign MEP instruction `displacement` fields with the desired values coupled with an 8-bit `immediate` field. This is because we use struct member 'imm8' which is of type char, the compiler will generate assembly instructions with an 8-bit `immediate` field. Likewise the `imm32_t` will generate instructions with 32-bit `immediate` fields when assigning struct member 'imm32' a value. As an example, assigning a value to the first position in the 'imm8' member in the `imm8_t` struct and the first position in the 'imm32' member of the `imm32_t` struct will generate the following assembly instructions respectively.

```
c6 00 0a                mov    BYTE PTR [eax],0xa
c7 00 0a 00 00 00       mov    DWORD PTR [eax],0xa
```

The first instruction only includes an 8-bit `immediate` field (0x0a) since the variable assigned is of type `char` and the second instruction includes a 32-bit `immediate` field (0x0000000a) because the type is `int`.

In order to generate 8-bit `displacement` fields with these structs the last two indices of either struct member would have to be set to zero. Since the struct

is allocated on the heap, the first index of the struct member must be confined to values between `0x00` and `0x7f`. The reason is that when referencing an element in the struct on the heap, we will do so by using a positive offset from `eax`. If the `displacement` value exceeds `0x7f` the compiler will put the value in a 32-bit `displacement` field in order to not get the incorrect sign it would get in an 8-bit value. Below is an example of two assignments of two elements in the 'imm8' member of the `imm8_t` struct, one where the `displacement` only requires one byte, and the second one where the `displacement` need to be zero-extended in a 32-bit `displacement` field in order to not become a negative number.

```
c6 40 7f 0a             mov    BYTE PTR [eax+0x7f],0xa
c6 80 80 00 00 00 0a    mov    BYTE PTR [eax+0x80],0xa
```

The design of the 3-dimensional matrix corresponding to the struct allows the HEP designer to set the exact bytes wanted of the `displacement` field directly in the source code. For example, the following assignment of the 'imm8' member in the `imm8_t` struct illustrates the process.

```
imm8_t *a = malloc(...);
a->imm8[0x10][0x20][0x30] = 0xa;
```

The compiler generates the following machine code from the assignment statement.

```
c6 80 30 20 10 00 0a  mov  BYTE PTR [eax+0x102030],0xa
```

The indices chosen are seen directly in the decoding of the instruction. Note that it is in little-endian and the bytes are thus reversed.

In Figure 9.4 we can see that when referencing an element on struct member 'imm8', the `displacement` value is derived in the following way.

```
imm8[x][y][z] -> displacement = x<<16 + y<<8 + z
```

Thus, index positions will result in bytes in the `displacement` field.

Worth noting is that with our structs in an instruction with a 32-bit `displacement`, the most significant byte will always be `0x00`, as can be seen in the example above. The reason is that our structs have not allocated enough bytes such that indices can be chosen to specify the most significant byte of the `displacement` field without going out of bounds.

The first index of the `imm32_t` member 'imm32' is set to 64 because it stores integer type values which take up four bytes of memory per element. This means that for each index increment, there will be an increment of four for the least significant byte of the `displacement` value. Thus, any value put into the

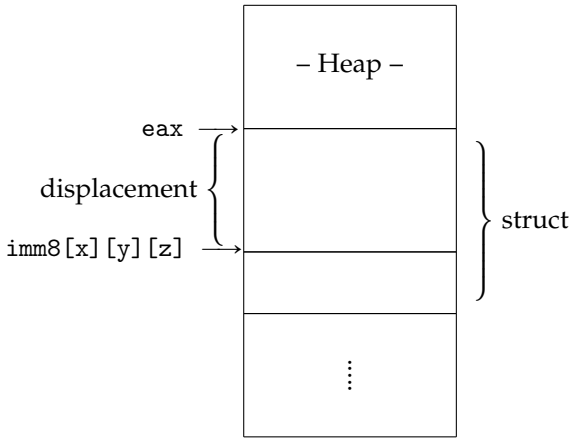

Figure 9.4: Memory layout of struct with displacement to wanted element

first index position must be divided by four in order to get the correct value representation in the `displacement` field. One drawback of using `imm32_t` for encoding HEP instructions is that the first index position cannot assume values not divisible by four, thus limiting its use to only 25% of all byte values. Should this one byte not be useful in the HEP fragment, it could simply be skipped and the HEP fragment's first byte be what is specified in the second index. Below is an example of how the assignment of `imm8[0][0][4]` with `0x0a` is represented in the `displacement` field.

```
c7 40 10 0a 00 00 00    mov    DWORD PTR [eax+0x10],0xa
```

The `displacement` is a single byte with value `0x10` and in order to get the desired value, which is 4, we have to code the assignment statement as `[0][0][4>>2]`. This will generate the following assembly instruction.

```
c7 40 04 0a 00 00 00    mov    DWORD PTR [eax+0x4],0xa
```

Here we can see that the value of the `displacement` field now is 4.

We have defined a set of macros defining all one-byte opcodes and prefixes that can be used to program the HEP. For example, if we want to create the following HEP fragment

```
58          pop eax
57          push edi
c3          retn
```

we would define the following.

```
#define POP_EAX     0x58
#define PUSH_EDI    0x57
#define RETN        0xC3

struct imm8_t *a = malloc(...);
a->imm8[RETN][PUSH_EDI][POP_EAX] = 0;
```

This fragment generates the following machine code for the assignment statement

```
mov eax, DWORD PTR [ebp+0x08]       8b 45 08
mov BYTE PTR [eax+0xc35758],0x0     c6 80 58 57 c3 00 00
```

If execution starts from the third byte in the second instruction, our desired HEP instructions will be executed. We also assume that register `edi` holds an address to some other HEP fragment such that when `edi` is pushed to the stack, the `retn` instruction will pop that value and continue execution from there. In this example the `immediate` value can be set to anything as it will not be a part of the HEP, allowing for some flexibility in designing the MEP.

9.2.1 STRUCTS ALLOCATED ON THE STACK

It is possible to allocate the structs on the stack. This works equally well as long as some measures are taken.

- Instead of having a positive offset from the `eax` register, dereferencing a stack variable does so with a negative offset from the base pointer register `ebp`.
- For 8-bit `displacement` fields we need the value in the first index position to be between `0x80` and `0xff`. This is due to the negative offset used to dereference stack variables.
- Similarly as for the heap, if the value of the `displacement` field drops below `0x80`, the compiler will generate the instruction with a 32-bit `displacement` field and sign-extend it to keep the value negative.
- The first index position need to be adjusted based on what the stack layout looks like. For example, if there is an integer variable allocated before the struct on the stack, the first index need to be added with 4 to make the compiler generate the desired `displacement` value.

9.2.2 CONTROL FLOW

Previous sections showed how we build the HEP and that it is done by fragmenting it and providing branching instructions to direct control flow between the fragments. This section aims to explain some options on how to navigate between these HEP fragments.

INTRA-FUNCTION CONTROL FLOW

It is possible to include all fragments that constitutes the HEP to be confined to a single C function, but it may be difficult to design that function in any meaningful way. The fragments in this case should only need to end with branch instructions that jump short, i.e `jmp rel8`, where the jump can be taken 127 bytes forward or 128 bytes backwards from the instruction pointer.

INTER-FUNCTION CONTROL FLOW

The HEP fragments may also be scattered between multiple functions in the C code. This provides for greater flexibility in forming the MEP in a meaningful way. Assuming reasonably-sized functions, the `jmp rel8` could be used, but the HEP fragments need to be included near the prologues or epilogues of the function to make sure it can reach the next fragment. In other words if one HEP fragment is located in the epilogue of function A then the next fragment should be located in the prologue of function B, where function B comes directly after function A in memory. A more flexible option is to use `jmp rel32` to control execution flow between HEP fragments. Flexibility is provided by allowing the HEP designer to place fragments further away from each other such that they can be placed in areas of the code where they fit better with the MEP.

HEP-TO-MEP-TO-HEP

Jumping from a HEP fragment to the MEP and back into the HEP is possible if the MEP can include indirect branch instructions, e.g., `jmp eax` or `call edi`. Function pointers tend to generate these types of assembly instructions and could be used to achieve HEP-to-MEP-to-HEP control flow. The register destination of the indirect branch instruction need to be prepared before temporarily jumping into the MEP, so control flow can resume in the HEP when the MEP is finished. Normal instructions in the program can now be designed to be used in the HEP, and still allow control flow to resume at hidden instructions at the indirect branch. Below is an example of what such a construction may look like.

```
HEP: add/sub [ebp-XX],YY ;Prepare jump back to hep
MEP: ...
     ...
     mov eax,[ebp-XX]
     call eax
```

`ebp-XX` is modified to make sure it includes an address to a HEP fragment. After this instruction the control flow will continue executing MEP instruction until it is time to jump back to the HEP when the HEP fragment address is loaded in `eax` and then branched to with `call eax`.

HEP CONTROL FLOW

For branching between fragments we can use one of the following instructions as the final instruction in a HEP fragment.

```
call rel32,jmp rel32,jmp rel8,retn
```

The `call` instruction is used when we want to branch to a set of HEP fragments that end with a `retn` and provide some functionality that will be called more than once. The `call` instruction is encoded in the following way.

```
struct imm32_t *a = malloc(...);
a->imm32[T3][T4][CALL_REL32>>2] = (T5<<16)|(JMP_REL8<<8)|T1
```

and will result with the following instructions

```
8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
c7 80 e8 a8 24 00 00    mov    DWORD PTR [eax+0x24a8e8],0x40eb00
eb 40 00
```

Executing from the third byte of second instruction, the `call` instruction will be executed with the an instruction pointer offset off `0x24a8`, i.e., `T1-T4`. Following this instruction is a `jmp rel8` with an instruction pointer offset off `0x40`, i.e., `T5`. The `jmp rel8` instruction could alternatively be shifted 24 bits instead of 8, placing itself as the last byte of the instruction and use the first byte of the following instruction as its target byte, thus eliminating the need to encode `T5` in the `immediate` field. This `jmp` instruction is needed to proceed to the next HEP fragment.

The target of the `call` instruction will eventually return with a `retn` instruction and thus we need to provide another branch instruction after the `call`. In this example we use a `jmp rel8` instruction to branch to the next HEP fragment. `T5` specifies the offset from the instruction pointer which should point

to the next HEP fragment. It should be a power of two to keep the weight of the `immediate` value as low as possible.

Note that `call rel32` has opcode `0xe8` which is evenly divisible by four and thus can be specified in the struct member 'imm32's first index position.

9.3 EVADING ANALYSIS

In this section we will discuss some approaches to evading analysis of the hidden code. For this to be achieved it is important that the MEP does not appear out of the ordinary. Design decisions in the source code is discussed, as well as how various instruction fields should be crafted to be properly mapped from MEP to HEP in order to optimize analysis evasion.

9.3.1 IN THE SOURCE CODE

The previous section provided a straightforward and easy to implement way of constructing the HEP from source code. However, this provides very little stealth for the hidden code. The most obvious changes would be to rename the defined opcode macros to something that fits the main functionality of the application. This is not ideal either, because we have code that accesses a seemingly arbitrary place in the matrix. The solution here is to modify the struct to have some single variables of it at a specific offset and not rely on multi-dimensional matrices to achieve the desired `displacement` layout. For example, if we want a HEP fragment of `pop eax, push edi, retn` we would redesign the struct used in the following way.

```
struct imm8_t {
    char a[0xc35758];
    char status;
};
```

With this struct, to generate the desired HEP instructions, we would not need to access some seemingly random element in a huge matrix. Instead we would access the struct member 'status'. Accessing this member would make the compiler generate an instruction with the desired `displacement` field. The filler struct member 'a' could be divided into several variables and arrays of different sizes, as long as the total amount of memory they occupy is the same as the array above. This is necessary in order to make the 'status' struct member generate the desired `displacement` field. The struct is designed to fit into the application in general.

Multiple different structs may have to be designed this way to accommodate for the entire HEP. Still, it is possible to use the same struct for multiple

fragments. Imagine we want the same HEP fragment as above, but with the only difference of having `pop edx` instead of `pop eax`. We would simply add another char member to the struct after 'status' and reference it to set the `displacement` field to include the new HEP instructions

```
59          pop edx
57          push edi
c3          retn
```

9.3.2 IN THE MEP

Improved stealthiness can also be applied to the `immediate` field. Large value constants are not so common in compiled code, so we aim at using either low-weight `immediates`, so they can be used as flag-fields or, in the case of 4-byte `immediates`, have the three most significant bytes set to zero and the least significant byte to anything. Focusing on the former where we want to create low-weight `immediate` values that can be used to create HEP instructions we must make sure that whatever HEP instruction fields are mapped to the MEPs `immediate` field must be of low-weight. Below we explore some options and best practices to achieve this.

9.3.3 IN THE HEP

HEP instruction fields can be designed to accommodate reasonable MEP instructions. In this section we only consider HEP instruction fields encoded to MEP `immediate` fields.

OPCODES

Finding opcodes with a low weight limits us to one-byte opcodes, as a two-byte opcodes always start with 0x0F, which already has four bits set. Moreover, the most common and widely used opcodes are those of one byte. Many opcodes can have their weight lowered by disabling the direction bit, which is the second least-significant bit and is used to reverse the operands specified in the `mod-r/m` byte. Table 9.1 provide examples of some low-weight opcodes.

MOD-R/M AND SIB

The `reg`, `r/m`, `scale` and `index` bits specify the source and destination operands of the instruction. In Table 9.2 we show how the different operands are encoded and their respective weight.

Clearly the register `eax/al` is the best choice if we want to lower the weight of the `mod-r/m` byte, but we must often also work with a second operand.

Table 9.1: Examples of some opcodes and their weight.

Opcode	Mnemonic	Weight
0x00	add r/m8, r8	0
0x20	and r/m8, r8	1
0x40	inc eax	1
0x50	push eax	2
0x80	add r/m8, imm8	1

Table 9.2: Examples of some operand encodings and their weight.

operand	encoding	weight
eax(al)	000	0
ecx(cl)	001	1
edx(dl)	010	1
esp(ah)	100	1
ebx(bl)	011	2
ebp(ch)	101	2
esi(dh)	110	2
edi(bh)	111	3

`ecx/cl` and `edx/dl` are the second-best choice. Although `esp/ah` also has a weight of 1, `esp` is often not desirable to use due to it being used to keep track of the stack.

DISPLACEMENT AND IMMEDIATE

These fields are straightforward in that they need to keep a low weight to keep the MEPs `immediate` field a low weight.

9.4 A PROOF OF CONCEPT

We have implemented and released a proof-of-concept backdoor [cja] which relies on the technique described in this chapter. The backdoor opens up a listening TCP port and gives shell access to whoever connects to that port. The HEP fragments are scattered throughout multiple functions of the application and includes sequences of HEP fragments as functions that are called from other places in the HEP multiple times.

The backdoor is a modified variant of the one found in [gey]. It was modified to make the number of bytes for each instruction as small as possible in order to simplify encoding them in the `displacement` and `immediate` fields

of the MEP instructions. We also reduced the total number of instructions used by placing some sequences of instructions inside functions that could be called multiple times.

For this proof-of-concept we have not implemented any real functionality for the application. In order to make the HEP work, we have inserted a specific amount of filler bytes, namely `nop` instructions such that the branches that transfers control flow between the HEP fragments reach each other accordingly. These `nop` instructions may be replaced with other instructions, so long as the replacement instruction are of equivalent size in terms of number of bytes produced by the compiler for said instructions. Any number of functions that are needed to implement the functionality for the intended application could be written beforehand, and only after design the backdoor into the program.

Below is an example function from the proof-of-concept called `prologue`.

```
void prologue(byte3_jmprel8_t *c, imm8_t *a,
              retn_pushedi_popeax_t *f)
{
    /**************** p1 ***********///
    c->popedi_popebx = JMP_REL8;    //    POP EDI
    t1((imm8_t*)a->a,FLAG27);      //     POP EBX
    /****************************///

    FILLER84

    /**************** p2 ***********///  POP EAX
    f->popeax_pushedi_retn = 0;     //    PUSH EDI
    /*****************************///     RETN

    FILLER38
}
```

The HEP fragments in this function are called multiple times from other places in the HEP. The HEP fragment labeled `p1` includes hidden instructions for `pop edi` and `pop ebx`. The struct member `popedi_popebx` is located at an offset such that the hidden instructions are encoded in the displacement field of the assignment instruction. Naturally, the struct members variable name in the example can be changed to anything, in order to blend in with the application to provide more stealth. The third byte of the displacement field for all members of this struct is the opcode for `jmp rel8`. This is necessary because the fourth displacement byte is `0x00` we will branch past it to the next opcode that is encoded in the immediate field. The struct member is of

a char type, and the value `JMP_REL8` is assigned to it, encoding the immediate field with the opcode of the `jmp rel8` instruction that is designed to take us to the next HEP fragment with the `prologue` function, namely `p2`. The `jmp` instruction will derive how far it will jump based on the next byte in machine code. This byte is derived from the machine code generated by the `t1` function call. This function call will push the last argument of the function call to the stack and the `jmp` opcode will be paired with the value `0x68` which is the opcode of the `push` instruction. In order to hit HEP fragment `p2` we must include some instructions in between them. `FILLER84` is a macro that will insert 84 bytes of `nop` instructions. These can later be replaced with whatever code the developer chooses, as long as it is exactly 84 bytes.

HEP fragment `p2` include additional instructions and a `retn` instruction to return to the place where `prologue` was called. The immediate field is not used in this HEP fragment and can be assigned any value.

The proof-of-concept consist of two functions, `prologue` and `epilogue`, which include two HEP functions, i.e., functions that are used in other parts of the HEP. There are an additional 9 functions simply labeled from `1` to `9` which implement the entire backdoor including calls to the two HEP functions. A total of 87 HEP fragments are created and each fragment includes at least one assignment statement and sometimes a function call. The function calls were mainly used to provide an offset for a preceding `jmp` instruction. Since function calls with arguments start by pushing these arguments to the stack, the values `0x6a (push imm8)` or `0x68 (push imm32)` were used, but other statements that generate other byte values can be used, if they fit better into the application. Thus the programmer has some flexibility when designing the backdoor.

9.5 DISCUSSION

In this section we will provide an evaluation of the methodology described in this chapter based on its stealthiness and maintainability. We will also enumerate its strengths and weaknesses and come up with ways on how an attacker utilizing these methods can be detected and prevented.

9.5.1 MAINTAINABILITY

It is difficult to maintain the hidden code mainly because of the fact that if we need to change, add or remove some HEP fragments, we must adjust the jumping offset from any other fragment which crosses the fragment that was changed. For a software project maintained by multiple developers this could be problematic, as they may change code which could break the HEP. Trying to fix the code is difficult if the `immediate` and `displacement` values used

need to change and are already conformed to make sense in the applications source code.

To have some degree of maintainability of the HEP, the attacker should try to confine the entire HEP to closely located functions, which offer specialized functionality in the MEP that only the attacker is bound to change, thus maximizing the chances of other developers leaving that specific region of code alone.

Other issues that may break maintainability is if the compiler configuration changes such that the offsets between HEP fragments changes or different instructions are generated than expected by the compiler. Due to this the backdoor may have a limited lifetime expectancy.

9.5.2 STEALTHINESS

The HEP is well-hidden since it is not explicitly visible in either source code nor compiled binaries. It can be detected during dynamic binary analysis, but only if the HEP is explicitly selected for execution by whatever trigger the attacker has implemented for it. A code reviewer may react on some coding styles that were necessary for the attacker to implement the HEP, but should not discover the hidden code unless delving deeper into the compiled binaries to find the overlapping instructions that constitutes the HEP. The source code contain no semantics whatsoever of the hidden instructions.

Many binary analysis tools (e.g, BAP [BAP]) turns the assembly code into its own intermediate representation by lifting it to a higher-level representation and including explicitly the side-effects of each instruction. This would enable analysis on the intermediate language no matter what platform (ARM, x86, etc.) the binary is targeting. In these types of tools, the hidden code should be completely removed from the intermediate representation and not be detectable at all.

Naturally, the bigger the software project, in terms of lines of code, the greater the chance for remaining undetected. The attacker has a greater flexibility in designing the HEP because of a plethora of choice on where to implement it. Where to place the HEP in the source code may also be of importance and is discussed in more detail in Section 9.5.4.

9.5.3 STRENGTHS AND WEAKNESSES

The obvious strengths of this approach is that it enables hiding malicious code in deterministic builds which is not easily detectable by source and binary code analysis.

In this chapter we assume that we have a deterministic build since we can make an assumption on compiler software and its configurations, this makes the attack ineffective on non-deterministic builds since each compilation could

produce different binaries. Some other weaknesses relate to the difficulty of getting such code commit accepted to a project due to the appearance of the code. In this chapter we present one way of achieving this attack and we also present a more stealthy-looking code of the attack, shown in Section 9.3.

Some other use-cases for this technique are software obfuscation and tamper-resistant software. In the case of obfuscation, if large portions of the code are constructed using this overlapping technique with many different trigger conditions throughout the code, it will be time-consuming for manual reverse engineering efforts to identify all the hidden code. In order to obfuscate software further, bogus code can be inserted in some HEP segments in order to mislead. Regarding tamper-resistance, this relates to modifications of software during runtime since we assume that the binary is signed or part of a deterministic build verification and cannot be tampered with statically. In this case, so called guards, code that check the integrity of other code portions could be implemented in the hidden code segments throughout the application. However, for this to be feasible in terms of usability, the properties of obfuscation and tamper resistance should be easy to achieve, preferably using an automated transformation of the original source code.

9.5.4 PREVENTION AND DETECTION

One approach to break the attack is to utilize instruction randomization [PPK12b], i.e., swapping instructions for semantically identical ones and rearranging instructions independent of each other. Doing this during the loader process or runtime will keep all the benefits of deterministic builds. A binary can be instrumented with CFI [ABEL09b] which ensures that runtime control-flow transfers are valid according to the static analysis of the program, e.g., during the compilation process. For this attack, the transfers to the HEP will not be detected during static analysis and therefore prevented during runtime. Yet another prevention mechanism is to eliminate during the compilation process, all unaligned instructions that can change the program control flow, so called unaligned free-branches as described by Onarlioglu et al. [OBL+10b].

At the source code level, frequent adding of new code or refactoring existing code can render a defect backdoor when being triggered since the hidden assembly code is based on a snapshot of the code repository and it assumes memory layout is static. The attacker is best of to place the code in functions that are rarely maintained in order to preserve the backdoor functionality. However, planting backdoor code in functions that are untouched for a very long time might rise suspicion. Therefore, there is a tradeoff between visibility and persistence of the backdoor that the attacker has to balance. Adding code to functions with a medium-level of commit frequency might be the most stealthy approach but it comes with some risk that the backdoor might be

rendered defective by any future commit. Project owners that would employ frequent new code insertions, e.g., declaration of new variables in order to break the attackers assumption of the memory layout, seems to be the best strategy to prevent this type of attack at the project management level.

At the user-side, cautious users may execute untrusted programs in a NaCl [YSD+09] type of sandbox where instructions pointers and `call` and `jmp` instructions must point to a valid instruction boundary. A program with this type of backdoor will be prevented to jump into a HEP with such constraints.

9.6 CONCLUSION

In this chapter we have presented a technique that allows an attacker to craft malicious code in an applications source code and have its semantics hidden in both the source code and compiled binaries. One requirement to achieve this it that the software is compiled on a CISC architecture and in a reproducible manner. Our proof-of-concept backdoor shows that the technique is feasible and that a determined attacker could utilize it to compromise a software project. Given an insider attack, like a trusted developer in a software project, would increase the probability for a successful attack. Since this attack is exploiting the increased trust in deterministic builds, it could very well be stealthy for a long time. We also discuss how to detect and prevent this type of attack, both at the project management level and regarding defense that can be employed at the target platforms.

10

Detecting ROP Payloads in Data Streams

In this chapter we present eavesROP, which is a lightweight approach to detecting ROP attacks. We try to identify ROP payloads by looking at network traffic only, i.e., we do not make any modifications to machines, programs, libraries or operating systems; nor do we try to execute any of the received data. We do not even require any kind of access to the machines. Scenarios could be an implementation in a gateway to a corporate network, ROP payload detection in switches or at an ISP before data is forwarded to the end user. The question that we try to answer is: How much information can we deduce by just looking at the data? We target ROP exploits where gadget addresses can be explicitly found in the data sent to the application. We assume that ASLR is enforced by the operating system, and that the attacker has information about the location of libraries. Of course, our detection mechanism has no such information. We show how to filter out possible ROP payloads and how to determine if the candidate payload is a ROP attack or not. Even with just a moderate number of gadgets, we can detect the payload efficiently. This is true even if there is a large amount of noise present.

Our assumption is that the attacker can find the location of gadgets in the memory of a target computer. This can for example happen through another vulnerability introducing an information leak, which reveals some memory address to the attacker.

One example of an information leak is one part of the GHOST attack [Qua15]. In this attack, the attacker can—from a remote host—get the heap address of a configuration structure internal to the Exim mail server. This address is then used when performing the actual exploit. The address can be found even though ASLR is enabled, and the executable is a PIE. Even though the attack is not a ROP attack, it shows that it is not unreasonable to assume

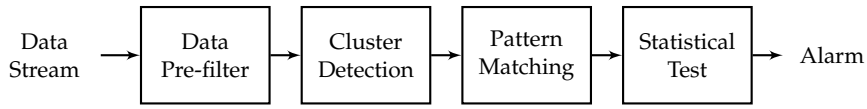

Figure 10.1: Data flow overview.

that an attacker may have knowledge of memory addresses internal to the application.

10.1 OVERVIEW OF APPROACH

In this section, we give a brief overview of eavesROP. The entire exploit payload detection mechanism can be regarded as a black box device that takes a data stream as input and raises an alarm if it finds a ROP payload. The analysis of the data is based on the property that a payload consists of several 4-byte aligned addresses to gadgets within one library or chunk of executable code. Note that not all gadgets have to be in the same library. We only require a certain number (T) to be in one known library. The rest can be located elsewhere.

The analysis performed inside the black box is divided into four layers, each with a distinct task, see Figure 10.1.

OPTIONAL DATA PRE-FILTER

The optional data pre-filter is a simple filter designed to discard all data that is unlikely to represent memory addresses.

Even though eavesROP is not very noise sensitive for detection, its performance is sensitive to data with very small variations, as this type of data could potentially represent memory addresses that are closely grouped. Plain text data has this near-recurring property and can thus for performance reasons be thrown away in a data pre-filter. A well-designed data pre-filter will significantly lower the processing requirements of the exploit payload detection mechanism since fewer blocks of data will be passed on to the next layer—the address cluster detection layer.

CLUSTER DETECTION

An actual exploit payload will contain several gadget addresses that lie close together with respect to the entire addressable memory space. The purpose of the address cluster detection is to find and isolate the congested parts of

the memory space for further processing. If several of the memory addresses in a data window of maximum exploit payload size end up close together, then this address window is worth further scrutiny. Other data will scatter randomly over the addressable memory space, since their correlation to actual gadget addresses in any given library is low.

The address cluster detection layer can be viewed as a transformation from data space to address space. Given a data window of some size, the clustering algorithm detects address windows that may contain dense memory activity. Thus, this layer also has the task of aligning data into 4-byte chunks since ROP gadgets are called using a 4-byte address, assuming a 32-bit architecture. After alignment, a second filter can be used in order to sort out certain addresses.

Based on clustered address windows, this layer will output a binary address vector, P_{obs}, of size L, i.e., the size of the largest targeted library. These vectors indicate addresses found in the data.

PATTERN MATCHING

The vector P_{obs} from the previous step is matched with binary library vectors. The relative distances of the memory addresses in an address window form a very distinct pattern. This pattern is matched with the gadget address patterns of libraries P_{lib}.

Without ASLR, this pattern matching could be trivially achieved by simple constant-time lookups into a suitable hash table (for example, see [PR04]). However, when we allow ASLR, the precise memory location of the library is unknown, making pattern matching more complex. Still, using limitations of ASLR together with the relatively few addresses that passes the cluster detection step allows for very efficient pattern matching between P_{obs} and P_{lib}.

A data window containing only actual gadget addresses will give us a perfect matching—all addresses will be matched to the library pattern. A data window containing random or non-gadget related addresses will not match very well. The output of the pattern matching layer is a quantification of the maximum matching of an address window with respect to a library.

STATISTICAL TEST

An address window containing only gadget addresses will give a perfect match. However, since we do not know the size of a payload, an address window may contain addresses that are just noise. The goal of the statistical test is to minimize false positives (α) and false negatives (β) by using a threshold value for the maximum overlap between P_{obs} and P_{lib}. This threshold will depend on the Hamming weight of P_{obs} and the targeted values of α and β.

The maximum matching measure is the core ingredient of a distinguisher—a decision mechanism. Statistical tests are used to decide whether a given maximum matching corresponds to an actual gadget address pattern or not. The behavior of random vs. non-random addresses plays an important role here, as does the number of addresses in the address window, the address window weight.

10.2 A MORE DETAILED DESCRIPTION

In this section we give a more detailed overview of the different parts of eavesROP.

10.2.1 OPTIONAL DATA PRE-FILTER

Certain input data can be expected to exhibit properties that make them look like addresses close to each other in the memory space—thus looking like ROP payloads—even though the data is actually non-malicious. Our goal is to filter out these addresses before they reach later steps in the algorithm, to reduce the total computational overhead of our system.

Of special interest are printable ASCII characters, not only because much data is readable text, but also because large portions of adjacent ASCII data may—when combined into 32-bit words—look like adjacent addresses. Filtering is however a trade-off between performance and false negatives. There are techniques to make ROP payloads printable [LZWG11]. Such a payload would be removed if a filter for printable characters is enabled. This is why the filtering step should be considered optional.

If the pre-filter is enabled, it removes blocks of UTF-8 strings. In our implementation, we define a block as a sequence of five or more adjacent, printable UTF-8 characters. A printable UTF-8 character is defined as all ASCII characters in the range from `0x20` up to and including `0x7e`. We also include the `0x09`, `0x0a` and `0x0d` for tab separator, line feed, and carriage return, respectively. Even though the last three are not printable characters, they occur frequently in the same context as the printable characters and whitespace. All valid multi-byte encoded UTF-8 characters are also considered printable.

When a matching block is found, the complete block is removed from the input. This leads to potential noise as non-adjacent bytes become adjacent after the data between them is removed. This does, however, only affect a few addresses, which does not cause any problems in practice since our ROP pattern matching is very precise and noise tolerant.

The filter has been designed by hand, by iteratively trying different heuristics and looking at the addresses returned by the Address Cluster Detection Algorithm described later. Initially, the filter only looked at ASCII charac-

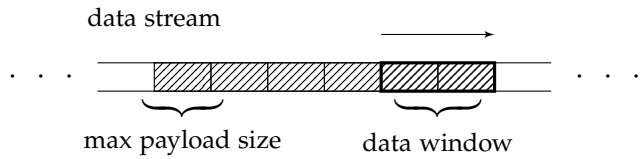

Figure 10.2: Data window progression.

ters in the range `0x20` to `0x7e`, but by looking at the matched addresses we could see that significant amounts of the matched addresses either contained whitespace characters, or UTF-8 characters. By applying a new set of rules the number of false matches decreased, and the performance of the Address Cluster Detection Algorithm was increased.

As our approach is modular, it is possible to add other heuristic pre-filters. Possible ideas may be to filter out specific network packets, such as ARP-requests. Even when using a very aggressive filter, if one or a few gadget addresses are filtered out, the detection will still succeed as long as there are enough gadgets left in the data that passes the filter.

10.2.2 CLUSTER DETECTION

We let M denote the maximum size of a ROP payload in 4-byte words that our detection is guaranteed to support. A naïve approach to detect the gadget addresses is to pick M words of data, map them to P_{obs} and match this vector with a known P_{lib}. Doing this byte by byte in the data would produce the correct maximum matching, but it is a very slow approach. Moreover, all words but one will repeat every 4 bytes. Another problem is that the addresses contained within the data window can be spread out over the entire ASLR address space (N bytes), making P_{obs} very large. We propose to use an algorithm that is much more efficient, and will still always find the correct maximum matching.

Instead of considering M addresses, we pick a data window of size $D = 2M$. Thus, we consider twice as much data as the maximum payload, but in return we consider $M + 1$ possible payloads simultaneously. This is illustrated in Figure 10.2. Doubling the data window size introduces a little more noise (more data in one window), so a few more data windows will pass the cluster detection stage, but this effect is marginal compared to the significant gain in processing efficiency.

When the data window slides over the next data chunk of size M, we begin by extracting potentially viable addresses. As the offset of a ROP payload in the data buffer is unknown, but we know that each address is four bytes and

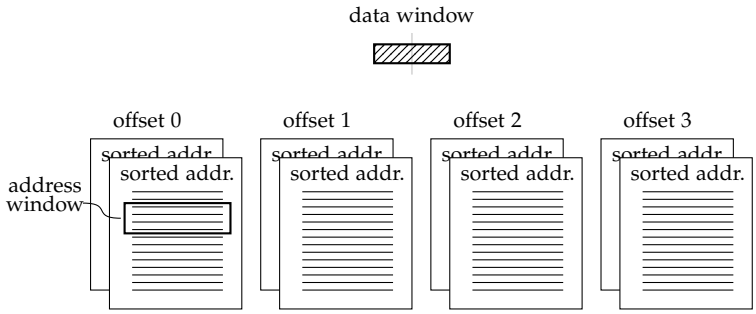

Figure 10.3: Memory address pattern extraction from data window.

addresses are aligned inside a payload, we create four lists, one for each offset, see Figure 10.3. It is possible to store all such addresses in one combined list, but the accuracy of the pattern matching is maximized by separating and grouping the addresses as prescribed.

As Figure 10.3 further illustrates, we need to keep track of eight such address lists, four for each of the two M-word data chunks covered by the D-word data window. Separating the lists per data chunk allows for incrementing the data window in steps of M words, while reusing the four lists corresponding to the previous M words.

The four new address lists are sorted using an efficient linear-time sorting algorithm such as bucket sort [CLRS09]. Such efficient sorting is possible since all addresses are of the same size. Once sorted, we slide an address window of size L (same size as executable part of the largest library, currently only the .text section of the executable) down the combination of the two lists for each offset. Since each of the two lists is individually sorted, it is trivial to traverse the combination in sorted order efficiently. The time complexity for this is linear in the number of stored addresses, i.e., linear in D.

As mentioned before, the point of separating the addresses on offset is to make the pattern matching stage more accurate. If the lists were combined, we would still be able to match the pattern of the ROP addresses to the corresponding library pattern. However, we achieve a higher rate of false positives as the overlaid noise addresses are collectively more likely to trigger false alarms.

Let T be a threshold value that determines the minimum number of gadgets in an exploit that we want to be able to detect. A small T leads to detection of more exploits, but it also results in more pattern matching, slowing down the detection algorithm. In practice, the lowest value that our algorithm can handle is $T \approx 7$, depending on the instruction size of each gadget (see Sec-

Algorithm 1 – Address Pattern From Data Stream

Input: data stream, maximum payload size in words M, address window weight threshold T, size of library L, word size in bytes n.
Output: set of P_{obs} (address window patterns).

```
pos = 0; /* current byte position in data stream */
A_(0,0) = ... = A_(0,n-1) = ∅; /* two address lists per offset */
A_(1,0) = ... = A_(1,n-1) = ∅;
while (data stream not exhausted) {
   for (each byte offset i ∈ {0,...,n-1}) {
      A_(1,i) = M words from data stream starting at pos + i;
      sort A_(1,i);
   }
   pos = pos + nM;
   for (each offset i ∈ {0,...,n-1}) {
      slide address window of size L over A_(0,i) ∪ A_(1,i) and
      find clusters;
      A_(0,i) = A_(1,i);
   }
}
```

tion 10.2.3) and the error probabilities (see Section 10.2.4).

If we find an address window that contains at least T unique addresses, the binary vector P_{obs} is constructed by entering a '1' in each position corresponding to an address in the address window. Then we proceed to perform pattern matching. Algorithm 1 summarizes the cluster detection procedure.

10.2.3 PATTERN MATCHING

In this section we give more details on the pattern matching layer. In order to pattern match the P_{obs} vector we first need to construct a vector P_{lib} of gadgets.

IDENTIFYING GADGETS IN A LIBRARY

In order to find all possible gadgets in a library, the executable part of it is scanned for the opcode of different types of return instructions, namely `0xC2` (`retn imm16`), `0xC3` (`retn`), `0xCA` (`retf imm16`) and `0xCB` (`retf`). For each position of these bytes in the library we search backwards one byte at a time and try to assemble a legal instruction flow ending with the return. We define the *entry zone*, z, as the number of instructions we allow for each gadget, not including the return instruction. This means that we can find many gadgets

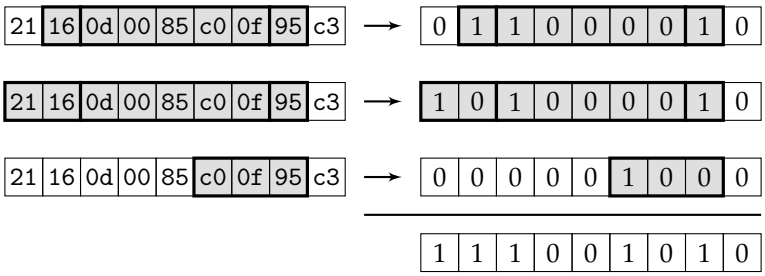

Figure 10.4: Translation of maximal length gadget sequences to binary pattern.

ending at the same return instruction due to the possibility of instruction overlapping in the x86 architecture.

The starting byte of every possible gadget is used to construct the binary vector P_{lib}. This is the vector that is used for pattern matching with P_{obs}, which is the output of the address cluster detection algorithm.

To understand how the gadget structure in a library is translated into a binary pattern, consider the following sequence of nine bytes (hex):

```
21 16 0d 00 85 c0 0f 95 c3
```

Using an entry zone of size $z = 3$ (at most three instructions), we construct maximal gadget chains by interpreting the bytes preceding the return instruction `c3` as consecutive instructions. There are three possible maximal gadget chains in the above byte sequence, as illustrated in Figure 10.4.

The top two gadget chains are both of length three. While the top chain begins with a single-byte instruction `16`, the second chain extends this to a two-byte instruction `21 16`. The third chain is of length 1, but it is maximal since it cannot be further extended.

A sequence of bytes belonging to a library is translated into a binary pattern according to the following rules. Every byte position is designated a '0' or a '1'. A '1' is assigned if any maximal gadget chain has an instruction that begins at that byte position. If not, a '0' is assigned. Note that any unique

Table 10.1: The number of gadgets in libc for some choices of entry zone.

entry zone (z)	1	3	5	7
Number of gadgets (G)	13175	36898	58151	77830

Algorithm 2 – Gadget Pattern From Library

Input: entry zone z, library file f.
Output: library gadget pattern P_{lib}.

```
P_lib = (0,...,0); /* same length as f */
for (every byte position i in f) {
   if (byte i in f is not a return opcode) continue;
   /* disassembly */
   G = set of maximal gadget chains of length ⩽ z ending at byte i;
   for (every maximal gadget chain g in G) {
      for (every instruction j in g) {
         k = location of first byte position in instruction j in f;
         if (k&0xFF != 0x00) /* NUL bytes break the payload */
            P_lib[k] = 1;
      }
   }
}
return P_lib;
```

sequence of valid instructions leading to a return counts as a gadget. Counting the instruction subsets of the top two chains, there are five gadgets in Figure 10.4. Algorithm 2 summarizes the construction of a P_{lib} vector.

As an example, Table 10.1 shows the number of gadgets, G, found in libc given the size of the entry zone. It should be noted that the gadget identification can be greatly optimized by filtering out useless gadgets. The only optimization applied right now is the removal of gadgets with a least significant byte of null, since exploit payloads generally won't work if they have null-bytes in them.

PATTERN MATCHING OF P_{LIB} AND P_{OBS}

Pattern matching, here, means that we want to find the maximum weight of the overlap between two patterns that are overlaid, possibly displacing the patterns linearly with respect to one another. We also want this matching to be perfect, which is to say that all actual gadget addresses that are used in an exploit will be counted. All actual gadget addresses in an exploit will contribute positively to the weight of the maximal pattern match.

Recall that L denotes the maximum size of the executable part of the libraries. Focusing on one such library, P_{lib} is a binary vector of length L. If the ith byte of the executable part of the library can be interpreted as a gadget address, set $P_{\text{lib}}[i] = 1$, otherwise set $P_{\text{lib}}[i] = 0$. The vector P_{lib} now represents

the gadget pattern of the library. Correspondingly, P_{obs} is a binary vector of length L from the address clustering detection step. From an address window of length L containing at least T addresses, set $P_{\text{obs}}[i] = 1$ if the ith address appears in the address window, and set $P_{\text{obs}}[i] = 0$ otherwise. The vector P_{obs} now represents the address pattern of the observed data.

If both patterns are aligned, the maximum matching can be calculated as the dot product between P_{lib} and P_{obs} according to

$$P_{\text{lib}} \cdot P_{\text{obs}} = \sum_{i=0}^{L-1} P_{\text{lib}}[i] P_{\text{obs}}[i].$$

However, we have no way of knowing if the alignment is correct, so we rather need to try all alignments to see which one produces the highest fit. That is, we need to calculate the dot products for all possible shifts of the two patterns. In the general case, such pattern matching can be performed efficiently using a Fast Fourier Transform (FFT). The FFT computes the circular discrete convolution c of two vectors a and b of length L,

$$c\,[t] = (a * b_L)\,[t] = \sum_{i=0}^{L-1} a\,[i]\, b\,[(t-i) \mod L]\,. \tag{10.1}$$

The FFT approach (see [Bra99]) has time complexity $O(L \lg L)$, compared to $O(L^2)$ for the naïve approach. Letting $\mathcal{F}$ denote the FFT version of the Discrete Fourier Transform (DFT), we may compute c as

$$c = \mathcal{F}^{-1}\left(\mathcal{F}\,(a) \odot \mathcal{F}\,(b)\right),$$

where $\odot$ denotes componentwise multiplication.

We let a and b be the vectors P_{lib} and P_{obs} respectively after the zero padding as described above. The weight of the maximum matching is given as the maximum component of c,

$$c_{\max} = \max_i c\,[i]\,. \tag{10.2}$$

Note that $\mathcal{F}\,(P_{\text{lib}})$ can be precomputed since libraries are known in advance.

Even though FFT is efficient in the general case, the naïve approach is actually (almost always) faster in our particular application. This is due to two distinguishing properties.

1. The vector P_{obs} is typically of very low weight as seen in the simulation results in Table 10.4.

2. Libraries are aligned at memory pages in memory.

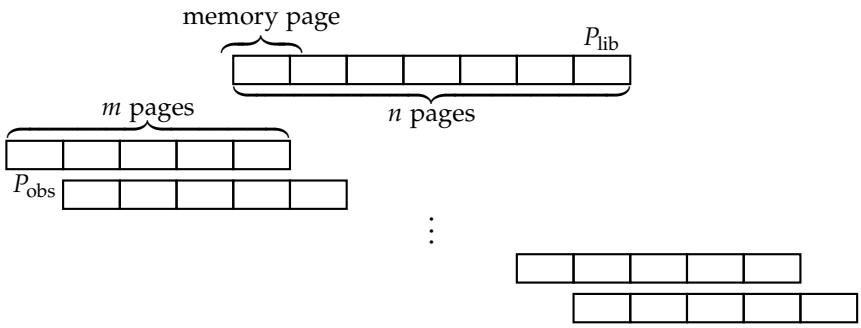

Figure 10.5: Optimized algorithm. P_{obs} slides one page at a time along P_{lib} and calculates at what alignment the maximum match occurs. It then returns that $c_{\max}$. $m \leqslant n$

The $O(L^2)$ complexity of the naïve approach is based on picking an entry in the first vector and then count the number of overlaps when iterating over both vectors. This is done for all entries in the first vector. Since we want to count overlaps of '1's, we would instead do this for each '1' in the first vector. Thus, the complexity is actually $O(wL)$, where w is the weight of P_{obs}. Furthermore, the size of a page in memory is 4kB for most operating systems. This means that we do not have to compute the overlap for all offset, but only a fraction 2^{-12} since we known some address bits. This will give as a resulting complexity that is $w \cdot L \cdot 2^{-12}$. which is significantly smaller than $O(L \lg L)$ for all practically possible values of w. The mechanics of the algorithm is illustrated in Figure 10.5.

10.2.4 STATISTICAL TEST

In order to find an expression for the maximum number of overlaps in the pattern matching step, we make the following approximations.

- Locations corresponding to gadgets in P_{lib} are uniformly distributed.
- The number of overlaps found for different offsets are approximated as independent events, all with the same probability.

Using these approximations, the number of overlaps between P_{lib} and P_{obs} is binomially distributed, $X^{(w)} \sim \text{Bin}(w, \frac{G}{L})$, where w and G denote the Hamming weights of P_{obs} and P_{lib}, respectively. Recall that G should here be understood as the number of gadgets in a library for a given entry zone and w is the number of addresses in an address window. Thus, the probability

that there are s overlaps is given by

$$\Pr(X^{(w)} = s) = \binom{w}{s} \left(\frac{G}{L}\right)^{s} \left(1 - \frac{G}{L}\right)^{w-s}, \tag{10.3}$$

with expected value and variance given by

$$\begin{aligned} E(X^{(w)}) &= \frac{wG}{L}, \\ V(X^{(w)}) &= \frac{wG}{L}(1 - \frac{G}{L}). \end{aligned}$$

With P_{lib} of size L we have $L/4096$ such binomially distributed samples. In order to find the probability distribution for the maximum value of the convolution array, we write the probability that any single value is at most s as

$$\Pr(X^{(w)} \leqslant s) = \sum_{t=0}^{s} \binom{w}{t} \left(\frac{G}{L}\right)^{t} \left(1 - \frac{G}{L}\right)^{w-t}.$$

The probability that all values are at most s is then, using the second approximation above, $\Pr(X^{(w)} \leqslant s)^{L/4096}$. From this it follows that the probability distribution for the maximum value in the pattern matching $c_{\max}^{(w)}$ is given by

$$f_{c_{\max}^{(w)}}(s) = \Pr(c_{\max}^{(w)} = s) = \Pr(X^{(w)} \leqslant s)^{L/4096} - \Pr(X^{(w)} \leqslant s-1)^{L/4096} \tag{10.4}$$

with cumulative distribution function

$$F_{c_{\max}^{(w)}}(s) = \Pr(c_{\max}^{(w)} \leqslant s) = \Pr(X^{(w)} \leqslant s)^{L/4096}.$$

and expected value and variance given by

$$E(\mathrm{C}_{\max}^{(a)}) = \sum_{i=0}^{a} i \left(\Pr(X^{(a)} \leqslant i)^{L/4096} - \Pr(X^{(a)} \leqslant i-1)^{L/4096})\right) \tag{10.5}$$

$$V(\mathrm{C}_{\max}^{(a)}) = \sum_{i=0}^{a} i^2 \left(\Pr(X^{(a)} \leqslant i)^{L/4096} - \Pr(X^{(a)} \leqslant i-1)^{L/4096})\right) - E(\mathrm{C}_{\max}^{(a)})^2. \tag{10.6}$$

Table 10.2 shows the simulated and theoretical distributions for $c_{\max}^{(w)}$ when using random data. The theoretical distribution is given by (10.4) and the simulations are performed by taking 10,000 arrays (P_{obs}) with Hamming weight w, uniformly distributed over the array. Using the optimized algorithm for

Table 10.2: Histogram comparison of maximum overlap ($c_{\max}^{(w)}$) for various address window weights over random data and libc. Entry zone size is 3 instructions, sample size 10,000.

weight w												
10	overlap	0	1	2	3	4	5	6	7	8	9	10
	simulation	0	12	6479	3315	189	5	0	0	0	0	0
	theory	0	0	4543	5053	389	14	0	0	0	0	0
20	overlap	0	1	2	3	4	5	6	7	8	9	10
	simulation	0	0	324	6747	2653	260	16	0	0	0	0
	theory	0	0	23	4693	4595	636	50	3	0	0	0
30	overlap	0	1	2	3	4	5	6	7	8	9	10
	simulation	0	0	0	1884	6207	1701	192	15	1	0	0
	theory	0	0	0	340	5640	3404	551	59	5	0	0
40	overlap	0	1	2	3	4	5	6	7	8	9	10
	simulation	0	0	0	115	4492	4310	936	127	19	1	0
	theory	0	0	0	1	1547	5718	2284	393	50	5	0
50	overlap	0	1	2	3	4	5	6	7	8	9	10
	simulation	0	0	0	0	1231	5484	2659	520	94	12	0
	theory	0	0	0	0	87	3481	4697	1436	257	37	5

pattern matching, the maximum overlap between the array and an array corresponding to libc with entry zone 3 is computed. A histogram for the 10,000 samples is used to illustrate the probability distribution.

A threshold value for $c_{\max}^{(w)}$ is chosen, denoted $\hat{c}_{\max}$. If $c_{\max}^{(w)} \geqslant \hat{c}_{\max}$ the payload is considered a ROP. Associated with this decision are false positives and false negatives. The false positive rate, denoted α, is defined as the probability that non-malicious data is considered malicious (i.e., a ROP payload) while the false negative rate, denoted β, is the probability that a malicious payload is mistaken for non-malicious data. To write expressions for α and β, let the Hamming weight w of P_{obs} be written as $w = w_G + w_N$, where w_G is the number of ROP gadgets and w_N is the number of noise addresses. The distribution of $c_{\max}^{(w)}$ for non-malicious data is given by Eq. (10.4). The value of $c_{\max}^{(w)}$ for a ROP payload is given by

$$c_{\max}^{(w)} = w_G + X^{(w_N)},$$

where $X^{(w_N)}$ is distributed according to Eq. (10.3). Now, we can write the two error probabilities as

$$\begin{aligned}\alpha &= \Pr(c_{\max}^{(w)} \geqslant \hat{c}_{\max}) = 1 - \Pr(c_{\max}^{(w)} \leqslant \hat{c}_{\max} - 1) \\ &= 1 - \Pr(X^{(w)} \leqslant \hat{c}_{\max} - 1)^{L/4096} \\ \beta &= \Pr(X^{(w_N)} < \hat{c}_{\max} - w_G) = \Pr(X^{(w_N)} \leqslant \hat{c}_{\max} - w_G - 1)\end{aligned} \tag{10.7}$$

The false positives rate α is only for one library. If we want to test the payload against a set of ℓ libraries, the total false positive rate α_ℓ is given by

$$\alpha_\ell = 1 - (1 - \alpha)^\ell.$$

By choosing $\alpha = 0.0001$ we allow $\ell = 100$ libraries to be supported, still keeping the total false positive rate α_ℓ below 0.01. (We assume here that all libraries are of approximately equal size.) The false negative rate can only be $\beta > 0$ if we relax the number of gadgets needed (w_G), assuming noise will contribute to the number of overlaps. Computing $\hat{c}_{\max}$ and w_G for different values of w shows that this is only useful for very large values of w. As an example, using $\beta = 0.01$, for $z = 7$ we are able to relax the requirement on w_G only when $w \geqslant 92$ ($\hat{c}_{\max} = 21$ and $w_G = 20$). For smaller z we need even larger w. If we allow large β, the requirement can be relaxed for smaller w but such a large false negative is typically undesirable. Note that β is not affected by multiple libraries since the payload will only match one library. Thus, as a rule of thumb, the number of gadgets required for successful detection can be assumed equal to the threshold

$$w_G \approx \hat{c}_{\max}.$$

To see the number of gadgets W_G needed for successful detection, we compute this value for some different values of α and w in Table 10.3. For all given values $\beta = 0$.

The standard deviation of Eq. (10.4) turns out to be very small, with almost all probability mass concentrated to only a few values for s. This makes the detection algorithm efficient, allowing us to choose small error rates while still requiring few gadgets to succeed, even in the presence of a large amount of noise.

The false positive rate has been simulated using the data from Table 10.6 and Table 10.7. The simulations show that there may be cases where $c_{\max}$ is too high, such that it would be unreasonable to select a $\hat{c}_{\max}$ only to remove these instances of false positives as the false negative rate could drastically increase. This is especially true if we look at the exploit analysis in section 10.3.1 where most of the exploits would go undetected if too high a value for $\hat{c}_{\max}$

Table 10.3: Minimum number w_G of gadgets needed for ROP payload detection in an address window of weight w. Note that $\beta = 0$ for all cases.

entry zone	entity	values								
1	w	6	10	15	20	25	30	50	100	200
	$\hat{c}_{\max}$	6	7	7	8	9	9	11	13	17
	w_G	6	7	7	8	9	9	11	13	17
3	w	7	10	15	20	25	30	50	100	200
	$\hat{c}_{\max}$	7	8	9	10	11	12	15	20	27
	w_G	7	8	9	10	11	12	15	20	26
5	w	8	10	15	20	25	30	50	100	200
	$\hat{c}_{\max}$	8	9	11	12	13	14	17	24	35
	w_G	8	9	11	12	13	14	17	24	33
7	w	9	10	15	20	25	30	50	100	200
	$\hat{c}_{\max}$	9	10	11	13	14	15	19	27	40
	w_G	9	10	11	13	14	15	19	26	36

Table 10.4: Average and maximum weight, w, for different types of data. $ez = 3$, $D = 200$ and $w \geqslant 6$. 10000 windows examined.

type of data	average w	max w
random	6.01	7
web	10.35	37
mp3	7.09	15
pdf	6.60	55
mkv	9.49	33

would be set. The number of windows that receive a high $c_{\max}$ is low, it could be argued that those windows trigger an acceptable amount of false positives. In an IDS setting where this would only trigger an alert, and no active response, this could be acceptable.

10.3 PERFORMANCE

The performance of eavesROP depends on the parameters used in the various stages of the system. All simulations have been performed on an Intel Core i7 4770 @ 3.4 GHz with 16 GB of RAM.

A more aggressive filtering in each step will reduce the amount of data sent to the next stage, which will increase the overall performance. This is illustrated in Table 10.5, where the throughput and input/output size ratio is given for various types of input data when passed through the data pre-filter.

Table 10.5: Performance of data pre-filter.

type of data	throughput (MiB/s)	input/output size ratio
random	34.7	0.965
web (HTML, JPG,...)	52.7	0.068
mp3	39.5	0.956
pdf	38.6	0.811
mkv (H.264/MPEG-4)	34.5	0.965

After the optional data pre-filter—which may have reduced the total amount of data—the data is passed to the cluster detection step. This step has a throughput of around 10 MiB/s. The output of the cluster detection step is multiple matched windows, i.e. multiple P_{obs}. Table 10.6 shows how many

P_{obs} vectors that are passed to the pattern matching layer, for some different types of data and different choices for T and D.

Table 10.6: Number of matching address windows per GiB of input data, for a data window of size D, and with at least T addresses within distance $L = 1231620$, for different types of data. L is here the size of libc 2.20.

	$D = 50$			$D = 200$			$D = 1000$		
type of data	$T = 6$	8	10	$T = 6$	8	10	$T = 6$	8	10
random	0	0	0	12	0	0	24749	53	0
web (HTML, JPG,...)	1795	689	590	5589	1878	1208	40795	8007	3292
mp3	42	8	2	631	106	10	162014	8472	1023
pdf	4068	248	61	34718	5266	1316	1011850	176992	45289
mkv (H.264/MPEG-4)	354	2	0	513	81	66	35545	841	125

Each P_{obs} outputted from the cluster detection stage will be passed to the pattern matching step.

All parts of eavesROP have been implemented and tested using real-world exploits. Those analyses are presented in subsection 10.3.1

10.3.1 DETECTING EXPLOITS

In this section we will analyze how well our technique detects real-world exploits. A total of six exploits was tested with varied results. In order to minimize false positives we set the threshold value to 7. Some exploits utilize gadgets from multiple libraries. For these tests we only check against the library containing the most gadgets. Table 10.8 summarizes the results.

Two out of six exploits could not be detected due to a lack of gadgets, four and six, clearly under our set threshold value. In the case of the Easy File

Table 10.7: Maximum $c_{\max}$ for different file types. $ez = 3$ and $w \geqslant 6$.

type of data(size)	$D = 50$	$D = 200$	$D = 1000$
random(3GiB)	< 6	< 6	< 6
web(150MiB)	8	10	19
mp3(487MiB)	< 6	< 6	7
pdf(2.9GiB)	14	18	18
mkv(586MiB)	6	11	22

Table 10.8: Exploits to vulnerable applications tested by eavesROP.

Vulnerable application	#gadgets	#libraries	detected?
BlazeDVD Pro 7.0 [Bar14]	14	1	true
DVD X Player 5.5.0 Pro [sic11]	11	1	true
Easy File Management Web Server v5.3 [Ahr14]	4	1	false
RM Downloader 3.1.3 [Nod10a]	18	1	true
The KMPlayer 3.0.0.1440 [xpl11]	21	1	true
Winamp 5.572 [Nod10b]	19	7	false

Management Web Server exploit there are no gadgets that need to handle DEP and thus the exploit requires less gadgets to achieve its goal. If the application was properly secured with DEP, more gadgets would have been necessary in order for it to work and would most likely have been detected with eavesROP.

The Winamp exploit had gadgets from multiple libraries which hindered our technique from detecting it. Our technique could however be extended to search multiple libraries simultaneously and in that case it would have been detected. The library with most unique gadgets used by the exploit only had six gadgets in it.

10.4 STRENGTHS AND LIMITATIONS

In this section we summarize and highlight the different strengths and limitations in our ROP payload detection approach.

10.4.1 STRENGTHS

An important feature of the detection mechanism is that it works even when ASLR is enabled on the targeted systems, assuming the attacker manages to bypass ASLR through e.g., information leakage or brute force.

While we have described the address cluster detection algorithm for 32-bit systems, it can also be applied to 64-bit systems. In this case brute-force attacks are out of scope due to the large entropy of ASLR, but absolute addresses could still be found using information leakage. The main modification is that the address cluster detection must consider eight offsets instead of four, but in return there will be much fewer addresses passing the cluster detection due to the large address space.

A distinguishing feature is the ability to detect ASLR brute-force attempts. As our detection mechanism only considers differences between gadget addresses, and not the addresses themselves, a probabilistic attack attempt will

be detected as a ROP payload.

In the optional data pre-filter, we only apply one very simple UTF8-filter. However, filtering options are abundant, and it is easy to imagine other ad hoc filters which will significantly reduce the computational overhead of the detection.

While most examples have focused on one library, it is very cheap to add support for a large set of libraries. It may also be noted that pattern matching over a set of libraries is inherently parallelizable and can also take advantage of dedicated hardware for computing the FFT.

Last but not least, the modular structure of eavesROP provides a flexible framework that is easily adaptable to the characteristics of the target network. This makes it possible to tailor the system to match the desired detection and performance requirements.

10.4.2 LIMITATIONS

Since we do not have access to the target machine, but only consider a stream of bytes in our search for a ROP payload, the detection mechanism has some limitations.

First, we need to know the libraries and binaries that can be used in an attack. In general, this could be any library or binary, but by choosing the most commonly used ones, it is still possible to detect a significant fraction of attacks. The FFT can be precomputed for each library and the online time for each library will be limited to a componentwise multiplication of two vectors and an inverse FFT computation.

The limited size D of the data window makes it possible to utilize the `ret imm16` instruction which pops `imm16` bytes from the stack. Using these returns with a large `imm16` would leave very sparsely located gadget addresses in the data window. This could potentially avoid detection. However, it should be noted that we only require T gadgets in the window of max payload size so the detection does in general not require all gadgets to succeed.

Since we do not execute any potential exploits, we will not be able to detect exploits that obfuscate the gadget addresses such that they are not visible in the data sent on the network. Such obfuscation would include e.g., polymorphic ROP attacks [LZWG11], or gadget addresses generated on the client-side using JavaScript or ActionScript.

We also note that it is sometimes possible to construct ROP payloads that will go undetected by not using gadget addresses in the payload. Instead, one gadget can be used to obfuscate the addresses to other gadgets. Consider the following gadget, or a variation of it, which could be found in a target library:

```
pop ESI
pop EBX
```

```
xor EBX,ESI
push EBX
ret 0
```

This gadget puts the next gadget's obfuscated address in EBX and the key for de-obfuscating it in ESI. The xor instruction de-obfuscates the next gadget address and pushes it back on the stack. The return will direct execution flow to the de-obfuscated gadget address. Thus, in this example only one gadget address would be needed.

A related limitation is when an attacker can craft an exploit small enough to avoid detection. By small enough we mean not enough gadgets to stand out from the noise.

As the FFT is rather computationally intensive, it would also be possible to mount a denial of service attack by sending data that would be interpreted as adjacent addresses, triggering false positives.

10.5 CONCLUSIONS

We have investigated to which extent it is possible to detect a ROP payload by only analyzing data, and assuming that ASLR is used on the target system. If we have the set of libraries and binaries that can be used to find gadgets, we show that it is possible to detect a ROP payload even in the presence of noise and by applying suitable data filters. The exact performance will depend on the type of data and the number of gadgets that are required for an exploit to be detected depends on the maximum allowed size for the payload and the amount of noise.

References

[AB14] D. Andriesse and H. Bos, »Instruction-level steganography for covert trigger-based malware,« in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2014, pp. 41–50.

[ABEL05] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, »Control-flow integrity,« in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. ACM, 2005, pp. 340–353.

[ABEL09a] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, »Control-flow integrity principles, implementations, and applications,« *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, 2009.

[ABEL09b] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, »Control-flow integrity principles, implementations, and applications,« *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[ACdC09] J. Aycock, J. M. G. Cardenas, and D. M. N. de Castro, »Code obfuscation using pseudo-random number generators,« *2012 IEEE 15th International Conference on Computational Science and Engineering*, vol. 3, pp. 418–423, 2009.

[Ada15] A. Adams. (2015) Research insights: Exploitation advancements. [Online]. Available: https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/2015/10/research-insights_vol-7-exploitation-advancementspdf/

[AdJ06] J. Aycock, R. deGraaf, AND J. Jacobson, Michael, »Anti-disassembly using cryptographic hash functions,« *Journal in Computer Virology*, vol. 2, no. 1, pp. 79–85, 2006. [Online]. Available: http://dx.doi.org/10.1007/s11416-006-0011-3

[AEH75] E. A. Akkoyunlu, K. Ekanadham, AND R. Huber, »Some constraints and tradeoffs in the design of network communications,« in *ACM SIGOPS Operating Systems Review*, vol. 9, no. 5. ACM, 1975, pp. 67–74.

[AGJS13] I. Anati, S. Gueron, S. Johnson, AND V. 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.

[Ahr14] J. Ahrens. (2014) Easy file management web server 5.3 - userid remote buffer overflow (rop). [Online]. Available: https://www.exploit-db.com/exploits/33610/

[AJ94] B. Amstadt AND M. K. Johnson, »Wine,« *Linux Journal*, vol. 1994, no. 4es, p. 3, 1994.

[AJ07] J. Asundi AND R. Jayant, »Patch review processes in open source software development communities: A comparative case study,« in *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. IEEE, 2007, pp. 166c–166c.

[Ale96] Aleph One, »Smashing the stack for fun and profit, phrack, 49,« 1996.

[ANSF16] M. Ali, J. Nelson, R. Shea, AND M. J. Freedman, »Blockstack: A global naming and storage system secured by blockchains,« in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[Apa] Official website of the apache http server. [Online]. Available: http://Apache.org/

[att12] (2012) What is the story behind the attack on coiledcoin? [Online]. Available: https://bitcoin.stackexchange.com/questions/3472/what-is-the-story-behind-the-attack-on-coiledcoin#3482

[BAP] »Bap: The next-generation binary analysis platform,« http://bap.ece.cmu.edu.

[Bar14] G. Bartolomucci. (2014) Blazedvd pro 7.0 - '.plf' stack based buffer overflow (direct ret). [Online]. Available: https://www.exploit-db.com/exploits/34331/

[bc03] P. by corbet, »An attempt to backdoor the kernel,« https://lwn.net/Articles/57135/, 2003.

[BC13] A. Bosu and J. C. Carver, »Peer code review to prevent security vulnerabilities: An empirical evaluation,« in *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*. IEEE, 2013, pp. 229–230.

[BC14] A. Bosu and J. C. Carver, »Impact of developer reputation on code review outcomes in oss projects: An empirical investigation,« in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, p. 33.

[BCD+14] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, »Enabling blockchain innovations with pegged sidechains,« *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, 2014.

[BDF+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, »Xen and the art of virtualization,« in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 164–177.

[Bel02] S. M. Bellovin, »A technique for counting natted hosts,« in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 2002, pp. 267–272.

[Bem13] G. G. Bem, »shell_bind_tcp.asm,« 2013, https://github.com/geyslan/SLAE/blob/master/1st.assignment/shell_bind_tcp.asm.

[BHS93] D. Bayer, S. Haber, and W. S. Stornetta, »Improving the efficiency and reliability of digital time-stamping,« *Sequences II: Methods in Communication, Security and Computer Science*, pp. 329–334, 1993.

[bit16a] »Bitcoin,« https://www.bitcoin.org, 2016.

[bit16b] (2016) Bitcoin percentage of total market capitalization. [Online]. Available: https://coinmarketcap.com/charts/#btc-percentage

[BJF11] T. Bletsch, X. Jiang, AND V. Freeh, »Mitigating code-reuse attacks with control-flow locking,« in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. ACM, 2011.

[BJFL11a] T. Bletsch, X. Jiang, V. W. Freeh, AND Z. Liang, »Jump-oriented programming: A new class of code-reuse attack,« in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 30–40.

[BJFL11b] T. Bletsch, X. Jiang, V. W. Freeh, AND Z. Liang, »Jump-oriented programming: a new class of code-reuse attack,« in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

[BJFL11c] T. K. Bletsch, X. Jiang, V. W. Freeh, AND Z. Liang, »Jump-oriented programming: a new class of code-reuse attack,« in *ASIACCS'11*, 2011, pp. 30–40.

[BKL13] A. Buldas, A. Kroonmaa, AND R. Laanoja, *Secure IT Systems: 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. Keyless Signatures' Infrastructure: How to Build Global Distributed Hash-Trees, pp. 313–320. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41488-6_21

[BL15] M. F. BtcDrak AND E. Lombrozo, »Op_checksequenceverify,« https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki, 2015.

[BLT14] A. Buldas, R. Laanoja, AND A. Truu, »Efficient quantum-immune keyless signatures with identity.« *IACR Cryptology ePrint Archive*, vol. 2014, p. 321, 2014.

[Bra99] R. Bracewell, *The Fourier Transform and its Applications*, ser. McGraw-Hill Series in Electrical and Computer Engineering. McGraw-Hill Science/Engineering/Math; 3 edition, June 1999.

[BS14] A. Buldas AND M. Saarepera, »Document verification with distributed calendar infrastructure,« May 6 2014, uS Patent 8,719,576. [Online]. Available: https://www.google.com/patents/US8719576

[btc16] (2016) Bitcoin hash rate. [Online]. Available: http://blockchain.info/charts/hash-rate

[c0n] c0ntex, »Bypassing non-executable-stack during exploitation using return-to-libc,« Available at: http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf.

[CBF+14] J. Clark, J. Bonneau, E. W. Felten, J. A. Kroll, A. Miller, AND A. Narayanan, »On decentralizing prediction markets and order books,« in *Workshop on the Economics of Information Security, State College, Pennsylvania*, 2014.

[CBJW03] C. Cowan, S. Beattie, J. Johansen, AND P. Wagle, »Pointguardtm: Protecting pointers from buffer overflow vulnerabilities,« in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. USENIX Association, 2003, pp. 91–104.

[CDD+10] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, AND M. Winandy, »Return-oriented programming without returns,« in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 559–572.

[CE12] J. Clark AND A. Essex, »Commitcoin: Carbon dating commitments with bitcoin,« in *Financial Cryptography and Data Security*. Springer, 2012, pp. 390–398.

[cha] »Chainpoint,« https://github.com/chainpoint/chainpoint/.

[CJ03] M. Christodorescu AND S. Jha, »Static analysis of executables to detect malicious patterns,« in *In Proceedings of the 12th USENIX Security Symposium*, 2003, pp. 169–186.

[cja] »Hiding code in deterministically built binaries - Proof-of-Concept - Linux/x86,« https://github.com/cjamthagen/backdoor_deterministic_code.

[CLRS09] T. Cormen, C. Leiserson, R. Rivest, AND C. Stein, *Introduction to Algorithms, Third Edition*. MIT Press, 2009.

[Coh92] F. B. Cohen, »Operating system protection through program evolution,« 1992, http://all.net/books/IP/evolve.html.

[Con09] (2009) Conficker's virtual machine detection. [Online]. Available: http://nakedsecurity.sophos.com/2009/03/27/confickers-virtual-machine-detection/

[cou] »Counterparty,« http://counterparty.io/.

[Cov] »Coverity: Software Testing and Static Analysis Tools,« http://www.coverity.com/.

[CPM+98] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, AND Q. Zhang, »Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks,« in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. USENIX Association, 1998, pp. 63–78.

[CVC+02] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, AND M. H. Jakubowski, »Oblivious hashing: A stealthy software integrity verification primitive,« in *International Workshop on Information Hiding*. Springer, 2002, pp. 400–414.

[CW14] N. Carlini AND D. Wagner, »Rop is still dangerous: Breaking modern defenses,« in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, August 2014, pp. 385–399. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini

[CXS+09a] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, AND L. Xie, »Drop: Detecting return-oriented programming malicious code,« in *Information Systems Security*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5905.

[CXS+09b] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, AND L. Xie, »Drop: Detecting return-oriented programming malicious code,« in *International Conference on Information Systems Security*. Springer, 2009, pp. 163–177.

[CZM+14] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, AND R. Deng, »ROPecker: A generic and practical approach for defending against ROP attack,« in *NDSS*. Research Collection School Of Information Systems, 2014.

[dC16] M. del Castillo. (2016) The dao attacked: Code issue leads to $60 million ether theft. [Online]. Available: http://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/

[Deb16] »Debian: Reproducible builds,« https://wiki.debian.org/ReproducibleBuilds, 2016.

[DHP+04] K. Driscoll, B. Hall, M. Paulitsch, P. Zumsteg, AND H. Sivencrona, »The real byzantine generals,« in *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, vol. 2. IEEE, 2004, pp. 6–D.

[DHSZ03] K. Driscoll, B. Hall, H. Sivencrona, AND P. Zumsteg, »Byzantine fault tolerance, from theory to reality,« in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2003, pp. 235–248.

[DL15] S. Demian Lerner. (2015) Rsk: Bitcoin powered smart contracts. [Online]. Available: https://uploads.strikinglycdn.com/files/90847694-70f0-4668-ba7f-dd0c6b0b00a1/RootstockWhitePaperv9-Overview.pdf

[DMS04] R. Dingledine, N. Mathewson, AND P. Syverson, »Tor: The second-generation onion router,« DTIC Document, Tech. Rep., 2004.

[DSLM14] L. Davi, A.-R. Sadeghi, D. Lehmann, AND F. Monrose, »Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection,« in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, August 2014, pp. 401–416. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi

[DSW11] L. Davi, A. Sadeghi, AND M. Winandy, »ROPdefender: A detection tool to defend against return-oriented programming attacks,« in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11, 2011.

[Dur02] T. Durden, »Bypassing PaX ASLR protection, phrack, 59,« 2002.

[Edg10] J. Edge, »A backdoor in UnrealIRCd,« https://lwn.net/Articles/392201/, 2010.

[ES14] I. Eyal AND E. G. Sirer, *Financial Cryptography and Data Security: 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, ch. Majority Is Not Enough: Bitcoin Mining Is Vulnerable, pp. 436–454. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-45472-5_28

[Eva11] C. Evans, »Alert: vsftpd download backdoored,« http://scarybeastsecurity.blogspot.com/2011/07/alert-vsftpd-download-backdoored.html, 2011.

[Fdr15] »F-Droid: Deterministic, reproducible builds,« https://f-droid.org/wiki/page/Deterministic,_Reproducible_Builds, 2015.

[Fla] »Flawfinder,« http://www.dwheeler.com/flawfinder/.

[FLM+08] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, AND L. Van Doorn, »Remote detection of virtual machine monitors with fuzzy benchmarking,« *ACM SIGOPS Operating Systems Review*, vol. 42, no. 3, pp. 83–92, 2008.

[Fra12] I. Fratric, »Ropguard: Runtime prevention of return-oriented programming attacks,« 2012.

[Fyo04] Fyodor. (2004) Return on investment. [Online]. Available: http://insecure.org/stc

[GDN11] C. Gehrmann, H. Douglas, AND D. Nilsson, »Are there good reasons for protecting mobile phones with hypervisors?« in *Consumer Communications and Networking Conference (CCNC), 2011 IEEE*, jan. 2011, pp. 906 –911.

[gey] »shell_bind_tcp.asm,« https://github.com/geyslan/SLAE/blob/master/1st.assignment/shell_bind_tcp.asm.

[Git] »Gitian,« https://gitian.org/.

[GKKB12] A. Gupta, S. Kerr, M. Kirkpatrick, AND E. Bertino, »Marlin: Making it harder to fish for gadgets,« in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. ACM, 2012.

[Glo10] GlobalPlatform, *TEE Client API Specification*, GlobalPlatform, July 2010.

[Glo11a] GlobalPlatform, *TEE Internal API Specification v1.0*, GlobalPlatform, December 2011.

[Glo11b] GlobalPlatform, *TEE System Architecture v1.0*, GlobalPlatform, December 2011.

[Gre12] J. Greene, *Intel Trusted Execution Technology - Hardware-based Technology for Enhancing Server Platform Security*, Intel, 2012.

[Gro11a] T. C. Group, *TPM specification - Design Principles*, Trusted Computing Group, 2011.

[Gro11b] T. D. O. S. Group. (2011) The fiasco microkernel. [Online]. Available: http://os.inf.tu-dresden.de/fiasco

[Gro16] B. Group, »Digital assets on public blockchains,« http://bitfury.com/content/5-white-papers-research/bitfury-digital_assets_on_public_blockchains-1.pdf, 2016.

[Han16] T. Hanke, »Asicboost-a speedup for bitcoin mining,« *arXiv preprint arXiv:1604.00575*, 2016.

[Hen09] R. Hensing, »Understanding DEP as a mitigation technology,« Available at: http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-amitigation-technology-part-1.aspx, 2009.

[hex] »Hex-rays ida pro disassembler,« https://www.hex-rays.com/products/ida/index.shtml.

[HNTC+12] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, AND J. Davidson, »Ilr: Where'd my gadgets go?« in *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012.

[How06] M. A. Howard, »A process for performing security code reviews,« *IEEE Security & privacy*, vol. 4, no. 4, pp. 74–79, 2006.

[HS90] S. Haber AND W. S. Stornetta, *How to time-stamp a digital document*. Springer, 1990.

[HT15] D. A. Harding AND P. Todd, »Opt-in full replace-by-fee signaling,« https://github.com/bitcoin/bips/blob/master/bip-0125.mediawiki, 2015.

[int] »Intel 64 and IA-32 Architectures Software Developer's Manual,« https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[Int13] Intel, *Intel(R) 64 and IA-32 Architectures Software Developer Manuals*, Intel, June 2013.

[Jäm13] C. Jämthagen, »Hidden execution paths project website,« 2013, http://www.eit.lth.se/index.php?uhpuid=dhs.cej&hpuid=864&L=1.

[Jen16] C. Jentzsch. (2016) Dao whitepaper. [Online]. Available: http://download.slock.it/public/DAO/WhitePaper.pdf

[JJV07] M. Jacob, M. H. Jakubowski, AND R. Venkatesan, »Towards integral binary execution: implementing oblivious hashing using overlapped instruction encodings,« in *Proceedings of the 9th workshop on Multimedia & security*, ser. MM&Sec '07. New York, NY, USA: ACM, 2007, pp. 129–140. [Online]. Available: http://doi.acm.org/10.1145/1288869.1288887

[JLH13] C. Jämthagen, P. Lantz, AND M. Hell, »A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries,« in *Anti-malware Testing Research (WATeR), 2013 Workshop on*. IEEE, 2013, pp. 1–9.

[Jor09] G. Jordan. (2009) Stealing profits from stock market spammers. [Online]. Available: https://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-grant_jordan-stock_market_spam.pdf

[KA03] D. Kundur AND K. Ahsan, »Practical internet steganography: data hiding in ip,« in *Proceedings of the Texas workshop on security of information systems*, vol. 2, 2003.

[KC04] C. Kreibich AND J. Crowcroft, »Honeycomb: Creating intrusion detection signatures using honeypots,« *SIGCOMM Computer Communications Review*, vol. 34, no. 1, pp. 51–56, Jan 2004.

[KC06] S. T. King AND P. M. Chen, »Subvirt: Implementing malware with virtual machines,« in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 14–pp.

[Kin10] J. Kinder, »Static analysis of x86 executables,« 2010.

[KK04] H. Kim AND B. Karp, »Autograph: Toward automated, distributed worm signature detection.« in *Proceedings of the 13th Conference on USENIX Security Symposium*. USENIX Association, 2004.

[Lam14] J. Lambert. (2014). [Online]. Available: https://twitter.com/JohnLaTwC/status/442760491111178240

[LD03] C. Linn AND S. Debray, »Obfuscation of executable code to improve resistance to static disassembly,« in *Proceedings of the 10th ACM conference on Computer and communications security*, ser. CCS '03. New York, NY, USA: ACM, 2003, pp. 290–299. [Online]. Available: http://doi.acm.org/10.1145/948109.948149

[Li11] N. Li, »Discretionary access control,« in *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 353–356.

[Lie93] J. Liedtke, »Improving ipc by kernel design,« in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, ser. SOSP '93, 1993.

[LRV09] J. C. Lagarias, E. Rains, AND R. J. Vanderbei, »The kruskal count,« in *The Mathematics of Preference, Choice and Order*. Springer, 2009, pp. 371–391.

[LS10] B. Lau AND V. Svajcer, »Measuring virtual machine detection in malware using dsd tracer,« *Journal in Computer Virology*, vol. 6, no. 3, pp. 181–195, 2010.

[LSP82] L. Lamport, R. Shostak, AND M. Pease, »The byzantine generals problem,« *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[LSPM12] C. LeDoux, M. Sharkey, B. Primeaux, AND C. Miles, »Instruction embedding for improved obfuscation,« in *Proceedings of the 50th Annual Southeast Regional Conference*, ser. ACM-SE '12. New York, NY, USA: ACM, 2012, pp. 130–135. [Online]. Available: http://doi.acm.org/10.1145/2184512.2184543

[LWJ+10] J. Li, Z. Wang, X. Jiang, M. Grace, AND S. Bahram, »Defeating return-oriented rootkits with "return-less" kernels,« in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. ACM, 2010.

[LXG14] K. Lu, S. Xiong, AND D. Gao, »Ropsteg: Program steganography with return oriented programming,« in *Proceedings of the 4th ACM conference on Data and application security and privacy*. ACM, 2014, pp. 265–272.

[LZWG11] K. Lu, D. Zou, W. Wen, AND D. Gao, »Packed, printable, and polymorphic return-oriented programming,« in *Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, R. Sommer, D. Balzarotti, AND G. Maier, Eds. Springer Berlin Heidelberg, 2011, vol. 6961, pp. 101–120.

[MAQ99] H. Massias, X. S. Avila, AND J.-J. Quisquater, »Design of a secure timestamping service with minimal trust requirement,« in *the 20th Symposium on Information Theory in the Benelux*, 1999.

[Mer80] R. C. Merkle, »Protocols for public key cryptosystems.« in *IEEE Symposium on Security and privacy*, vol. 122, 1980.

[Mer88] R. C. Merkle, *Advances in Cryptology — CRYPTO '87: Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, ch. A Digital Signature Based on a Conventional Encryption Function, pp. 369–378. [Online]. Available: http://dx.doi.org/10.1007/3-540-48184-2_32

[Mer14] D. Merkel, »Docker: lightweight linux containers for consistent development and deployment,« *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[MFk15] N. D. Mark Friedenbach, BtcDrak AND kinoshitajona, »Relative lock-time using consensus-enforced sequence numbers,« https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki, 2015.

[Mic] Microsoft. Choosing good passwords: The lmhash. [Online]. Available: https://technet.microsoft.com/en-us/library/dd277300.aspx#ECAA

[MW06] S. McCann AND M. West. (2006) Tcp/ip field behavior. [Online]. Available: http://www.ietf.org/rfc/rfc4413.txt

[MÂÍ08] T. MÂÍuller, »ASLR Smack & Laugh Reference,« 2008. [Online]. Available: http://www-users.rwth-aachen.de/Tilo.Mueller/ASLRpaper.pdf

[Nak08] S. Nakamoto, »Bitcoin: A peer-to-peer electronic cash system,« 2008.

[nam] »Namecoin,« https://namecoin.info/.

[NBZ06] N. Nagappan, T. Ball, AND A. Zeller, »Mining metrics to predict component failures,« in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 452–461. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134349

[neg13] negux, »DEP bypass with ROP,« Available at: http://www.exploit-db.com/exploits/24944/, 2013.

[NKS05] J. Newsome, B. Karp, AND D. Song, »Polygraph: automatically generating signatures for polymorphic worms,« in *IEEE Symposium on Security and Privacy*, May 2005, pp. 226–241.

[Nod10a] Node. (2010) Rm downloader 3.1.3 - local seh exploit (windows 7 aslr + dep bypass). [Online]. Available: https://www.exploit-db.com/exploits/14150/

[Nod10b] Node. (2010) Winamp 5.572 - local buffer overflow (windows 7 aslr + dep bypass). [Online]. Available: https://www.exploit-db.com/exploits/14068/

[obj] »Gnu binutils,« https://www.gnu.org/software/binutils/.

[OBL+10a] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, AND E. Kirda, »G-free: Defeating return-oriented programming through gadget-less binaries,« in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. ACM, 2010, pp. 49–58.

[OBL+10b] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, AND E. Kirda, »G-free: defeating return-oriented programming through gadget-less binaries,« in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 49–58.

[One96] A. One, »Smashing the stack for fun and profit,« Phrack, 1996, http://phrack.org/issues.html?issue=49&id=14#article.

[PaX03] PaX Team, »Address space layout randomization,« Available at: http://pax.grsecurity.net/docs/aslr.txt, 2003.

[P.C11] P.C. Rigby, M-A. Storey, »Understanding Broadcast Based Peer Review on Open Source Software Projects,« in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 541–550. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985867

[PG74] G. J. Popek AND R. P. Goldberg, »Formal requirements for virtualizable third generation architectures,« *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[PGP12] D. Plohmann AND E. Gerhards-Padilla, »Case study of the miner botnet,« in *2012 4th International Conference on Cyber Conflict (CYCON 2012)*. IEEE, 2012, pp. 1–16.

[PK11] M. Polychronakis AND A. Keromytis, »ROP payload detection using speculative code execution,« in *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, ser. MALWARE '11. IEEE Computer Society, 2011.

[PK15] J. Peterson AND J. Krug, »Augur: a decentralized, open-source platform for prediction markets,« *arXiv preprint arXiv:1501.01042*, 2015.

[PPK12a] V. Pappas, M. Polychronakis, AND A. Keromytis, »Smashing the gadgets: Hindering return-oriented programming using in-place code randomization,« in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2012.

[PPK12b] V. Pappas, M. Polychronakis, AND A. D. Keromytis, »Smashing the gadgets: Hindering return-oriented programming using in-place code randomization,« in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 601–615.

[PPK13] V. Pappas, M. Polychronakis, AND A. Keromytis, »Transparent ROP exploit mitigation using indirect branch tracing,« in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, 2013.

[PR04] R. Pagh AND F. F. Rodler, »Cuckoo hashing,« in *Journal of Algorithms*, vol. 51. Duluth, MN, USA: Academic Press, Inc., 2004, pp. 122–144.

[PRA] M. PRATI.

[pro00] T. P. project, Available at: http://pax.grsecurity.net/, 2000.

[Qua15] Qualys, »GHOST: glibc gethostbyname buffer overflow,« Available at: https://www.qualys.com/2015/01/27/cve-2015-0235/GHOST-CVE-2015-0235.txt, 2015.

[RNS] J. Rubin, M. Naik, AND N. Subramanian.

[Ros04] R. Rose, »Survey of system virtualization techniques,« 2004.

[Ros12] M. Rosenfeld, »Overview of colored coins,« *White paper, bitcoil. co. il*, 2012.

[Rus81] J. Rushby, »The design and verification of secure systems,« in *Eighth ACM Symposium on Operating System Principles (SOSP)*, Asilomar, CA, December 1981, pp. 12–21, (ACM *Operating Systems Review*, Vol. 15, No. 5).

[Rut04] J. Rutkowska, »Red pill,« *Or How to Detect VMM Using (almost) One CPU Instruction. Internet Archive: http://web. archive. org/web/20110726182809/http://invisiblethings. org/papers/redpill. html [accessed 25 February 2014]*, 2004.

[Rut06] J. Rutkowska, »Introducing blue pill, 2006,« *SyScan, http://theinvisiblethings. blogspot. com/2006/06/introducing-blue-pill. html*, 2006.

[Rut16] J. Rutkowska. (2016) Critical xen bug in pv memory virtualization code (xsa 182). [Online]. Available: https://github.com/QubesOS/qubes-secpack/blob/master/QSBs/qsb-024-2016.txt

[RW10] J. Rutkowska and R. Wojtczuk, »Qubes os architecture,« *Invisible Things Lab Tech Rep*, p. 54, 2010.

[RW13] J. Rutkowska and R. Wojtczuk, »Qubes os,« 2013.

[SAB11] E. Schwartz, T. Avgerinos, and D. Brumley, »Q: Exploit hardening made easy,« in *Proceedings of USENIX Security 2011*, 2011.

[SC13] S. Smalley and R. Craig, »Security enhanced (se) android: Bringing flexible mac to android,« *20th Annual Network and Distributed System Security Symposium (NDSS '13)*, February 2013.

[Sch08] C. Schaufler, »Smack in embedded computing,« in *Proceedings of the Linux Symposium*, 2008, pp. 179–186.

[SD15] M. Seaborn and T. Dullien, »Exploiting the dram rowhammer bug to gain kernel privileges,« *Black Hat*, 2015.

[SE11] ST-Ericsson, *The NovaThor platforms for smartphones and tablets*, ST-Ericsson, 2011.

[Sec10] SecurityFocus.com, »ProFTPD Backdoor Unauthorized Access Vulnerability,« http://www.securityfocus.com/bid/45150, 2010.

[Ser09] F. J. Serna, »CVE-2012-0769, the case of the perfect info leak,« Available at: http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf, 2009.

[SH12] M. Sikorski and A. Honig, *Practical Malware Analysis, the Hand-On Guide to Dissecting Malicious Software*. no starch press, 2012.

[SH13] F. Schuster and T. Holz, »Towards reducing the attack surface of software backdoors,« in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 851–862.

[Sha07a] H. 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*, ser. CCS '07. ACM, 2007, pp. 552–561.

[Sha07b] H. 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*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315313

[sic11] sickness. (2011) Dvd x player 5.5.0 pro / standard - universal exploit (aslr + dep bypass). [Online]. Available: https://www.exploit-db.com/exploits/17754/

[Sie13a] Sierraware, *SierraTEE for ARM TrustZone*, Sierraware LLC, 2013.

[Sie13b] Sierraware, *SierraVisor Hypervisor*, Sierraware LLC, 2013.

[SJ04] P. Silberman and R. Johnson, »A comparison of buffer overflow prevention implementations and weaknesses,« *IDEFENSE, August*, 2004.

[SLGL08] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, »Impeding malware analysis using conditional code obfuscation.« in *NDSS*, 2008.

[SMD+13] K. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, »Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization,« in *Security and Privacy (SP), 2013 IEEE Symposium on*, May 2013, pp. 574–588.

[SML10] J. Sahoo, S. Mohapatra, and R. Lath, »Virtualization: A survey on concepts, taxonomy and associated security issues,« in *Computer and Network Technology (ICCNT), 2010 Second International Conference on*. IEEE, 2010, pp. 222–226.

[Spl] »Splint,« http://www.splint.org/.

[SPP+04] H. Shacham, M. Page, N. Pfaff, E. Goh, N. Modadugu, and D. Boneh, »On the effectiveness of address-space randomization,« in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. ACM, 2004, pp. 298–307.

[SSO+13] B. Stancill, K. Z. Snow, N. Otterness, F. Monrose, L. Davi, and A.-R. Sadeghi, »Check my profile: Leveraging static analysis for fast and accurate detection of ROP gadgets,« in *Research in Attacks, Intrusions, and Defenses*, ser. Lecture Notes in Computer Science, S. Stolfo, A. Stavrou, and C. Wright, Eds. Springer Berlin Heidelberg, 2013, vol. 8145, pp. 62–81.

[SSZ15] A. Sapirshtein, Y. Sompolinsky, AND A. Zohar, »Optimal selfish mining strategies in bitcoin,« *CoRR*, vol. abs/1507.06183, 2015. [Online]. Available: http://arxiv.org/abs/1507.06183

[sta16] (2016) State of cyber security: Implications for 2016. [Online]. Available: http://www.isaca.org/cyber/Documents/state-of-cybersecurity_res_eng_0316.pdf

[STP+14] F. Schuster, T. Tendyck, J. Pewny, A. MaaÃ§, M. Steegmanns, M. Contag, AND T. Holz, »Evaluating the effectiveness of current anti-rop defenses,« in *Research in Attacks, Intrusions and Defenses*, ser. Lecture Notes in Computer Science, A. Stavrou, H. Bos, AND G. Portokalidis, Eds., vol. 8688. Springer International Publishing, 2014, pp. 88–108.

[SW11] Y. Shin AND L. Williams, »An initial study on the use of execution complexity metrics as indicators of software vulnerabilities,« in *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, ser. SESS '11. New York, NY, USA: ACM, 2011, pp. 1–7. [Online]. Available: http://doi.acm.org/10.1145/1988630.1988632

[sza97] »The idea of smart contracts,« http://szabo.best.vwh.net/idea.html, 1997.

[Szt15] P. Sztorc. (2015) Peer-to-peer oracle system and prediction marketplace. [Online]. Available: http://bitcoinhivemind.com/papers/truthcoin-whitepaper.pdf

[td09] A. technical documentation, *ARM Security Technology - Building a Secure System using TrustZone Technology*, ARM, 2009.

[Tea16] P. Team. (2016) Rap: Return address protection. [Online]. Available: https://www.grsecurity.net/rap_announce.php

[THT05] T. H. Toshiharu Harada AND K. Tanaka. (2005) Towards a manageable linux security. [Online]. Available: http://sourceforge.jp/projects/tomoyo/docs/lc2005-en.pdf/en/2/lc2005-en.pdf.pdf

[Tin09] J. Tinnes. (2009) Linux null pointer dereference due to incorrect proto-ops initializations (cve-2009-2692). [Online]. Available: http://blog.cr0.org/2009/08/linux-null-pointer-dereference-due-to.html

[Tor15] »Tor: Deterministic builds,« https://blog.torproject.org/category/tags/deterministic-builds, 2015.

[vB09] E. van Buskirk, »Denial-of-service attack knocks twitter offline,« http://www.wired.com/2009/08/twitter-apparently-down/, 2009.

[VC09] S. V. Vugt AND R. Clark, *Pro Ubuntu Server Administration*. Apress, 2009.

[Ven00] Vendicator, »Stack shield: A "stack smashing" technique protection tool for linux,« Available at: http://www.angelfire.com/sk/stackshield/, 2000.

[Vir] Oracle virtualbox vm, user manual. [Online]. Available: http://www.virtualbox.org/manual/

[Vre10] P. Vreugdenhil, »Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit,« Available at: http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf, 2010.

[wel13] welivesecurity.com, »Linux/SSHDoor.A Backdoored SSH daemon that steals passwords,« http://www.welivesecurity.com/2013/01/24/linux-sshdoor-a-backdoored-ssh-daemon-that-steals-passwords/, 2013.

[WHS12] M. Weiss, B. Heinz, AND F. Stumpf, »A cache timing attack on aes in virtualization environments,« in *14th International Conference on Financial Cryptography and Data Security (Financial Crypto 2012)*, ser. Lecture Notes in Computer Science. Springer, 2012.

[WLL+13] T. Wang, K. Lu, L. Lu, S. Chung, AND W. Lee, »Jekyll on ios: When benign apps become evil,« in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 559–572.

[WMHL12] R. Wartell, V. Mohan, K. Hamlen, AND Z. Lin, »Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code,« in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012.

[Woo14] G. Wood, »Ethereum: A secure decentralised generalised transaction ledger,« *Ethereum Project Yellow Paper*, 2014.

[WPLZ10] X. Wang, C. Pan, P. Liu, AND S. Zhu, »Sigfree: A signature-free buffer overflow attack blocker,« *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 1, pp. 65–79, Jan 2010.

[WZH+11] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, AND B. Thuraisingham, »Differentiating code from data in x86 binaries,« in *Machine Learning and Knowledge Discovery in Databases*, ser. Lecture Notes in Computer Science, D. Gunopulos, T. Hofmann, D. Malerba, AND M. Vazirgiannis, Eds. Springer Berlin Heidelberg, 2011, vol. 6913, pp. 522–536. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23808-6_34

[xen16] (2016) List of common vulnerabilities and exposures for xen. [Online]. Available: https://xenbits.xen.org/xsa/

[xpl11] xploitedsec. (2011) The kmplayer 3.0.0.1440 - '.mp3' buffer overflow (windows 7 + aslr bypass). [Online]. Available: https://www.exploit-db.com/exploits/17383/

[Y. 03] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, M.H. Jakubowski, »Oblivious Hashing: A Stealthy Software Integrity Verification Primitive,« in *Revised Papers from the 5th International Workshop on Information Hiding*, ser. IH '02. London, UK, UK: Springer-Verlag, 2003, pp. 400–414. [Online]. Available: http://dl.acm.org/citation.cfm?id=647598.732027

[YHD+16] K. Yang, M. Hicks, Q. Dong, T. Austin, AND D. Sylvester, »A2: Analog malicious hardware,« 2016.

[YSD+09] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, AND N. Fullagar, »Native client: A sandbox for portable, untrusted x86 native code,« in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 79–93.

[ZS13] M. Zhang AND R. Sekar, »Control flow integrity for COTS binaries,« in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. USENIX Association, 2013.

[ZW11] C. J. D. G. Z. Wang, J. Ming, »Linear Obfuscation to Combat Symbolic Execution,« in *Computer Security - ESORICS 2011*, ser. Lecture Notes in Computer Science, V. Atluri AND C. Diaz, Eds. Springer Berlin Heidelberg, 2011, vol. 6879, pp. 210–226. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23822-2_12

[ZWC+13] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, AND W. Zou, »Practical control flow integrity and randomization for binary executables,« in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. IEEE Computer Society, 2013.