MONASH University

Formal Model of Exploit-Resistant Systems

Sepehr Minagar
Bachelor of Computer Hardware Engineering
Master of Information Technology (Honours)

A thesis submitted for the degree of Doctor of Philosophy at
Monash University in 2018
Faculty of Information Technology

Copyright notice

© Sepehr Minagar 2018

Abstract

Malicious code execution is a form of exploitation that plays a significant role in various forms of cyber crime. The statistics, of vulnerabilities reported in the last few years that could lead to arbitrary code execution, show that malicious code execution is still a significant problem. The body of research can be divided into two branches: (i) Vulnerability Removal; (ii) Detection and Prevention of Exploitations.

Currently there is an iterative trend of proposed solutions that are followed by circumvention methods. This is due to the fact that the majority of the solutions do not follow a formal approach in expressing the proposed method. A formal approach can provide the proof of correctness for a solution that can withstand any future attack. Providing a formal method will also clearly state the assumptions or preconditions that need to be satisfied for the proof to be valid. Unfortunately, the lack of formal approaches has led to solutions that have hidden assumptions. Another shortcoming of ad-hoc solutions is unclear guarantees.

The underlying cause of almost all remote malicious code execution attacks is the memory corruption where the adversary gains write access to memory outside the scope provisioned by the programmer. The adversary then can use various methods to exploit this capability. The exploitation method can be divided into two broad categories: (i) Code Injection; (ii) Code Reuse. In code injection the adversary uses the memory corruption capability to write instructions in the memory of the target machine to achieve a malicious intent. The flow of execution is then changed to the injected code in the target machine's memory. The code reuse can be divided into two subtypes: (i) Control Flow Hijack; (ii) Control Flow Bending. In both methods the adversary changes the flow of execution to an existing code in the target machine's memory. The difference between control flow hijack and control flow bending is that in the former the control flow is violated compared to benign execution of the program, whereas in the latter form of attack the flow of execution follows a valid edge of the benign execution of the program. Another form of attack defined as memory leakage is used to gain information about the target machine,

such as memory addresses for code and data that can be used in crafting malicious code execution attacks.

This thesis develops a theoretical model for exploit-resistant systems that can prevent the code execution on a vulnerable target machine. The Ideal Control Flow Integrity (ICFI) model protects against code injection and code reuse leading to control flow hijack attacks by protecting all of the valid edges of the flow of execution. To address the difficulty of the dynamic linking in previous formal work of the classic Control Flow Integrity, a dynamic registration is provided for authentic calls and their corresponding returns. This model is the first formal work that is provably secure against code injection and code reuse leading to control flow hijack attacks.

The memory model divided into integrity and confidentiality, prevents memory corruption and address leakage attacks respectively. The model enforces defined memory protection rules through a combination of memory cell protection levels and machine instruction micro operations that verify these levels. The integrity model protects against corruption attacks that could lead to any form of malicious code execution on the target machine, including code reuse leading to control flow bending which cannot be prevented by the ICFI method. The integrity model can provably protect against memory corruption attacks leading to control flow change including the non-control data attack. The confidentiality model will prevent the address leakage which can be used in preparing malicious code execution attacks. The formal approach enables proving the correctness of the solution as well as general applicability for various architectures, operating systems, and compilers.

An implementation of code memory authenticity is provided as a proof of concept for content-based protection of executable memory. This is achieved by combining the social concept of trust in producer of the code and integrity of the code content by performing code authentication before execution. In this implementation the code of the executable is verified in a demand paging approach where only the part

of the code that must be loaded into memory for execution is verified, reducing the verification time to its minimum.

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature: . . . .

Print Name: **Sepehr Minagar**

Date: 5 July 2018

Publications during enrolment

Minagar, S., Srinivasan, B. and Le, P.D., 2017, December. A Formal Model for an Ideal CFI. In International Conference on Information Security Practice and Experience (pp. 707-726). Springer, Cham. Melbourne, Australia.

Acknowledgements

This research was supported by a Monash Graduate Scholarship.

This research was supported by an Australian Government Research Training Program (RTP) Scholarship.

There are no words by which I could express my eternal debt and gratitude to my parents and my two lovely sisters for their boundless love and support.

I would like to thank professor Bala Srinivasan for his continuous support, everlasting patience, and invaluable feedbacks without which this thesis would not be in existence.

I would like to thank Dr. Phu Dung Le for his support and guidance during my candidature.

I would also like to thank Dr. Viranga Ratnaike for proofreading my work. His thesis was covering the areas of knowledge representation and emergent semantics with no little, if any, overlap with my work.

I am also thankful to all my friends who certainly have made life more enjoyable here in Australia.

Contents

List of Tables . . . . **xiii**

List of Figures . . . . **xiv**

1 Introduction **1**

1.1 Motivation 1

1.2 Background 2

1.3 Research Goals 4

1.4 Contribution 5

1.5 Thesis Structure 8

2 Literature Review **11**

2.1 Introduction 11

2.2 Vulnerability Removal 12

2.2.1 Formal Software Verification 12

2.2.2 Model Checking 12

2.2.3 Static Analysis 14

2.3 Safe Programming Language 15

2.3.1 Type Safety 16

2.3.2 Proof Carrying Code 17

2.4 Exploitation Prevention 18

2.4.1 Non-executable Stack or Data Memory 18

2.4.2 Address Space Layout Randomisation 21

2.4.3 Return Oriented Programming and Defences 22

2.4.4 Heap-based Attacks . . . 28
2.4.5 Control Flow Integrity . . . 29
2.4.6 Coarse-Grained CFI . . . 33
2.4.7 Fine-Grained CFI . . . 37
2.4.8 Attacks on CFI . . . 44
2.4.9 Proposed Method . . . 45
2.5 Summary . . . 46

3 A Formal Model of Ideal Control Flow Integrity . . . 47
3.1 Introduction . . . 47
3.2 Instruction Types and Flow of Execution . . . 48
3.3 Machine Model . . . 50
3.3.1 Notations . . . 51
3.3.2 Propositional Dynamic Logic . . . 53
3.4 Attack Model . . . 58
3.5 Protective Measures . . . 62
3.5.1 Forward Edge . . . 64
3.5.2 Backward Edge . . . 66
3.6 Theorems of ICFI . . . 70
3.6.1 Premises . . . 70
3.6.2 Theorems . . . 71
3.7 Summary . . . 73

4 Memory Integrity Model . . . 77
4.1 Introduction . . . 77
4.2 Memory Corruption . . . 78
4.3 Memory Safety . . . 79
4.3.1 Spatial Memory Safety . . . 79
4.3.2 Temporal Memory Safety . . . 83
4.3.3 Complete Memory Safety . . . 84

4.3.4 Tag-based Architecture 84
4.3.5 Type-based Non-interference Languages 85
4.4 Memory Model 86
4.4.1 Memory and Register Representation 87
4.4.2 Preliminary Definitions for Memory Integrity Model 87
4.4.3 Requirements of Memory Integrity Model 94
4.4.4 Instructions Enforcing Memory Integrity Model 103
4.4.5 A Discussion on Compiler Requirements 115
4.4.6 Theorem of the Memory Integrity Model 120
4.5 Memory Confidentiality Model 123
4.5.1 Preliminary Definitions 123
4.5.2 Requirements of Memory Confidentiality Model 125
4.5.3 Instruction Requirements for Confidentiality Model Enforcement 127
4.5.4 Theorem of Confidentiality Model 132
4.6 Summary 134

5 Realisation of Code Memory Authenticity and Alternative Memory Models 137
5.1 Introduction 137
5.2 Code Memory Integrity 139
5.2.1 A Signature-based Code Memory Integrity 139
5.2.2 Memory Blocks and Paging 141
5.2.3 Implementation of Page-by-Page Verification 144
5.2.4 Code Integrity Verification in Linux 148
5.2.5 Content-based Access Control and Authentication: A Discussion 152
5.3 x86 Sample Code Analysis 153
5.3.1 Simple Buffer Overflow in ICFI Model 154
5.3.2 Simple Buffer Overflow in Memory Integrity Model 158
5.4 Address Space Division for Memory Integrity Model 160

5.4.1 Division of Address Space 160
5.4.2 Instruction Requirements for Address Space Division 161
5.5 Protection of Operating System Memory 165
5.5.1 Memory Privilege Model 166
5.5.2 Requirements of the Memory Privilege Model 169
5.5.3 Instruction Requirements for the Memory Privilege Model . . 170
5.6 Summary 179

6 Conclusion 181
6.1 Introduction 181
6.2 Contributions 182
6.3 Future Research 185

References 187

Appendix A Page-by-Page Verification for Linux Kernel 205
A.1 Introduction 205
A.2 Modification of Linux Kernel 205
A.2.1 Modification of main.c 205
A.2.2 Required Changes in main.c 214
A.2.3 Code of Modified memory.c 220
A.2.4 Required Changes to memory.c 241
A.2.5 Compiling Kernel with initrd 252
A.2.6 Message Authentication Code for Executable Code Pages . . . 253
A.2.7 Time Measurements of MAC Function for Sample Executable Files 258

List of Tables

2.1 Static Analysis Precision for Forward edge in Burow et al. survey [1] 36

3.1 Notation summary 52

3.2 Machine instructions and semantics 53

3.3 PDL expressions of machine instructions 57

3.4 Instruction preconditions to prevent control flow hijack attack 75

4.1 Notation summary 104

4.2 Summary of instruction semantics 105

5.1 Time Measurements for the Page-by-Page Verification Method versus the Entire Content Verification Method (μ seconds) 151

5.2 Memory Privilege Levels 166

A.1 MAC function time measurements for `/bin/tar` file (μ seconds) . . . 258

A.2 MAC function time measurements for `/bin/grep` file (μ seconds) . . 258

A.3 MAC function time measurements for `/bin/ls` file (μ seconds) 258

List of Figures

1.1 Reported vulnerabilities that could lead to arbitrary code execution . 3

2.1 Buffer overflow attack with injected shell code 19

2.2 The general idea of return-into-libc exploitation technique 20

2.3 The general idea of Return Oriented Programming 23

2.4 Loading a value into `%edx` register using ROP 23

2.5 Storing the content of `%eax` into memory cell at `%edx+24` using ROP 24

2.6 Difficulty of implementing CFI with library functions 31

2.7 Difficulty of implementing CFI with library functions in equivalent classes technique 32

3.1 Control Flow Graph in dynamically linked executable 63

3.2 Set of Authentic Calls for the executable `foo` and library `lib`. 65

3.3 Functions' entry points and their associated return point(s) 67

3.4 Verifying an authentic call (forward edge) 68

3.5 Run-time mappings of associated return point(s) of authentic function calls 69

4.1 Relations between defined sets for program variables, assigned memory and process memory 98

5.1 Page message authentication code generation process. 146

5.2 Folder name generation for page message authentication code. 147

Chapter 1

Introduction

1.1 Motivation

The role of computer systems in our daily life is ever increasing. The number of systems capable of executing programs will only increase with the addition of a variety of devices such as smartphones, tablets, and the Internet of Things. Errors in design, implementation and usage of these devices, operating systems, and applications running on these devices will also increase as long as security is not considered in all phases. These errors will lead to vulnerabilities that when exploited can cause a variety of problems. Malicious code execution is a form of exploitation that runs a piece of code crafted by the attacker on a targeted vulnerable machine to achieve nefarious goals. This could be used to gain access to the target machine to disclose sensitive information to the attacker which then may be used for financial gain. It can be used in digital espionage by a state against other states, organisations against competition, or state surveillance violating individuals' privacy and freedom. The attack on larger organisations, whether private or governmental, could have more adverse results, leading to disclosure of millions of records of sensitive information of the customers of the targeted organisations. The malicious remote code execution can be combined with other malware to form more advanced forms of attack, for instance in crafting ransomeware in digital extortion.

Preventing a successful exploitation can be achieved by one of the two possible approaches. The first approach is to remove all the vulnerabilities within the system. A computer system that has no vulnerability cannot be exploited. This requires the design, implementation, and usage to be all flawless which has been deemed very difficult to achieve. The second approach is to detect and protect against exploitation. Here a system that is vulnerable can resist the exploitation attempts. The benefit of such a system is that some of the errors or weaknesses can be tolerated which leads to faster development without the adverse effects of security attacks. It will also allow the usage of some of the development tools, such as efficient programming languages that lack security by design, with relatively small changes in the programming and compiler.

Common Vulnerability and Exposure (CVE), maintained by an international community, is a standardised database of publicly known security vulnerabilities [2]. Each vulnerability receives a unique identifier in the database when discovered and reported along with a standard description. A review of the reported vulnerabilities in the CVE database that could lead to arbitrary code execution shows that a steady number of new vulnerabilities have been discovered in the past eight years (Figure 1.1) [3]. The increase in the number of reported vulnerabilities in recent years shows the relevance and significance of this problem today.

1.2 Background

The programs in execution are stored in volatile system memory, hence the first step for malicious code execution is to gain access to system memory. Memory corruption is the underlying access point for almost all methods of exploitation that lead to malicious code execution. These methods of exploitation can be divided into two broad categories: (i) Code Injection; and (ii) Code Reuse.

In the Code Injection method the attacker uses the memory corruption capability to copy his or her crafted machine instructions into the target machine's memory, and change the flow of execution to the injected code.

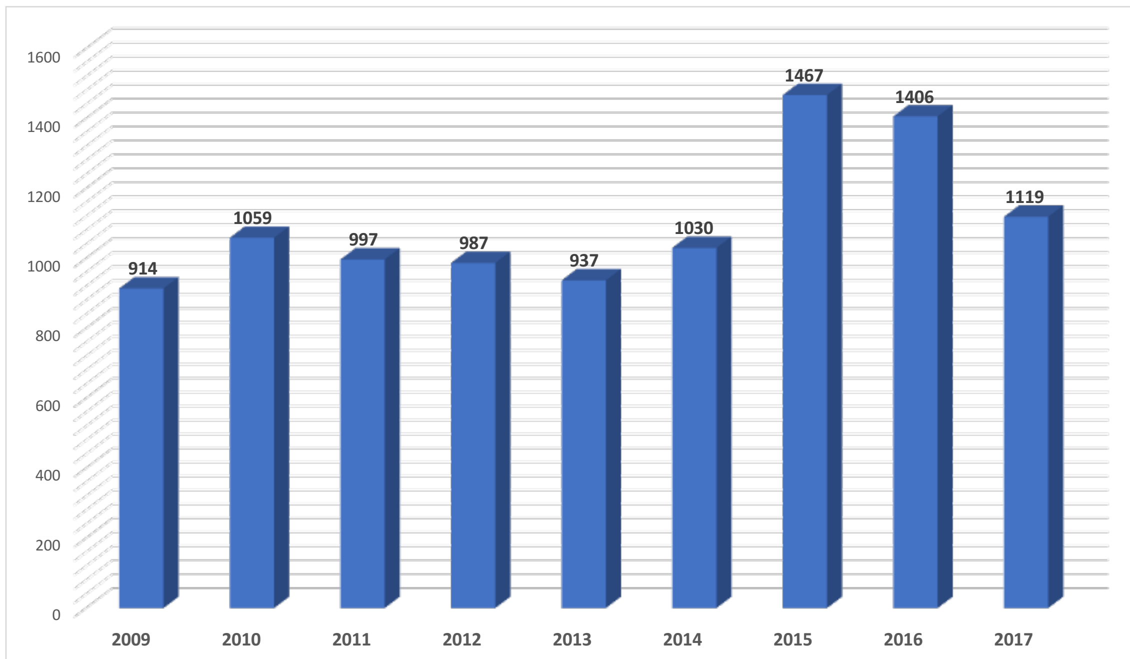

Figure 1.1: Reported vulnerabilities that could lead to arbitrary code execution

In the Code Reuse method the attacker changes the flow of execution to code that already exists in the target machine's memory as part of either the code of the vulnerable executable or the shared libraries. In this method the attacker injects selected addresses and parameters to guide the flow of execution toward his or her malicious intent, choosing functions or sets of instructions in the memory of the target machine.

If the intended execution of the program is defined as the benign execution of the program then the latter method can be divided into two subclasses. The first subclass can be defined as the attack that violates the control flow of the benign execution by changing it to a function or instruction that has never been part of the benign execution. This method of attack is called control flow hijack. The second subclass can be defined as the attack that changes the flow of execution to a path that does not violate the benign execution of the program. This is achieved by corrupting the non-control data of the program to influence the flow of execution. This method of attack is called control flow bending.

Considering that both code injection and code reuse leading to control flow hijack violate the benign execution path of an exploitable program, then both can be

classified as control flow hijack attacks. Another form of exploitation is information leakage where the adversary by exploiting the targeted programs discloses sensitive information such as cryptographic keys or memory addresses. The leaked memory addresses can be used in crafting exploitation where the attacker needs to change the flow of execution to a given address in the target system memory.

In the past few decades various methods have been proposed to counter various exploitation techniques. One common shortcoming of the majority of these methods is the lack of logically provable solutions. The problem with ad-hoc and heuristic solutions is that the assumptions are not defined clearly. For instance, in the case of a stack canary[1] the assumption is that to overwrite the return address stored in the stack the value of the canary must also be overwritten. This does not take into account the possibility of precisely overwriting the return address without overwriting the canary value which is possible with integer overflow of array indices. The other problem with informal solutions is that the provided guarantees are not well defined. For instance, in the case of Address Space Layout Randomisation or various forms of Coarse-Grain Control Flow Integrity, it is not clear to what extent the solution makes the system more secure.

1.3 Research Goals

To prevent a successful exploitation its requirements must be removed. To achieve this, the common requirements of exploitations that could lead to code execution must first be identified, as it is desirable to provide a solution that is generally applicable and not specific to one form of exploitation. All exploitation techniques that execute code on a target machine have to change the flow of execution to the adversary's intended code. The goal is to identify how this is achieved and how it can be prevented, regardless of the vulnerability of the target system. In order to

[1] A random value stored after the pushed return address at top of the stack. The given name refers to canaries used in coal mines as a warning method for the level of carbon monoxide.

prove the correctness of the solution a branch of formal logic is used to express the proposed work in this thesis.

1.4 Contribution

Current architectures allow a single privilege memory address space with flexible indirect jump instructions that can change the flow of execution to any location within that range. This flexible design combined with the mixture of control and non-control data storage in the same address space and memory corruption vulnerabilities enable the attacker to circumvent most of the proposed solutions. The solutions that can provide complete protection against particular vulnerabilities or exploitations are often too expensive to implement in current architectures. Proposing alternative architectural designs for processor instruction sets and memory structures can lead to better architectures with security policy enforcements embedded as part of an instruction set that would remove the possibility of circumvention. Usage of a formal approach to provide proofs of correctness and security will provide strong guarantees that will not be invalidated in time.

The contributions can be divided into the following:

(C_1) *Ideal Control Flow Integrity (ICFI) model to prevent control flow hijack attacks*: For a control flow hijack attack to succeed, the flow of execution must be changed from the programmer's intended path that this thesis refers to as benign execution, to the beginning of the adversary's intended code. ICFI is a formal model that can prevent all forms of control flow hijack attacks on a target machine. The ICFI model is the first formal method that allows dynamic linking and relocatable executables and is provably secure. It protects both forward indirect jumps used for function calls and backward indirect jumps used for function returns. ICFI uses the most precise control flow graph possible for an executable code for both forward and backward edges. The integrity precondition of the executable code assures the content of the executable code

is intact throughout the execution of the code. The model can be used as a foundation to evaluate whether an implementation satisfies any of the defined requirements, for modern processors, operating systems, or compilers.

(C_2) *Memory Integrity model to prevent control flow hijack and control flow bending attacks*: The control flow bending attack uses non-control data to change the flow of execution without violating the benign execution path of the code. This is due to the corruption of the user data that affects the flow of execution. To protect against this type of attack a memory integrity model is defined that divides all program variables into two groups: Trusted and Untrusted. The model protects the trusted variables by marking the memory cells associated to these variables as having higher level of integrity compared to untrusted variables. A method of classification of untrusted variables is defined as part of the policy enforcement of the model to allow flow of information from user-provided input. This model is provably secure and can prevent memory corruption that leads to control flow hijack and control flow bending attacks.

(C_3) *Memory Confidentiality model to prevent memory address leakage*: Exploitations that disclose memory addresses to the attacker are used in crafting code execution attacks. For instance if the address of the commonly used C library is disclosed to the attacker, it can be used to craft return-into-libc code reuse attacks or any of the Return Oriented Programming or Jump Oriented Programming attacks. To protect against this type of attack a memory confidentiality model is defined. Similar to the integrity model, the memory confidentiality model marks the memory cells containing addresses differently to all other variables. By defining rules that are enforced using the machine instructions as well as the compiler, address leakage attacks are prevented.

(C_4) *An implementation of Code Memory Authenticity as a proof of concept for memory integrity precondition of the ICFI model*: Code memory integrity is a

precondition that must be satisfied in all machine states during the execution of a program in the ICFI model. A code memory authenticity for a Linux kernel is implemented using the demand paging concept, where the code is divided into fixed-length non-overlapping pages and only required pages are loaded into memory. The implementation authenticates each code page using a system-wide key before the flow of execution is transferred to the newly loaded page. Since loading the demanded page into memory is part of the operating system responsibility, the Linux kernel is modified to perform the authentication. The implementation shows the feasibility and the cost of satisfying the code integrity precondition of the ICFI model using the existing architecture. It also uses authentication instead of integrity to emphasise the existing social model of trust in the producer of the code not to have malicious intent rather than finding methods to detect or deduce the intent from the executable code.

(C_5) *Address Space Separation model as a more cost-effective alternative to the memory integrity model*: The realisation of the memory integrity model requires an additional bit for every cell of the volatile memory of the system. In the Address Space Separation model, the process address space is divided into two parts: low and high integrity levels. A register will contain the address of the first memory cell that belongs to the high integrity address space of a process, allowing flexible division of the address space between low and high integrity memory cells. The required changes to the policy enforcement is then expressed in micro operations of the machine instructions. This alternative memory integrity model eliminates the need to change the structure of the system memory as well as the additional bit per cell requirement of the original model.

(C_6) *Memory Privilege model as an alternative memory model to combine integrity and confidentiality models*: The combination of memory integrity and confidentiality models requires the adjustment of the policy enforcement for each machine instruction. This model shows the possibility of the combined model

and introduces the additional protection of operating system memory against malicious processes.

1.5 Thesis Structure

The thesis is structured into six chapters starting with this introduction. Chapter Two provides a literature review that covers various topics. It starts with a review of the proposed methods that remove vulnerabilities by either finding the vulnerability in the body of the code or by defining safe programming languages that remove the possibility of certain types of errors. Then there is a discussion of methods that prevent a successful exploitation. These can be categorised into various types. Chapter Two ends with a discussion of the various methods of the Control Flow Integrity (CFI) which influences this work.

Chapter Three defines the ICFI model (C_1). A description of Propositional Dynamic Logic (PDL) is provided. It is used in expressing the attack and preventive measures in the formal model. The instructions of a machine model proposed in the literature is defined, and then modified according to the requirements of the ICFI model. The types of instructions and how they affect the flow of execution is then discussed followed by definition of required protective measures. The theorems and proof of the scheme is then discussed and the chapter concludes with a summary.

The memory integrity and confidentiality models are discussed in Chapter Four (C_2 and C_3). A review of the literature related to memory corruption and proposed solutions is provided at the beginning of the chapter. To define the requirements of the memory integrity model, the gap between high level programming languages and machine language concepts is bridged. The abstract concepts related to code and functions in high level programming languages are connected to machine language concrete instructions using preliminary definitions which are at times refined as the chapter progresses. The chapter then specifies the requirements of the integrity model. The requirements should be satisfied partly by the programmer; partly by the compiler and micro operations of the machine instructions. Once all of the

requirements are defined, the theorem is expressed and its proof is discussed. The memory confidentiality model section follows a similar structure and the chapter concludes with a discussion.

Chapter Five is dedicated to realisation ideas. The concepts of the authenticity of the code and the socially accepted trust in the producer of the code are discussed at the beginning of this chapter. Then follows a description of how this can be implemented using a public or private key signature scheme and in a system with on-demand paging. As a proof of concept, a page-by-page verification of executable code is implemented including the shared libraries for the Linux operating system (C_4). The logic of the page-by-page verification for a Linux kernel is discussed after the formal evaluation of the effect of on-demand paging on the proof of the ICFI method and the relation of authenticity and integrity of the code pages. The details of the implementation and the code of modified kernel source files are provided in Appendix of the thesis. An example of a compiled code of a program vulnerable to buffer overflow is then provided to show how each of the formal models would protect against this attack. The chapter then proposes an alternative realisation of memory integrity to decrease the cost in terms of memory space (C_5). The chapter shows how the instructions must change to accommodate such a realisation. The memory integrity and confidentiality models can be realised in a combined system (C_6). The combined model also includes the protection of operating system against malicious user processes. The required changes in instructions to achieve this combined model that protects the program against malicious users and operating system against malicious processes is then discussed. The chapter concludes with a discussion on other realisation aspects of the thesis.

The thesis concludes with a final chapter that summarises the contributions and discusses future work.

Chapter 2

Literature Review

2.1 Introduction

Successful execution of malicious code by an adversary on a target system relies on two requirements, the existence of vulnerabilities and a method of exploiting those vulnerabilities. The proposed protective methods in the literature can be categorised into two broad classes, namely removal of vulnerabilities and detection and prevention of exploitations. This chapter discusses various solutions proposed in the literature in both classes to protect computer systems against malicious code execution.

Various vulnerability removal methods and the difficulties of such solutions are described in Section 2.2. Eliminating the possibility of errors leading to the vulnerabilities, by removing features of the high level programming language or the low level machine language, is discussed in Section 2.3.

The methods of detection and prevention of the exploitation are studied in Section 2.4. An overview of the proposed method is given in Section 2.4.9 and a summary in Section 2.5.

2.2 Vulnerability Removal

This section discusses various methods of analysing the code to detect and remove vulnerabilities. The proposed methods range from formal through semi-formal to ad-hoc or heuristic.

2.2.1 Formal Software Verification

Formal software verification refers to techniques that aim at providing a formal method to verify if a software artefact meets its design goals and defined requirements [4], [5]. The formal aspect refers to techniques that are based on logic, set theory, and algebra. These methods are used to define the specification of software systems models and verification of their properties. A software system abstract model can be used to verify if the system satisfies a given set of functional requirements as well as assuring a set of defined non-functional properties [4].

2.2.2 Model Checking

Model Checking is a formal method whose goal is to detect behavioural anomalies of software systems by performing formal verification of suitable models of those systems. In this method a state reachability analysis is performed in which the verifier searches the state space for error states where the program fails to satisfy a specified property. The faults that made the error state reachable are then removed and the procedure is repeated [4]. Chen and Wagner [6] discuss a model checking approach in finding potential violation in sequence of security operations e.g. system calls, by modelling the security property as Finite State Automata and the program as a Push Down Automaton. Schwarz et al. [7] further extend their technique to analyse the entire Red Hat Linux 9 distribution with around 60 million lines of code. There are a few obstacles that prevent the integration of this technique into the development process. One obstacle is comprised of gaps between the concepts,

notations, and models that are used to design large-scale systems. Another obstacle is the scalability problem [4].

There exist techniques that will help reduce the state space and overcome the scalability issue. One such technique is *Predicate Abstraction* where only certain predicates over data are tracked, instead of tracking program data as the state [4]. The predicates are expressed in Boolean variables and compose an over-approximation of the original program. The use of predicates as Boolean variables reduces the size of the state space compared to the program state defined by the value of its data. Another technique is the use of pre- and post-conditions for procedures of a program and an automated theorem proving tool to check the defined conditions. In this method a statement that is a logical formula with true or false values identifies the conditions that must be met before entering a procedure. The pre-condition only involves the global variables and the input arguments of the procedure. The post-condition on the other hand describes the outcome of the procedure, and is accomplished if the procedure is called and its pre-conditions met. Both pre- and post-conditions are expressed as Boolean variables with for-all ($\forall$) and there-exists ($\exists$) quantifiers. The verification process aims to formulate the correctness problem as the verification of the relationship between the pre-conditions assumed to be true at the beginning of the program execution and the post-conditions that must be true at the end of execution [4].

The advantage of the theorem proving approach over model checking is that the defined constraints are on states rather than instances of states. Hence a theorem proving technique can reason about an infinite state space without exhaustive searching. The draw back of this method is the required effort and expertise in proper expression of predicates for pre- and post-conditions and in deciding the validity of a theorem. In practice the output needs to be verified by a human [4]. This to some extent defeats the purpose of using an automated tool.

2.2.3 Static Analysis

Static Analysis is a semi-formal method in which the source code of a program is analysed to retrieve information that can be used for purposes such as optimisation or security flaw discovery [4]. The static analysis focus is on particular issues of the programming language that could lead to security flaws, for instance uninitialised variables, dereferencing uninitialised pointers, or leaving out the allocated memory. Wagner et al. [8] propose a static analysis to detect buffer overrun vulnerabilities by formulating the string operation as an "integer constraint" problem and using graph theory to solve the constraints. In a similar approach Ringenburg and Grossman [9] combine a whitelist with static analysis to compensate for the precision versus security trade-off. Yet another example is the work of Chen and Wagner [10] where the goal is to eliminate format string vulnerability in Debian Linux, taking advantage of type qualifiers in their static analysis. Avots et al. [11] have developed a pointer analysis for the C programming language based on a points-to algorithm using a binary decision diagram representation which could decrease the overhead of a dynamic string-buffer bounds checker. A shortcoming of this approach is that it is generally designed for a specific vulnerability. There is a trade-off between the precision and security which results in the generation of false positives and may require the human interaction in the verification process.

Breuer and Pickin [12] propose *Symbolic Approximation*, which is another semi-formal static analysis, as a customisable program logic for C which can be used to assign an approximation of semantics to programs written in this language. This is done by sweeping the code and constructing logical predicates with conditional variables where changes in their values depend upon the configured logic for each of the C statements. The code is then verified by evaluating all of the reachable predicates from each point of the code by propagating the point using the four principal components of compositional program logic, namely: *normal*, *return*, *break*, and *goto*. This is achieved by defining three conditional variables for three phases of the execution of a program fragment: *initial*, *during*, and *final* using a method that

is quite similar to the approach originally proposed by Hoare [13] which attempted to provide a basis for the logic of computer programming. A single precondition/triple post-condition format is proposed to specify the program logic of C and objective functions. The triple post-condition specifies the conditions under normal, return, and break exits of a program fragment. The analyser then verifies the code to determine if the propagation of a specified initial precondition would violate any of the rules defined as objective functions by developing the corresponding post-conditions [12]. The format also includes pre- and post-contexts which specify the context and output of a prevailing *goto* respectively, verifying whether a label is reached by execution of a *goto* literal or by sequential execution that traverses the associated label for the given *goto*. Breuer and Pickin [14] refer to their abstract interpretation of C code as a "configurable 3-phase Hoare-style logic" which can be used to analyse large scale code such as the kernel of a Linux operating system. They also provide an example for using this method to check for existence of deadlocks due to improper use of spinlocks which could lead to a denial of service attack. Breuer and Pickin [15, 16] further develop this approach, naming the approach *symbolic approximation* and introduce the use of symbolic machines constructed based on the proposed symbolic approximation.

2.3 Safe Programming Language

The goal in creation of safe programming languages is to remove the programming constructs that could lead to vulnerabilities. For instance the ability to use pointers to variables and the manipulation of pointers by the programmer could lead to incorrect address calculation and memory corruption. Removing this capability can remove a class of vulnerabilities in the programs produced under the safe programming language.

2.3.1 Type Safety

One of the major issues of flexible and powerful high level languages such as C is that it does not provide Type Safety. This has led to the proposal of *Type Qualifiers* and other type safety measures for high level languages. Foster et al. [17] propose a framework to extend a language, with a set of standard types, to a qualified type system where the rules that govern the language's standard types are followed. This is done by adding qualifier annotations that allow assignment of qualifiers to types and qualifier associations that identify what needs to be checked for qualified types. The qualifier inference rules then will check the invariants expressed by the programmer. Shankar et al. [18] use a type theoretic analysis technique based on the aforementioned framework [17] to detect format string vulnerability by assigning *tainted* qualifiers to variables containing user input and *untainted* qualifiers to internal and verified variables. They check when a tainted variable is sent to a function that expects untainted arguments. A similar technique was used by Johnson and Wagner [19] to find user and kernel pointer bugs in a Linux kernel using a type qualifier inference tool called CQual. Other examples of type qualifiers can be found in flow-sensitive [20], flow-insensitive [21] and semantic [22] type qualifiers.

In another approach, Necula et al. [23] describe their scheme of type safety for existing programs in C as the combination of type inference and run-time checking. Their type system, which is called CCured, keeps track of certain information about the memory area that is pointed to by each pointer. For dynamic pointers this information must be updated at run-time as the area pointed to by the pointer does not have a fixed type through out execution.

Any type safety defined in high level language may not be preserved after compilation to the low level assembly language which has resulted in the proposal of *Typed Assembly Languages* (TAL). Morrisett et al. [24] define a type system for a conventional assembly language that could maintain the type information through all compilation phases. Their compiler is designed to translate a polymorphic lambda calculus system to a TAL for a generic RISC instruction set. This strongly typed

TAL allows more control over certain operations for certain types such as arithmetic over pointers, dereferencing, or control transfer. To provide a more realistic example of their work, Morrisett et al. [25] present a TAL designed for the Intel IA32 architecture along with a C-like high level language called Popcorn. They show how high level language features can be compiled to a TAL assembly language. They define a variant of TAL for a stack that will preserve the type of anything pushed into the stack including the return address [26].

2.3.2 Proof Carrying Code

The concept of Proof Carrying Code (PCC) proposed by Necula [27] is defined as a framework to provide a general mechanism that would enable the receiver of a code fragment to verify that it can be executed safely on the receiver's machine. In this scheme it is suggested that the code fragment will carry a detailed and precise "explanation", i.e. proof, of why it satisfies the receiver's safety policy. It can be verified that the explanation is correct and it belongs to the code fragment [27]. Appel [28] argues that Necula's PCC is type-specific and proposes a scheme that would avoid any commitment to a particular type system. Instead the operational semantics of the machine code are defined in an expressive logic that can be used as a foundation for mathematics suitable for higher-order logic. Hamid et al. [29] propose a syntactic approach to Foundational PCC (FPCC) which avoids reasoning about the types in underlying typed machine by providing a typing derivation accompanied with the soundness proof of the type system rather than its semantic.

As mentioned before in FPCC a properly expressive logic is used to define both the concept of safety and the operational semantic of machine code without committing to any particular type system. The producer of the executable code must then provide a proof that the code satisfies the safety conditions using the foundational logic. This proof will accompany the executable code and is verifiable by the receiver of the code. Hamid et al. [29] use a Calculus of inductive Constructions (CiC) which is an extension of a higher order typed lambda calculus called the Calculus

of Constructions (CC) to define safety policies and proofs. Their defined machine model is quite powerful to model any stored-program computer system. Its modified variant is used by Abadi et al. [30] in their proposed Control Flow Integrity (CFI).

2.4 Exploitation Prevention

This section discusses the techniques that try to hinder the exploitation of memory errors. These techniques are generally ad-hoc solutions and lack a formal approach. Van der Veen et al. [31] provide a twenty-five year survey of literature regarding the attacks, countermeasures and statistics of memory errors.

The Morris worm, released in 1988, was the first widespread Internet worm. It was considered as a wakeup call for much needed security in design, implementation, and configuration of any system connected to distributed networks such as the Internet [31, 32]. The analysis of the worm revealed that it exploited a buffer overflow vulnerability in the `gets()` function of the `fingerd` BSD-Unix daemon and a misconfiguration in `sendmail` program that allowed the attacker to use the `DEBUG` command to execute other commands on the server [33]. The worm had the ability to copy multiple files compiled for different systems and could access the publicly readable password file for further analysis. The Computer Emergency Response Team Coordination Center (CERT/CC) was formed as a response to Morris worm [34].

2.4.1 Non-executable Stack or Data Memory

In a classic buffer overflow attack, the buffer located at the top of the stack of a vulnerable process is overflowed. The input to the vulnerable process that overflows the stack contains the attacker's shell code which is comprised of machine instructions. The overflowed buffer precisely overwrites the return address that is pushed by the calling function onto the top of the stack, so that it would point to the shell code. Once the called function executes the `ret` instruction, the control of the execution

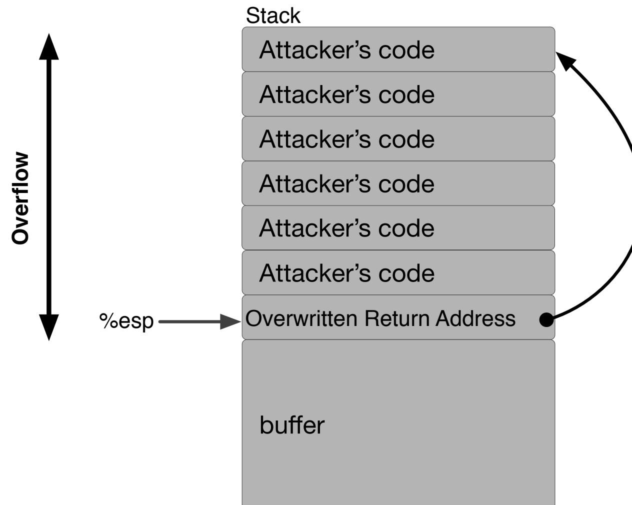

Figure 2.1: Buffer overflow attack with injected shell code

will be transferred to the attacker's shell code which generally creates a remote shell for the attacker to access the target system [35, 36]. Figure 2.1 shows the idea of a classic buffer overflow attack.

The non-executable stack, proposed by Solar Designer[1] [37], was the first security countermeasure against stack overflow [31]. Cowan et al. [38] proposed a scheme named StackGuard that inserts random bit patterns between the return address and the function arguments and variables. The random bit pattern referred to as a canary or cookie will be checked before return. This technique will detect an overwrite of the return address in case the overflow overwrites the canary as well. In some special cases the canary is predictable or a low cost brute-force technique, such as the scenario described in [39], can be used to guess the value of the canary. In such cases, it is possible to overwrite the canary with a correct value and bypass this protection mechanism. Under certain conditions an adversary may be able to use a pointer located after the buffer to change the return address without overwriting the canary [40]. Etoh[2] and Yoda[3] [41] proposed the rearrangement of local variables at compile time in such a way that buffers would be located after pointers to avoid pointer corruption by buffer overflow. The rearrangement of the variables is

[1] A hacker alias name.
[2] A hacker alias name.
[3] A hacker alias name.

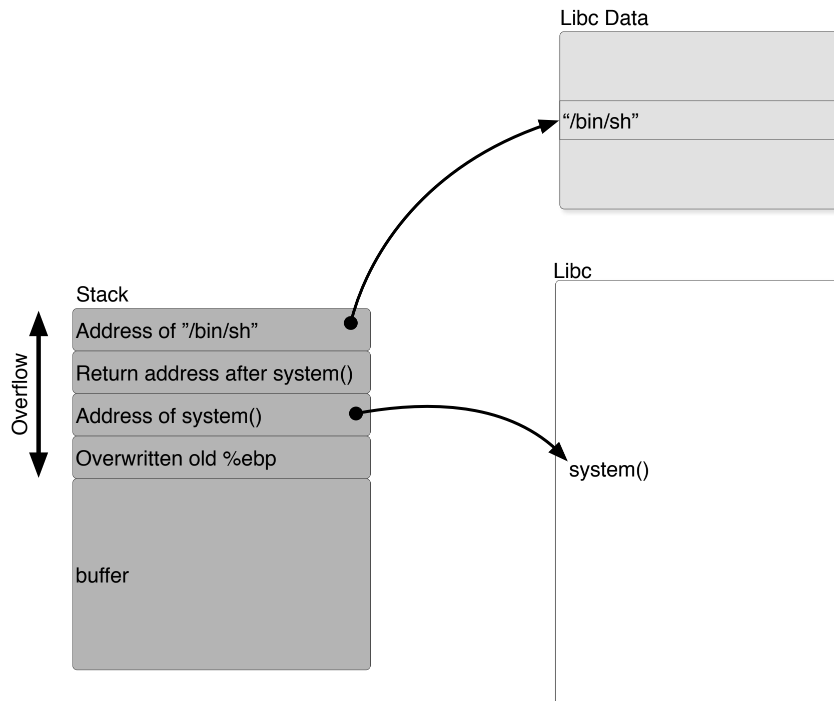

Figure 2.2: The general idea of return-into-libc exploitation technique

not always possible for instance for pointers inside structures, dynamic arrays, and functions that accept variable numbers of arguments [39].

Solar Designer [42] explains an exploit to bypass the non-executable stack. This exploit, which is referred to as return-into-libc, instead of injecting code as part of the user input that overflows the stack of the vulnerable process, overwrites the return address with the address of `system()` followed by the address of a `/bin/sh` string both within the C library. Upon the execution of the return instruction the control is transferred to the `system()` function and the address of the `/bin/sh` string will be treated as the argument for the function. This will create a shell for the attacker. Since no code is actually executed from the stack, the non-executable feature of the stack is bypassed. Figure 2.2 displays the idea of a return-into-libc exploitation technique.

In the C programming language, strings are terminated with a NULL which is an all-zero byte, hence the shell code or return-into-libc exploit which is fed as input to the vulnerable process cannot contain zero bytes anywhere in the middle of the input as it would terminate the string and the attacker's payload would be incomplete.

Solar Designer [42] proposed that the shared libraries be loaded to addresses that contain a zero-byte to prevent the return-into-libc exploit. This technique however was shown to be ineffective by Nergal[4] [43].

2.4.2 Address Space Layout Randomisation

The PaX security team [44] proposed that the non-executable feature should be generally applied to all data pages, and instead of mapping the shared library to addresses that contain zero, the entire address space of the process should be randomised. This approach that is referred to as Address Space Layout Randomisation (ASLR) uses random addresses to memory map the text, dynamic loader, and shared library for the executable code. It also randomises the addresses for allocated heap and stack. As the attacker needs the address of the injected shell code in a classic buffer overflow attack or the address of a function within the C library for a return-into-libc exploit, the ASLR technique makes it difficult to guess these addresses. Nergal [43] explains a more advanced return-into-libc exploit where arbitrary number of function calls can be chained together and the knowledge of the stack or shared library address is not required to succeed. The chaining multiple function calls allows the attacker to insert zero bytes whenever required by using one function call to insert zeros as argument for the next function call, hence overcoming the problem of having zeros as part of the overflow payload. Nergal [43] also points out that the ASLR technique used at the time had weaknesses such as:

- local exploits could retrieve the addresses from `/proc` file system;
- the random base addresses could be brute-forced, this is also discussed by Shacham et al. [45];
- the library and stack addresses could be leaked by exploiting format string vulnerabilities;

[4] A hacker alias name.

- functions that are not position-independent and cannot be memory-mapped randomly (e.g. `su`) can be targeted by the attacker;
- a function can be called using its Procedure Linkage Table (PLT) entry [46], as the process itself needs to find the shared library for instance the executable files with Executable and Linkable Format (ELF) in Linux use such mechanism;
- by passing appropriate arguments to the dynamic linker's `dl-resolve()` function the actual address of a function can be determined.

2.4.3 Return Oriented Programming and Defences

The non-executable attribute for data pages is widely adopted by most operating systems and supported by most hardware manufacturers [31]. This feature is also referred to as either $W \oplus X$ for mutually exclusive Write and eXecute access rights or simply as NX for Non-eXecutable. The $W \oplus X$ technique and Stack Smashing Protection which are added to the GNU C compiler and used by default make the classic code injection technique difficult, however there exist some special cases where certain attacks are possible [39].

One such case is the general exploitation technique of return-into-lib [43] which is further developed to return to a chunk of code rather than any library function [47]. A Turing-complete programming language is developed where the attacker can induce arbitrary behaviour by chaining short instruction sequences, referred to as *gadgets*, that are present in the target program's address space [48–50]. Each of these sequences end with a return instruction. This technique is called Return-Oriented Programming (ROP). Figure 2.3 shows the general idea of a ROP attack.

The return instruction is needed to take advantage of the stack of the vulnerable process. The overflow payload contains a combination of addresses for identified gadgets and any argument that may be needed for any of the gadgets. The gadgets are divided into five groups: load/store, arithmetic and logic, control flow, system

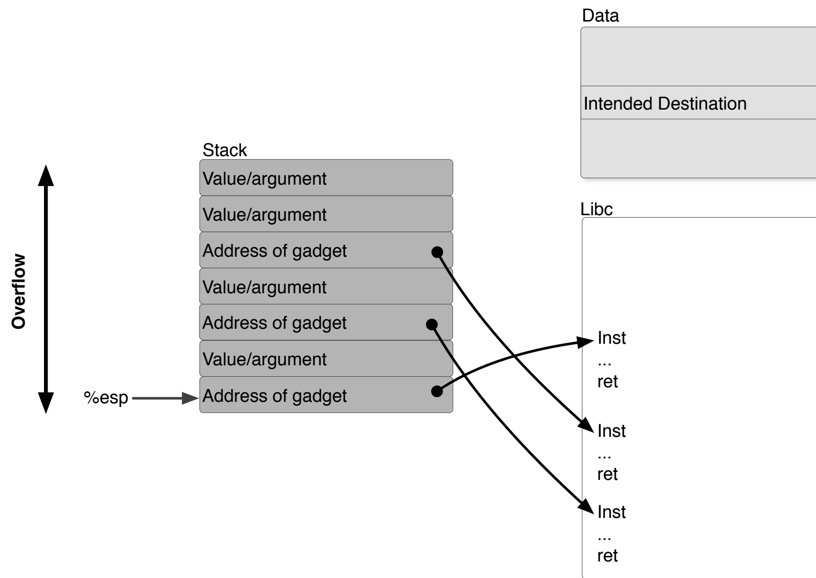

Figure 2.3: The general idea of Return Oriented Programming

calls, and function calls. Figure 2.4 shows an example for loading a value into the `%edx` register [50].

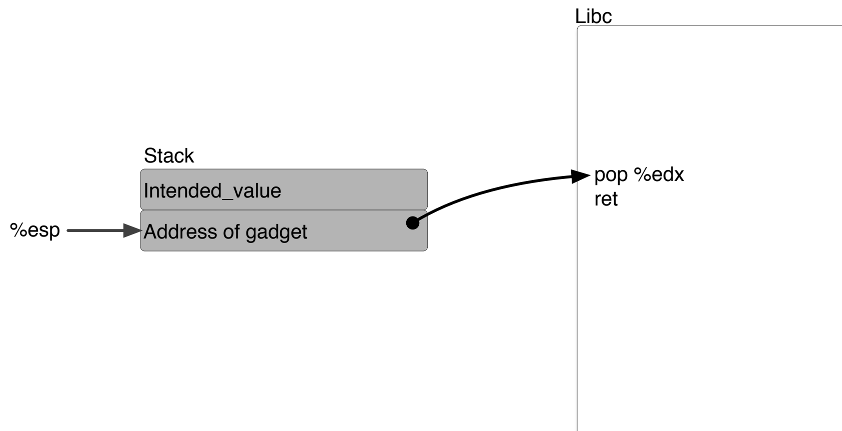

Figure 2.4: Loading a value into `%edx` register using ROP

Figure 2.5 shows a store example where the content of the `%eax` register is stored in memory. The first gadget pops the intended address of the destination minus 24 into the `%edx` register. The second gadget then stores the content of the `%eax` register at location `%edx+24` [50]. The choice of gadgets is based on their availability in the executable code and any of the shared libraries accessible by the executable code.

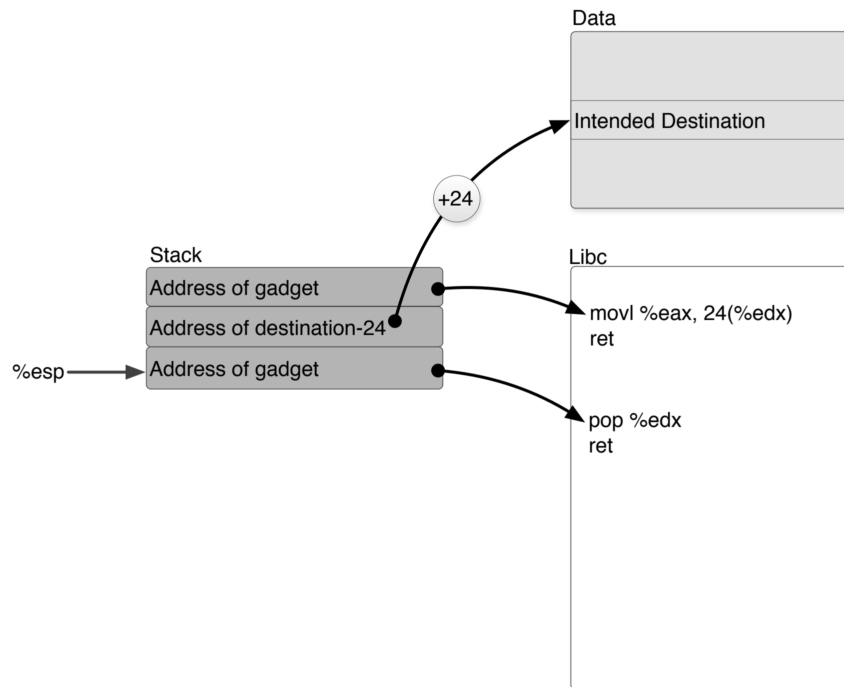

Figure 2.5: Storing the content of `%eax` into memory cell at `%edx+24` using ROP

Although ROP is introduced as a form of stack buffer overflow attack, it can be used with other memory errors. This is done by using the first gadget to set up the `%ebp` register, which in x86 architecture points to the stack frame, to point to an arbitrary memory location to launch the rest of the chains of gadgets [50]. This will be triggered by the execution of the `ret` instruction from the first gadget [50]. Among other memory errors that can be used in conjunction with ROP are heap overflow, integer overflow, and format string vulnerabilities. In an integer overflow attack the array bounds are manipulated by overflowing array indices, which could in turn result in heap or stack overflow [51, 52]. A format string vulnerability could allow the attacker to write fixed size arbitrary values anywhere in memory for instance updating the Global Offset Table (GOT) entry [53, 54]. The ROP exploitation technique has been shown to be effective on various architectures including x86, SPARC and ARM processors [50, 55]. Bletsch et al. [56] propose Jump-Oriented Programming (JOP) as another form of code reuse attack where the gadgets end with an indirect jump rather than return. The methods discussed in the remainder of this section aim to detect and/or prevent ROP attack or its variants.

Chen et al. [57] have designed a detection mechanism based on some of the characteristics of ROP attacks and perform a statistical analysis on ROP gadgets to determine two thresholds that they use to detect ROP: the gadget size and the sequence length. They perform a run-time analysis for each `ret` instruction to detect if any of the two thresholds is violated. Davi et al. [58] propose a run-time integrity monitoring scheme that verifies static measurements of the executable code at load time. Run-time checks are then added to the loaded code using code rewriting techniques that will monitor changes in the data segment particularly in stack to detect ROP attacks. In this mechanism the untrusted data is marked as tainted and a dynamic taint analysis makes sure that the taint data is not misused for instance as a pointer. A taint tracking mechanism counts the number of instructions between two returns based on the analysis that most of ROP gadgets have 3 to 5 instructions before `ret` instruction [58]. The aforementioned techniques however rely on modifiable characteristics of ROP attacks which can be changed by the adversary. Lu et al. [59] have shown that it would be possible to make ROP attacks, packed, using printable ASCII code and polymorphic.

Chen et al. [60] focus on three control flow instructions: `call`, `ret`, and `jmp`. They argue that by using a shadow stack the `ret` instruction that does not have a corresponding call entry can be identified. As the `call` instruction is often used to transfer the flow of execution to a function, it can be checked to see if the code at the called address matches one of the two possible function prologues: frame function, or non-frame function. Each of these two types start with a fixed set of instructions. For the `jmp` instruction, the direct jump is considered benign as the adversary cannot control the offset, but for the indirect jump their observation is that it is not used between different library functions and this feature can be used to identify a ROP attack [60].

Onarlioglu et al. [61] have developed a scheme to remove mechanisms that are necessary to chain gadgets from the executable code. Their focus is also on `call`, `ret`, and indirect `jmp` instructions. To make sure that a function is called using

its start address, a cookie that is based on a run-time encryption key and a pre-determined function identifier is pushed on top of the stack. Every `call` or `jmp` instruction is preceded by a code which verifies the cookie for the called function. This is done using code-rewrites. In the x86 architecture it is possible for the opcode of one instruction to appear as part of the opcode of another instruction due to the complexity of the architecture. To avoid the use of such unintended instructions in the x86 architecture, they propose a technique for aligned execution of free-branch instructions [61]. This is achieved by inserting an alignment sled e.g. using the `nop` instruction before the instruction that contains for instance an unaligned `ret`. This will enforce aligned execution of this instruction whenever the execution flow reaches the alignment sled [61]. Both of these techniques require code rewriting and as previous methods lack formal approaches.

Davi et al. [62] propose a scheme to incorporate a ROP detection and prevention mechanism, referred to as ROPDefender, into a Just-In-Time (JIT) compiler-based instrumentation that is called a Dynamic Binary Instrumentation (DBI) framework. This is done using a code cache and a virtual machine that contains the JIT-compiler. The program is executed under DBI control which will follow the ROPdefender return address check enforcement. The ROPdefender uses several shadow stacks in a multi-thread environment. This technique is based on a specific feature of ROP attack namely the use of a `ret` instruction at the end of each gadget and cannot detect similar code-reuse attacks such as JOP [56, 62]

Another form of ROP is the Return Oriented Rootkits which target the kernel of operating systems for code-reuse attacks. Li et al. [63] have developed a scheme to remove the `ret` instruction from the kernel of the FreeBSD operating system. This includes the unintended appearance of `ret` opcodes within other instruction opcodes as mentioned before due to the complexity of x86 architecture. They also propose a return indirection technique that instead of storing the return address, an index to a table is pushed on top of the stack. Each index points to an entry on a centrally managed table that contains the return address for its corresponding call.

This technique makes it difficult for the adversary to choose arbitrary gadgets and use their addresses as return points. This however cannot protect against classic return-into-lib type of attack as the return index can still be overwritten with a different index [63].

Bletch et al. [64] propose a technique that checks the control flow transfer after it takes place to detect any violation. This is achieved by adding a small snippet of lock code before each instruction capable of indirect transfer of flow of execution. The valid destination of the indirect transfer correspondingly has the unlock code. A violation is detected when a control transfer tries to lock an already locked value in memory. This results in termination of the execution. In this technique called Control Flow Locking the adversary may succeed at most once.

Pappas et al. [65] have developed a mechanism to randomise the executable code in-place in order to hinder the adversary's attempt to use some of the identified gadgets. They use several code transformation techniques including atomic instruction substitution, instruction reordering, and register reassignment. According to their experimental results their approach prevents some of the existing ROP attacks on known vulnerable software.

Tran et al. [66] have shown that return-into-lib(c)[5] attacks can be used for their side-effects despite the flexibility or expressiveness of such attacks being questioned in the literature [50, 63]. In this technique, POSIX standard functions are used to construct Turing-complete return-into-lib(c) code reuse attacks capable of performing arbitrary operations solely using the side-effects of these functions. That is instead of using the straight forward features of library functions the focus is on performing the normal operations, i.e. arithmetic/logic, memory access, branching, and system calls, using the effects of certain functions on memory and registers. For instance to perform arithmetic operations they take advantage of `wordexp()` function along with `itoa()` and `atoi()` functions as the `wordexp()` function only works with strings. On a Windows platform they use the `OffsetRect()` function designed

[5]Referring to either return-into-libc or more generic attacks as return-into-lib.

to move a rectangular shape within the screen to perform addition and subtraction [66]. The use of standard functions allows for developing cross-platform attacks while identified ROP gadgets would differ between different operating systems.

2.4.4 Heap-based Attacks

While the stack-based buffer overflow has received more attention in both exploitation techniques and countermeasures, the heap-based buffer overflows are no less important. The heap overflow vulnerability has been around for almost as long as stack-based overflows, however due to the availability and ease of use of the latter they have been less popular. Conover and the `w00w00` security team [67] describe a heap-based buffer overflow that could lead to overwriting a function pointer. In [68] heap overflow techniques for System V and GNU C Library implementations are described where the attacker is capable of overwriting the management information of the adjacent block which could lead to an arbitrary memory write using the unlink technique [69]. Another heap overflow technique takes the advantage of doubly linked lists in managing memory chunks. The attacker crafts a fake chunk with corrupted forward and backward pointers, and tricks the `dlmalloc()` function into processing it [70]. The forward pointer points to the address of a function pointer (e.g. an entry on GOT) minus 12 and the backward pointer points to the attacker's shell code. When the `unlink()` macro tries to adjust the freed fake chunk it overwrites the function pointer with the address of the attacker's shell code. The vulnerabilities used in aforementioned techniques were patched in 2004, but in an article published under the alias of Phantasmal Phantasmagoria six other techniques were discussed that could theoretically lead to exploitation [71]. The practical proof of concept for these techniques were published in 2009 under the alias of Blackngel [72]. The heap overflow exploitation has become more difficult, due to the $W \oplus X$ memory page attribute and ASLR technique which requires a memory leak exploitation for a successful code execution. This issue has been discussed in more recent articles and is specified as the reason why the underground hacker community

is reluctant in sharing these techniques publicly [73–75]. A heap overflow exploitation technique is described in [73] for `jemalloc()`, a user space memory allocator that is used in Mozilla Firefox and FreeBSD as well as a standalone version. Other examples of the technique are shown as a remote heap overflow of the Microsoft IIS 7.5 [74] and the VLC media player on FreeBSD [75].

One technique that is used to counter ASLR is a heap spray attack where the adversary allocates either a large number of small memory chunks or a sufficient number of large chunks filled with the shell code [76]. The goal is to make the address of the shell code predictable. Each piece of shell code is wrapped in a large `nop` sled to make transfer of control easier. This technique is mainly used in malicious PDF documents [77] and in various web browsers using JavaScript [78]. This technique can also use ROP to bypass the non-executable data page protection [77]. Several detection techniques are proposed which mainly focus on the characteristic of the spraying technique [79–81] and more can be found in the literature.

2.4.5 Control Flow Integrity

Control Flow Integrity (CFI) proposed by Abadi et al. [82] is an exploitation prevention technique that focuses on the transfer of the flow of execution to prevent a malicious transition. This technique has created a new branch of research in the past decade where the most of the focus has been on providing a practical implementation of the original work without formal analysis. The research in this area can be divided into three broad categories: (i) Classic CFI; (ii) Coarse-Grained (CG) CFI; and (iii) Fine-Grained (FG) CFI. Each of the CG and FG methods can be divided into forward or backward if the protection is only afforded to one of the edges. Burow et al. [1] take a more in-depth analysis approach and provide more details and finer classification based on the precision and accuracy of the studied methods as well as the performance and security aspects. They have assigned scores to the surveyed methods which will be reported in the following subsections when these methods are discussed.

Classic CFI

Classic CFI provides a mechanism to enforce the static Control Flow Graph (CFG) of executable code at run-time [82]. Abadi et al. [82] modified a machine model developed by Hamid et al. [29] by adding a label instruction and dividing the memory into code and data where the non-executable condition holds for all data memory. The executable code is then instrumented and rewritten, so that each indirect jump is preceded with a series of instructions that verifies whether the calculated address is the intended destination. The main difference of this approach with other protection techniques is that they represent the proposed scheme formally and provide logical proofs of the correctness of the mechanism given their assumptions. This approach, however, has a major drawback that has made the implementation of the original work impractical. The issue is that the label instruction introduces rigidity to the architecture where flexibility was intended. To clarify this point the introduction of the indirect jump instruction in modern processors is to enable dynamic linking and shared libraries, whereas the use of the label instruction with unique labels prevents the use of shared dynamically linked libraries. Among their proposed solution is the use of multiple labels however the return is using the indirect jump instruction as well. Since all indirect jumps must have a calculated address that is preceded with a unique label using the label instruction, a return from a library function must check all of the potential destinations to return to within all executable codes that call that library function. Figure 2.6 shows the difficulty of implementing CFI with library functions.

Another solution is the use of code duplication, in which case much better security is achieved at the expense of additional memory. The in-place code duplication removes the need for an indirect jump instruction as well as the use of dynamic linking, shared libraries, and even function calls, in which case the indirect jump instruction can be eliminated from the architecture altogether. The code duplication with the assumption of protected code memory can be proven secure without CFI.

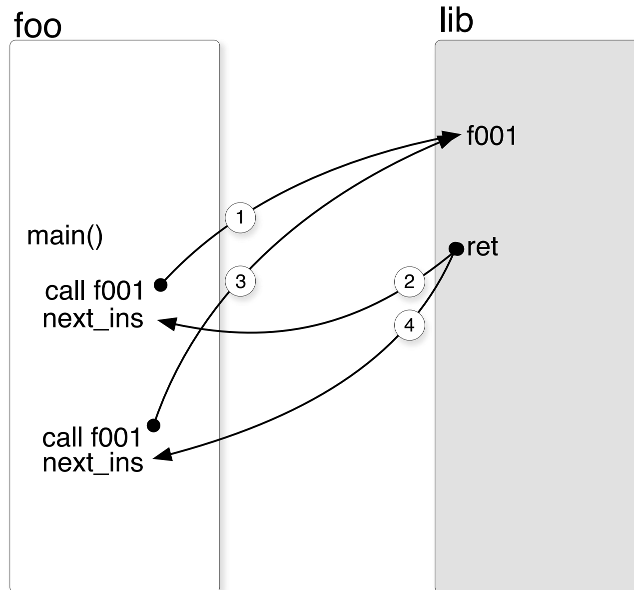

Figure 2.6: Difficulty of implementing CFI with library functions

To address the problem of multiple call sites and return, Abadi et al. [30] suggest the use of equivalent classes of IDs. That is indirect jumps from multiple sources to the same destination would receive the same ID (e.g. vertices 1 and 3, and vertices 2 and 4 in Figure 2.6). This technique however would be more permissive than the actual execution flow of the program. In their original work it is stated that the dynamic library is excluded for simplicity [82]. In fact the code instrumentation for ID checks with equivalent classes will pose a problem in linking dynamically with shared libraries. This is due to the fact that functions could be called from different executable code. Hence the class of return ID could have members of different executable code. For instance vertices 3 and 6 in Figure 2.7 belong to the same equivalent class, and will be assigned the same ID which could enable the attacker to return to the middle of the function `f001` from the call made in the `foo` executable. To counter this a shadow call stack is used to securely associate the return of a function call to its most recent call site [30]. Use of an equivalent class violates the condition used to prove the correctness of the approach namely the uniqueness of the labels. The unique label distinguishes the edge that is part of the CFG of the program from any other destination. When it is replaced with a

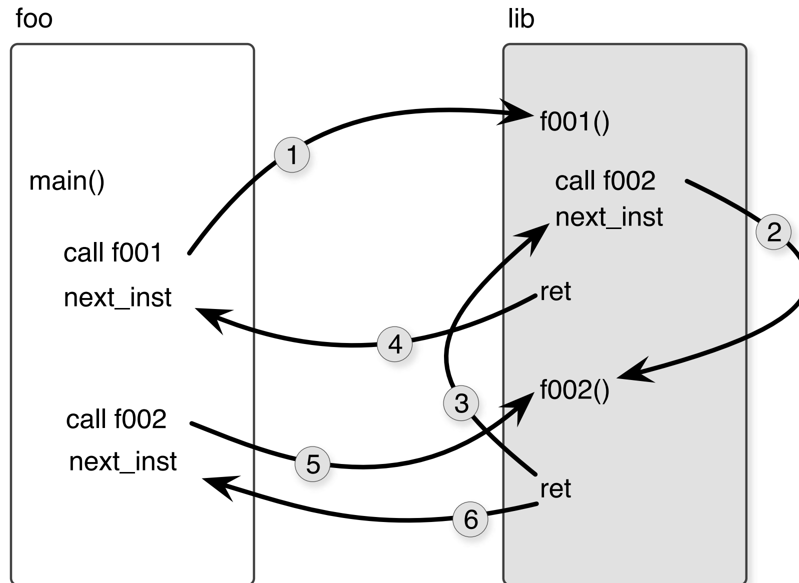

Figure 2.7: Difficulty of implementing CFI with library functions in equivalent classes technique

class of addresses it no longer provides the security guarantees that the execution will follow the intended path.

In their later work [30] shadow call stack is used for returns but is not proven to be equivalent with unique labels and is not part of the formal work. In the classic CFI, forward and backward edges are treated the same. Both use an indirect jump [82], whereas in all implementations that provide protection for both forward and backward edges returns are treated differently.

Another issue of the classic CFI is that the in-line policy enforcement does not address the problem of ROP gadgets in architectures with variable length instruction sets where the opcodes of indirect jump instructions could occur within other instructions in the code memory since these invalid jumps cannot be protected. One thing to keep in mind is that all of the proposed work that can be categorised as exploitation prevention is under the vulnerability assumption. That is the system or code that is being analysed is assumed vulnerable. It is not known in advance the exact technique that will be used to exploit that vulnerability. Hence the proposed protection must also provide adequate measures to protect against the assumption that the adversary can find a way to use an address which points to the middle of

an instruction through a new form of attack. In other words it must be formally proven that in the proposed solution it is not possible to jump to the middle of an instruction regardless of the used vulnerability or exploitation technique.

2.4.6 Coarse-Grained CFI

The second category of CFI is Coarse-Grained where the security is sacrificed for better performance. In this category valid destinations are divided into two or three equivalent classes and the indirect jumps can only jump to their identified class.

Bletch et al. [64] propose a method to protect against abuse of three types of control flow operations. The first type is identified as *unintended code* which can be defined as the opcode of instructions such as `ret`, `call` and `jmp` that appear within other instructions' opcodes. The second type is the `ret` instruction and the third type the `call` and `jmp` instructions. To protect against the first type they proposed the use of Software Fault Isolation (SFI) [83] techniques by aligning all variable-size instructions. This is achieved through changes at assembly level by preventing any instruction exceeding a predefined number of bytes in length. Then all indirect jump instructions are restricted to target only the aligned addresses. All the target addresses are also aligned which includes the instruction after a function call, first instruction of a function etc. To protect against the other two types they propose the use of a technique called Control Flow Locking. In this technique a mutex-like control flow key is used that will be locked at the call site at the time of the call; and unlocked at the target of the indirect transfer of control flow. The method has the capability to increase the precision of equivalent classes by using a multi-bit key, however in proposed implementation four equivalent classes are defined enforcing a policy based on static analysis of the source code [64].

Davi et al. [84] discuss their approach in preventing control flow attacks in smartphones. They face various challenges due to the differences between smartphone architectures and operating systems with Personal Computer (PC) systems. In this method a static analysis of the binary code is performed to generate the CFG of

the executable code. A list of valid target addresses are generated using heuristics, based on the indirect jump instruction used, and is saved as part of the generated CFG. At run-time the calculated address is checked against the saved values in the CFG to verify whether the current target address is among the list of valid addresses.

Niu and Tan [85] use a variable size alignment method which requires the use of an additional table for each executable code to record the beginning of the aligned chunks of code. The indirect transfer of the control flow is restricted to the beginning of the aligned chunks, by instrumenting the executable code to verify the target address using the chunk table before executing the indirect jump/call instruction. The use of the indirect jump/call instruction is prohibited for any target address within the same chunk. Only direct jump can be used to transfer the flow of execution to a target address within the same aligned chunk of code. In this method all aligned chunks of the code for an executable that are recorded in the same table can be considered as one equivalent class, as the verification allows any of the recorded addresses to be a valid target for any of the indirect transfer of control instructions.

Zhang et al. [86] have developed a Compact Control Flow Integrity and Randomisation (CCFIR) that only requires the binary code of the executable. In this scheme relaxed rules of CFI are enforced by redirecting all of the indirect control transfers to a dedicated section called Springboard. Springboard has a special structure for three different types of indirect transfer of control: call/jump to a function, return from a sensitive function (e.g. `system()` in `libc`), and return from a normal function. They use the equivalent classes in their scheme and assign a different ID for each type. Each type has a code stub with a special address on Springboard. SFI [83] techniques are used to protect access to the entries in the Springboard [86]. Zhang and Sekar [87] propose a coarser CFI by using two IDs: one for function calls and one for jumps and returns (replacing return with jump instruction).

Criswell et al. [88] devise a control flow integrity method for an operating system, by using a Secure Virtual Architecture (SVA) and instrumenting the code of the operating system with a virtual instruction set of the proposed SVA. The run-time

checks are then inserted when translating from the virtual instruction set to the target processor's native instructions. This method is coarse-grained as all functions are assigned to the same equivalent class and it is possible to return to a different call site from a called function. They also provide a partial formal proof of their proposed method in their SVA virtual machine model [88].

A forward-edge approach proposed by Gawlik and Holz [89], to protect `C++` virtual tables, performs a static analysis of the executable code to identify instructions that load a virtual table. Various policies then can be enforced using heuristics or type reconstruction of object-oriented code where the latter case is deemed impractical [89]. The heuristic is that a virtual table is stored in non-writable pages and contains virtual function pointers that point to read-only executable pages. The verification process checks whether the aforementioned conditions are met for identified virtual table dispatchers.

Another coarse-grained forward edge approach is proposed by Zhang et al. [90] to protect the `C++` virtual tables. In this method the virtual table is checked to be read-only and the binary code is rewritten to verify assigned IDs to virtual tables in virtual function dispatches.

Yuan et al. [91] propose a hardware-assisted method to detect and prevent violation of the flow of control of the executable code. In this method the Last Branch Register (LBR) and Performance Monitoring Unit (PMU) are used together to verify whether the target of an indirect jump is a valid destination or not. A bitmap table is created for valid jump-from addresses for indirect transfer of control instructions which are then mapped to valid target addresses using a hash table. Another bitmap table is used to identify valid entry points for each module. A static analysis is performed on the executable code to identify the call sites and the target of the indirect calls. If a function pointer is used the target is then classified based on the type of the return value. All functions that return a value of the same type are then classified into the same equivalent class. The verification is performed as part of the interrupt service routine of LBR + PMU hardware interrupts.

Mohan et al. [92] propose a technique to combine coarse-grained CFI with fine-grained artificial diversification. In this technique some of the indirect control transfer instructions are protected using a bounds checking technique. The bounds for each of the protected indirect control transfers are stored in a table whose base address is protected against information leakage attacks. A trade-off is made between the accuracy of the enforced coarse-grained CFI policy and the performance, but the approach has the potential to increase the precision to fine-grained CFI with higher efficiency penalties. The introduction of Intel Memory Protection eXtension (MPX) instructions will increase the efficiency of this technique. The goal of this approach, similar to other coarse-grained CFI, is to reduce the number of available ROP gadgets to the attacker. This depends on the selected bounds and the unavailability of the bounds table to the attacker.

All of the methods discussed above receive the same score for the forward-edge protection in the survey of Burow et al. [1], with the exception of the work of Zhang et al. [90] which receives a lower score for precision. The given score is classified as Static Analysis Precision Forward Edge (SAP.F.) score of 2 representing class-hierarchy analysis (Table 2.1).

Table 2.1: Static Analysis Precision for Forward edge in Burow et al. survey [1]

SAP.F. Score	Precision
0	No forward branch validation
1a	ad-hoc algorithms and heuristics
1b	context- and flow-insensitive analysis
1c	labelling equivalence classes
2	class-hierarchy analysis
3	rapid-type analysis
4a	flow-sensitive analysis
4b	context-sensitive analysis
6	dynamic analysis (optimistic)

Goktas et al. [93] have shown that coarser CFI approaches are vulnerable to attacks. They take advantage of the equivalent classes and have identified two types of gadgets that are still available to the adversary under CFI rules. Although the length of the gadgets are longer and this would decrease the flexibility in chaining

the gadgets, it is shown that the attack is still possible by either using the library functions for their side effects [66] or finding new gadgets that comply with the coarse CFI rules [93]. In their technique they succeeded in transforming a code-reuse attack to a code-injection attack, by calling an API function or making a system call to change the protection privileges on allocated memory to execute the injected code. Davi et al. [94] also analyse a range of CG CFI solutions and heuristic ROP attack detection techniques. They use a combined most restrictive CFI policy and show that an adversary that could only use one system library can achieve a Turing-complete gadget set. Carlini and Wagner [95] also discuss the inadequacy of heuristic techniques in detection of ROP attacks. It is shown that some of the CG CFI methods are completely insecure and others do not provide strong guarantees [96].

As mentioned earlier when the unique labels are replaced with classes of labels, the conditions of the formal work no longer hold. Hence the security of these schemes cannot logically be compared with the classic CFI as it does not satisfy the premises of classic CFI. However, a different property can be defined for this type of protection as a level of security to compare these types of techniques with a system that does not employ such techniques.

2.4.7 Fine-Grained CFI

The third category of CFI is Fine-Grained where the equivalent class of valid destinations is refined and not limited to two or three classes. The analysis of the executable code produces a more precise CFG. First the methods that have received the SAP.F. score of 4 as shown in Table 2.1 are discussed. This score represents a method that uses a flow- and or context-sensitive analysis of the executable code.

Wang and Jiang [97] use a technique to redirect indirect control transfer instructions to a secure target table containing valid destinations to provide control flow integrity for the code of a Hypervisor. If an adversary is able to perform arithmetic

operation with indices to the secure table it may be possible to change the flow of execution even with the added redirection.

Pewny and Holz [98] propose a compiler-based analysis of the source code of executables for iOS devices to determine valid targets for jump instructions. The policy is then enforced during compilation where binary rewriting will no longer be necessary. This will address the problem of encrypted or signed applications which is a common feature of iOS devices. First a list of allowed targets is generated for each indirect jump instruction and the basic block of the indirect jump instruction. The corresponding list of allowed targets is stored in a table referred to as Control Flow Wish (CFW). The CFG of the code is then generated based on the produced CFW. That is an edge between two basic block of the code in CFG will only be added if the target basic block is within the CFW of the source. To enforce the policy a piece of code is added before each jump that restricts the targets of that jump to the list of allowed targets in CFW.

SAFEDISPATCH can be categorised as a forward-edge fine-grained CFI but with a focused domain [99]. It provides protection for `C++` virtual tables where an attacker can use various techniques such as a heap-based dangling pointer to call a different virtual method than was intended. The virtual tables are a common feature in `C++` and are used for polymorphism. This approach is fine-grained as it uses the object class to verify the list of valid methods that can be invoked by an object. To identify the list of valid destinations it performs a variant static Class Hierarchy Analysis and then uses code instrumentation to add the verification code to the method calls of objects with various optimisations [99]. The domain of this protective measure by design is limited to the virtual tables of `C++` and is forward edge as it only protects against hijacking the virtual tables.

The work of Tice et al. [100] is conceptually similar to SAFEDISPATCH in that the goal is the protection of forward-edge indirect jumps in `C++`. Their approach differs from the previous work in regard to implementation aspect. The verification code is produced by the compiler rather than binary code instrumentation. The

advantage of this approach, as stated in [100], is that it does not restrict compiler optimisation, operation modes, and other features such as position independence and the exceptions of `C++` as well as limitation on the use of dynamically loaded libraries and ASLR. This method is classified as Fine-Grained since it introduces various imprecisions based on the type of target and applied analysis. The targets are comprised of virtual tables and indirect calls where different techniques are applied for each type. To address the issue of what is referred to as mixed code where the verification fails the control is transferred to a fail function which performs more in-depth analysis to verify the calculated address. A white-listed library technique is used in this scenario. This Forward-Edge FG CFI takes a better approach towards implementation of CFI in that it integrates the policy enforcement with production compilers, however it still relies on other techniques for protecting the return or backward-edge indirect transfer of control flow.

A hardware-assisted CFI policy enforcement is proposed by Davi et al. [101]. This is achieved by defining four states in the program execution namely: normal, entry to function, exit from function, and CFI policy violation. In this approach the CFI policy enforcement is assisted in hardware by the introduction of two new instructions: CFIBR and CFIRET. The CFIBR is used to label the function calls and the CFIRET instruction will label the return from function. At the beginning each function a CFIBR instruction is inserted which will have a label as an immediate operand. The forward edge policy is enforced by allowing the indirect jumps only to a CFIBR instruction preventing any jumps to the middle of a function. The CFIBR upon execution will store the embedded label in secure storage which will indicate that the function has been entered. The CFIRET instruction will verify whether this return is from a currently executing function by checking the secure storage where the CFIBR instruction stored its label. This is done by emitting CFIRET after each indirect jump instruction with the same label as the CFIBR at the beginning of the called function. HAFIX provides a hardware-assisted FG CFI policy enforcement

for backward-edges [102] where the protection for forward edge is the same as the work in [101].

Binary hardening is a method proposed by Payer et al. [103] that enforces an FG CFI policy. In this approach the three types of indirect transfer of control, `jmp`, `call`, and `ret`, are protected by the following policies. The `call` instruction can only use an address at the beginning of a valid function which is defined for an object by analysing the symbol table of the executable code. The `jmp` instruction is allowed only to use an address within the object itself or an address of a valid function where the latter case is for tail call optimisations. The `ret` instruction will always transfer the flow of execution back to the caller by using a shadow stack. The implementation uses the Executable and Linkable Format (ELF) binaries in Ubuntu. In case the executable code is stripped of its symbol table, the policy enforcement falls back to a CG precision for calls and jumps within the executable code.

The classic or original CFI methods receive the same SAP.F. score of 4, as discussed for the aforementioned methods in Burow et al. survey [1]. The following methods receive higher scores for better precision in analysing the executable code and produced CFG.

Prakash et al. [104] discuss a technique to provide a more accurate CFI policy to protect virtual function calls for COTS `C++` binaries. Their approach does not require the source code, symbol, and debug information. It relies on standard `C++` compilation. The steps to generate the policy are as follows: (i) identify all indirect jumps as candidates for analysis; (ii) transform the x86 instructions to an intermediate language for data flow analysis; (iii) identify virtual function call sites using the performed data flow analysis; (iv) create template signatures based on `C++` standard Application Binary Interface (ABI) and scan the read-only section of the binary for matches; (v) enforce the policy based on identified call sites and `vtables` using dynamic code instrumentation.

Niu and Tan [105] developed a method that allows separate compilation which is necessary for dynamic linking. In order to generate a precise policy, code modules are augmented with auxiliary type information which is then used when generating the CFG of the C code. To enable dynamic linking, the Equivalence Class Numbers (ECN) for branches and targets are linked to IDs and kept in two separate tables. This removes the global uniqueness condition for ECNs which was required in classic CFI. The ability to update the tables at runtime allows loading libraries dynamically and adjusting the CFG accordingly. This is done using a transaction technique where there are two transaction types: check and update. The more frequent check transaction is a read-only operation; the update transaction can lock and change the tables. For each indirect branch instruction the check transaction is performed using code instrumentation. In another work Niu and Tan use the Modular CFI approach to protect against JIT spraying attacks [106]. Modular CFI is an FG CFI with dynamic linking capability for code in C which protects both forward and backward edges of indirect control transfers.

Cryptographically enforced CFI is a method proposed by Mashtizadeh et al. [107] where a Message Authentication Code (MAC) is generated for all control flow objects when the object is stored. The MAC is verified when it is loaded. At the beginning of the execution of the code a random value is generated as the MAC key which is stored in reserved registers. Four types of pointers are defined: a pointer to a function, a return address, a pointer to a method, a pointer in `vtable`. The pointer along with the type of the pointer is then encrypted using the Advanced Encryption Standard (AES) algorithm with a 128-bit key. To reserve the registers, changes are made to the ABI that assures the compiler will not leak the MAC key. Two virtual instructions that will be translated to machine specific code are added: `macptr` which generates the MAC for a pointer; `checkptr` which verifies the MAC for a pointer. The compiler then identifies the type of pointer as stated above for indirect transfer of the flow of execution and inserts the virtual instructions accordingly.

Van der Veen et al. [108] have developed PathArmor, an FG Context-Sensitive CFI that uses the Last Branch Record (LBR) registers, a feature of Intel processors, to efficiently monitor the flow of execution and a kernel module to verify calls to sensitive functions. The efficiency of their work relies on the LBR capability of the processor as well as pruning the CFG of the program which avoids path explosion in path analysis. This is however, at the cost of precision of the CFI policy under the assumption that the adversary will need one of the sensitive functions to achieve a meaningful attack. To further enhance the efficiency of their method, the verified path is cached which eliminates the on-demand static analysis for the second execution of the same path. The kernel module intercepts calls to sensitive system functions which then sends a request to the path analysis component of the PathArmor. Context-Sensitivity is achieved by analysing the last 16 branches compared to other approaches where the individual indirect jumps are considered to be valid or invalid. The path is verified by searching the pruned CFG of the executable, to find a path that contains the recorded edges in LBR registers with the exact same order using a Depth First Search algorithm. PathArmor does not track the flow of execution within libraries and cannot protect against exploiting the vulnerable library functions. This is due to the limitation on the number of the LBR registers that would cause path pollution and destruction of the program context when tracing the execution flow of library functions. PathArmor is a practical and efficient CFI method with a delayed policy enforcement that would prevent a successful exploitation if it involves a call to a well-defined set of sensitive functions. It nevertheless is a FG and Context-Sensitive CFI approach.

Niu and Tan [109] propose an FG CFI enforcement policy that uses previous CFI techniques to generate a static CFG which is used as an upper bound for the enforced CFG. The required edges of the enforced CFG are added lazily depending on the program input. A performance optimisation technique used in this method is called address activation where instead of adding edges to the enforced CFG, the target addresses are added to a white-list. This technique achieves a better performance

at the cost of the precision of the enforced policy. Another technique to increase the performance is using idempotent policy enforcement code instrumentation that will change to no operations after the first execution. This requires runtime writes to the code pages. To address the potential code injection attack, a sandbox is used where the virtual code pages are marked read only within the sandbox and writable outside, both pointing to the same physical page. The backward-edge protection is also less precise than the shadow stack by allowing the function to return to all call sites that have called this function at the time of return. Their experimental work shows that even though the enforced CFG theoretically could grow to its upper bound, in practice it rarely grows to more than 30% of the static CFG. Although they argue that the adversary faces the program coverage problem, which is considered a hard problem, it seems the adversary could use a constraint problem solving technique to focus on the paths that could potentially achieve the intended results and reduce the number of paths that need to be checked. The per-input CFI builds on top of the Modular CFI approach with performance enhancing techniques that relax some of the forward and backward edge control policies. It requires complex runtime code rewrites.

To protect the kernel of the operating system Ge et al. [110] have developed a CFI policy enforcement for kernel software and have implemented the proposed method for the FreeBSD and MINIX operating systems. In this method a set of valid function addresses is generated for each kernel function pointer given the fact that the function pointers are used in a restricted way in the kernel. To determine the valid list of target addresses, a static taint analysis is performed. Since the operations on function pointers in the kernel is limited to assignment and dereference, two constraints are defined for pointers. If the pointer is a function pointer then the only allowed operation is an assignment. The second constraint is that no other pointer besides a function pointer can point to a function. The taint analysis then will report any violation of the described constraints. In another step the call sites are statically mapped to their corresponding return instructions by adding the call

site to the set of valid return addresses for that return. This approach is an FG CFI, as for both forward and backward edges a list of valid targets is generated which is more accurate than having one or two equivalent class for labels. This, however, introduces imprecision compared to a method that allows exactly one valid target for an indirect transfer of the flow of execution.

2.4.8 Attacks on CFI

Carlini et al. [96] assess the effectiveness of various practical implementations of CFI. They show that Coarse-Grained CFI is broken and the AIR[6] metric is not an effective measure of the security of a CFI scheme as it fails to capture two important properties: (i) the maximum reachable targets of a branch instruction; and (ii) the importance of the reachable targets. They propose a Basic Exploitation Test (BET) to rule out the broken schemes. If an enforced CFI fails the BET it immediately shows that the scheme is insecure whereas passing the test does not provide any security guarantees. They also propose attacks on a defined fully-precise static CFI and discuss the effectiveness of such a scheme with and without a shadow stack. Based on their analysis an attacker with partial control over memory can perform Turing-complete computation if a dispatcher function can be reached using non-control data attacks. The attacker can bend the enforced fully-precise CFI policy within the valid CFG to reach powerful library functions using the dispatcher function even with the shadow stack. The CG CFI schemes are shown to be broken, whereas a fully-precise static CFI provides a certain level of security when using the shadow stack. This cannot prevent against non-control data only attacks where the adversary can bend the rules of the enforced policy without violation.

Hu et al. [111] have proposed a systematic approach to the generation of data-oriented exploits that will follow the enforced CFI rules and avoid invalid memory accesses. The attacks could result in information disclosure which could be used to

[6] Average Indirect target Reduction.

retrieve sensitive data such as passwords and cryptographic keys, or privilege escalation which can be used to alter system call parameters or configuration settings. The data-flow stitching technique allows the adversary to search for data-flow paths that can exploit memory errors in the vulnerable software to launch attacks that do not hijack the flow of execution, do not violate the DEP, and can bypass ASLR protective measure. The technique uses a two dimensional data-flow graph based on memory addresses of variables and the execution time. To optimise the search scope, the path constraints are modelled using symbolic execution. The feasibility of an exploit is verified using a SMT[7] solver. Their prototype has successfully found data-flow paths to exploit various applications to leak sensitive information or privilege escalation or both. A data-flow exploit does not violate any of the protective measures enforced to protect against code injection or code reuse attacks, including an ideal CFI policy that protects the flow of execution.

2.4.9 Proposed Method

To give the reader an idea where the thesis method fits in the current state of the research, it is briefly discussed here. The method is similar to the classic CFI in that a formal approach is taken to express the problem and prove the correctness of the proposed solution. In this regard, as no complete implementation is proposed, it is different from all CG and FG methods discussed in this chapter. It differs from the classic CFI as it addresses the problem that makes the classic CFI impractical to implement, namely dynamic linking. Propositional Dynamic Logic (PDL) is used to express the execution of programs comprised of atomic instructions and express the consequences using logical predicates. The main problem of the Classic CFI method is that it introduces imprecision for any potential implementation. The approach addresses the precision problem of Classic CFI by expressing the required policy enforcements as part of atomic execution of each indirect jump instruction. The method allows the enforcement of the most precise control flow graph that is

[7]Satisfiability Modulo Theories.

possible in any CFI method without memory corruption prevention. Since different checks are needed for function calls (forward edge) and returns (backward edge), the indirect jump instruction is divided into two types for forward and backward edges of the flow of execution, and apply different checks according to the direction of the edge. The flexibility of the imposed controls allow the use of dynamic linking and shared libraries along with position independence. The result of the proposed solution is then expressed in two theorems with proofs.

2.5 Summary

Malicious code execution relies on two requirements: (i) a vulnerability within the target system; and (ii) an exploitation that uses the vulnerability to achieve a malicious intent. The trend of research idealistically aims at removing all possible vulnerabilities and produce error free systems both in hardware and software, but this has shown to be difficult in practice. This is due to the complexity of hardware and software as well as the gap between the semantics of high level programming languages and machine language.

Another approach is to either make a successful exploitation impossible despite the vulnerability or at least significantly reduce its chance. There has been many proposed solutions that address a particular problem in a very specific way, for instance the use of canaries in preventing stack buffer overflow. These types of solutions generally rely on certain assumptions which may not be true under different circumstances and are not clearly defined, making it difficult to reason about and evaluate their correctness. A better approach is to consider a formal method in which the problem, any assumption or precondition, and the proposed solution can be clearly and logically defined where the proof of soundness and correctness can be described and evaluated.

Chapter 3

A Formal Model of Ideal Control Flow Integrity

3.1 Introduction

In recent years a significant amount of work has been done to prevent or to mitigate the exploitation of a vulnerable machine. The focus of most of this trend of research has been on the transfer of the flow of execution by enforcing a policy that would make it impossible, impractical, or difficult for an adversary to successfully execute a crafted sequence of instructions which may or may not be part of the executable code of the vulnerable program. As discussed in the previous chapter all of the work in this area, with the exception of the original Control Flow Integrity (CFI) [82], focus on practical implementations rather than formal approaches that can provide provable and strong guarantees. Some of these works have been shown to be completely insecure. Others are hard to analyse formally. A formal approach provides the required foundation upon which implementations for various architectures, operating systems, and compilers can be realised.

In this chapter, a formal model is proposed to express the problem of exploitation and a solution with proof. The instruction types and their effect on the flow of execution are formally defined. It is proven that protecting two properties can

prevent control flow hijack attacks. The first property is the integrity of the block of code and the second property is the integrity of the flow of execution. To achieve this, the logic of the flow of execution and the effect of various instructions on change in the flow of execution is discussed in Section 3.2. To control the scope of the formal work without loss of generality, in Section 3.3, a machine model is defined that is capable of modelling modern processors. In Section 3.4 the attack model is defined to express the necessary conditions needed by the proposed CFI enforcement to protect against the defined attack. The focus is particularly on control data attacks that lead to a control flow hijack. In Section 3.5 the required protective measures to protect the integrity of the block of code as well as its flow of execution are described. The theorems that link all these concepts are presented in Section 3.6. The chapter concludes with Section 3.7.

3.2 Instruction Types and Flow of Execution

All machine instructions influence the execution path of an executable code. They can be divided into four types: sequential, conditional branch, direct jump, and indirect jump. For sequential instructions the Program Counter (PC) register is incremented by one. For direct jump instructions the destination is some address w embedded in the instruction. The change of the flow of execution for the conditional branch depends on a condition. When true, the flow directly jumps to an address w provided in the instruction; when false the PC is incremented by one. For indirect jump and return instructions the destination is an address provided from the content of a register. Other forms of change in flow of execution in more complex architectures may provide various versions of these types. These additional instructions provide more options such as specifying memory locations as operands and different sizes for immediate operands. Based on how an instruction changes the flow of execution, these instructions would fall into one of the aforementioned categories. To further clarify this point these categories of instructions are defined based on their effects on PC as follows:

- Sequential: $\{instruction |\ pc = pc + 1\}$;
- Direct Jump: $\{instruction |\ pc = w\}$ where w is an address given as an operand;
- Conditional Branch: $\{instruction |\ \text{if } condition \text{ then } pc = w \text{ else } pc = pc + 1\}$ where w is an address given as an operand;
- Indirect Jump: $\{instruction |\ pc = register\}$.

The flow of execution in high level languages is an abstract concept. The programming constructs provide more complex tasks to be performed with simpler syntaxes. Almost all high level programming languages provide constructs for conditional statements, loops, direct jumps, and function calls and returns. One important point about the flow of execution is that the functions in high level languages have well defined boundaries, a clear entry point, and one or more return point(s). The labels and direct jumps in languages that define such control structures are only allowed within the boundary of a defined function. That is the flow of execution cannot be changed to a label outside the boundary of a function. Although the compilers translate the high level language to correct equivalent code in machine language, the well defined function boundaries no longer exist in the equivalent code due to the flexibility of instructions such as indirect jumps which could allow paths that were not possible in the corresponding high level program. In this sense a function call in a high level language would be translated to a more permissive jump or call instruction with the memory address of the specified function as its parameter. It is more permissive as in a high level language the function call is restricted to the start of a defined function using a unique name, whereas a jump can be to any address within the address space of the program. For a program that does not rely on any external library at the end of compilation, all of the virtual addresses of the defined functions are known and their names can be replaced with their addresses. Local function calls can be performed with direct jump instructions, however calls

for dynamically linked library functions require indirect jump instructions where the address of a called function is determined at run time.

Before making the call to a function, at run-time, the return address must be recorded. The corresponding return instruction restores this recorded address to transfer the flow of execution back to the instruction after the call. The return instruction depends on information that is only available at run-time.

The formal approach is based on the types of the instructions and their effects on the execution path. To state the preconditions that are necessary to prevent certain types of attacks, a model capable of representing any stored program machine is used. Propositional Dynamic Logic (PDL) is used to formally express the machine model and the attacks, and to reason about the protective measures.

3.3 Machine Model

To make the propositions and the arguments easier to express, a simple but realistic machine model, which has been used previously in the literature for a similar purpose [29, 30], is used with improvements and in the context of PDL. The machine is comprised of a processor with a register file of 32 registers, a designated and separate register as Program Counter (PC), and byte-addressable random access memory. The state of the machine is considered as the content of memory, register file, and PC. The definition of words, memory cells, register files, and machine states are as follows:

$Word = \{0, 1\}^*$

$Mem = address \rightarrow Word$

$Regnum = \{0,\ 1,\ \ldots,\ 31\}$

$Regfile = Regnum \rightarrow Word$

$State = Mem \times Regfile \times PC$

The machine has a load-store architecture where no direct operation is performed on memory cells as operands except for `load` and `store` instructions. The machine

has six sequential instructions; one for each of direct jump, conditional branch, and indirect jump; and one instruction for return from a function call. The `halt` instruction is used to mark the end of an executable code, and is used in the model to stop the execution when a violation of the defined rules occurs. In implementation however these violations can be dealt with by redirecting the flow of execution to an exception handling code. As the machine model is a generic model so that it can be applied to various architectures, the memory is chosen as byte-addressable. The instructions can be of variable length. This will allow the model to be applied to complex architectures where the destination address for indirect jump instruction could be to the middle of the opcode of an instruction. The decoding function represents the notion of decoding the instruction i in machine language to its semantic and is defined as follows.

Definition 3.3.1. $Decode(i) : \{0,1\}^* \rightarrow A \cup \{illegal\}$

The *Decode* function can identify the length of an instruction. For sequential instructions $pc + 1$ represents the notion of the calculated address of the next instruction based on the length of the current instruction. Table 3.1 provides the summary of the notation used to express the semantic of the machine instructions. The instruction set is shown in Definition. 3.3.2.

Definition 3.3.2. $A \stackrel{def}{=} nop \mid add\, r_d, r_s, r_t \mid addi\, r_d, r_s, w \mid movi\, r_d, w \mid ld\, r_d, r_s(w) \mid st\, r_d(w), r_s \mid bgt\, r_s, r_t, w \mid jd\, w \mid jmp\, r_s \mid ret\, r_s \mid halt$

This abstract machine with arithmetic, load/store, and conditional branch instructions and Random Access Memory (RAM) can model any stored program computer system. The equivalence of RAM models and Turing machines are discussed in detail by Aho, Hopcroft, and Ullman [112].

3.3.1 Notations

In this section various notations that are used throughout this chapter are explained. The state of the machine is comprised of the content of the memory, register file,

and PC, however each instruction has a limited effect on the state. In other words, it only makes a small and specific change in a memory cell, register, and PC. The notations $Reg(r_s)$ and $Mem(w)$ express the content of a register or a specific memory cell. They represent the content of register r_s and the content of memory at address w respectively. The operator "$\leftarrow$" expresses the assignment in "$target \leftarrow value$" where $target$ represents a register in a register file or PC, or a memory cell and the $value$ represents the value of that register or memory cell after the completion of the action using the given notation for each. Since PC is a separate register from the register file (and for brevity), to express the content of PC the notation pc is used rather than $Reg(pc)$. To specify a particular element in a state the "." operator is used. For instance $s.Reg(r_s)$ expresses the content of the register r_s in state s. To express the new state versus the previous state the prime notation is used for the state that immediately follows the current one. For instance, the notation $s'.pc$ expresses the content of PC in the state that follows the s state where both s and s' states are present in a given expression. When necessary to express multiple states the subscript notation is used, for instance the states $s_1, s_2, \ldots, s_k$. Table 3.1 provides a summary of the notation used to express the semantic of the machine instructions.

Table 3.1: Notation summary

Notation	Semantic
$\leftarrow$	Assignment as $target \leftarrow value$
$Mem(w)$	Content of memory at address w
Mem	No change in memory state
$Reg(r_x)$	Content of register r_x in register file
Reg	No change in the state of the register file
pc	Content of the program counter
$\in_{emb}$	Embedded as immediate operand as $w \in_{emb} i_x$
dot / .	Partial element of the state e.g. $s.pc$: content of pc in state s

Table 3.2: Machine instructions and semantics

Operation	Instruction Semantic	State Transition
no operation	*nop*	$(Mem, Reg, pc+1)$
add registers	$add\, r_d, r_s, r_t$	$(Mem, r_d \leftarrow Reg(r_s) + Reg(r_t), pc+1)$
add registers and words	$addi\, r_d, r_s, w$	$(Mem, r_d \leftarrow Reg(r_s) + w, pc+1)$
move a word into a register	$movi\, r_d, w$	$(Mem, r_d \leftarrow w, pc+1)$
load	$ld\, r_d, r_s(w)$	$(Mem, r_d \leftarrow Mem(Reg(r_s) + w), pc+1)$
store	$st\, r_d(w), r_s$	$(Mem(Reg(r_d) + w) \leftarrow Reg(r_s), Reg, pc+1)$
branch greater than	$bgt\, r_s, r_t, w$	(Mem, Reg, w) when $Reg(r_s) > Reg(r_t)$
		$(Mem, Reg, pc+1)$ when $Reg(r_s) \leq Reg(r_t)$
direct jump	$jd\, w$	(Mem, Reg, w)
indirect jump	$jmp\, r_s$	$(Mem, Reg, Reg(r_s))$
return	$ret\, r_s$	$(Mem, Reg, Reg(r_s))$
Halt	halt	Halt state

To show the similarity of the machine model with previous work in the literature, the semantic of the instructions as the state transition is expressed in the form of a 3-tuple representing the content of memory, registers and PC before and after the execution of instruction i respectively. For simplicity the state transition is expressed as changes within the elements of the tuple. When there is no change in the content of memory or register file it is expressed as Mem and Reg respectively. For instance the state transition for the instruction $add\, r_d, r_s, r_t$ in Table 3.2 is expressed as $(Mem, r_d \leftarrow Reg(r_s) + Reg(r_t), pc+1)$ where the first element represents no change to the content of the memory, the second element represents the changes to the content of register r_d in the register file using the assignment operation, and the last element represents the change to PC where only the value is expressed.

For the remainder of this chapter, however, the PDL notation is used to express the semantics of instructions.

3.3.2 Propositional Dynamic Logic

Propositional logic can be used to provide a formal foundation in defining and discussing attacks and corresponding protective measures for executable code. The

correctness of such measures can be proven using the rules of propositional logic. Modern processors have finite sets of instructions designed to perform a well-defined task with a specific effect on the machine state. It is quite reasonable to use a branch of logic that can capture and combine those two aspects. PDL is used to reason about the abstract machine model where the set of basic actions is the set of machine instructions. The set of propositions is about the state of the abstract machine. The language and notations of PDL are described in [113], and were originally proposed by Fischer and Ladner [114].

Definition 3.3.3. *Language of PDL:* Let p and a range over the set of basic propositions P and set of basic actions A respectively. Then the formulas φ and action statements α of propositional dynamic logic are given by the following Bakus Naur form (BNF):

$\varphi \stackrel{def}{=} \top \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \langle\alpha\rangle\varphi \mid [\alpha]\varphi$

$\alpha \stackrel{def}{=} a \mid ?\varphi \mid \alpha_1;\alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha^*$

As shown in the definition above, the language of PDL is comprised of two BNF forms, representing the logical formulas and the action statements. These formulas are linked together by a label transition system which is defined later in this section. The atomic action a, which ranges over the instruction set of the machine model ($Decode(a) \in A$), defines a binary relation as follows where $Decode(a)$ is the decoding function (Definition. 3.3.1).

Definition 3.3.4. $R_a = \{\rightarrow^a : (s, s') \subseteq S \times S | Decode(a) \in A\}$

The binary relation R_a in a given state s, specifies that there is a state s' which is the result of executing the instruction a in state s and $(s, s') \in R_a$. A program or an executable code α is comprised of well-formed atomic actions according to the rules of action statements in Definition. 3.3.3. To express various notions of an executable code construct, the following operators are used in PDL.

- *Sequence* is expressed by the operator ";". For the action symbols a and b the binary relations R_a and R_b are defined on S respectively. The action sequence

$a;b$ is then given by the following relation:

$R_a \circ R_b = \{(s, s')|\exists s_0 \in S : (s, s_0) \in R_a \wedge (s_0, s') \in R_b\}$.

- *Choice* is defined by the operator $\cup$. For the action symbols a and b the binary relations R_a and R_b are defined on S respectively. The action sequence $a \cup b$ is given by $R_a \cup R_b$.

- *Test* is expressed using the notation $?\varphi$ where φ is a formula. A test can be defined as a binary relation which is a subset of the identity relation on S. The action is defined by the following relation:

  $R_{?\varphi} = \{(s, s)|s \in S, s \models \varphi\}$.

- *Iteration* is represented by the operator "$*$". For the action symbol a binary relation R_a is defined on S. The action sequence a^* is given by the following relation:

  ${R_a}^* = I \cup R_a \cup {R_a}^2 \cup {R_a}^3 \cup \ldots$.

The expressive power of PDL allows a unified and simple representation of programming language constructs [113]. For instance an `if` statement of the form: $if\ \varphi\ then\ \alpha_1\ else\ \alpha_2$ where φ is a well-formed logical formula and α_1 and α_2 are well-formed action statements can be represented as: $(?\varphi; \alpha_1) \cup (?\neg\varphi; \alpha_2)$. A loop of the form: *while* φ *do* α can be represented in PDL as $(?\varphi; \alpha)^*; ?\neg\varphi$. These simplify the program equivalence or correctness analysis regardless of the programming language by transforming the program to its PDL representation or vice versa. The focus of the thesis, however, is on atomic actions and their effect on the transition of the flow of execution in very simple combinations of these atomic actions rather than the overall logic of the program.

The two modal operators $\Diamond$ (diamond) and $\Box$ (box) capture the notions of $\exists$ and $\forall$ in predicate logic respectively, with regard to the execution of actions and logical statements. That is the modal operators *diamond* and *box* represent the following for the action statement α and logical formula φ:

- $\langle\alpha\rangle\varphi$: some execution of α satisfies φ;

- $[\alpha]\varphi$: all executions of α satisfy φ.

A *Label Transition System* (LTS) brings together the notion of actions and the propositions about the effects of those actions on the state of the system [113]. To construct an LTS as defined in [113], the set of states ranges over the machine states comprised of the content of memory, register file, and PC, the set P is the set of propositions and the set A the set of machine instructions which will form the set of labels. The LTS is then defined as follows.

Definition 3.3.5. The triple $M = (S, R, V)$ is a label transition system over the propositions P and basic actions A such that:

- S is the set of machine states as (Mem, Reg, pc);
- $R_a = \{\rightarrow^a : (s, s') \subseteq S \times S | Decode(a) \in A\}$ (Definition 3.3.1) is a set of labelled transitions from state s to state s' after the execution of the instruction a;
- $V : S \rightarrow P(p)$ is a valuation function that determines the value of a proposition $p \in P$ in a state $s \in S$.

The PDL expression for each of the machine instructions (atomic actions $a \in A$) in the (s, s') transition (R_a) is shown in the Table 3.3. The operator "=" in the proposition is the logical equality operator that is true if the left and right values are equal and false otherwise.

Using the PDL expression, the types of instructions in the abstract model are defined, and will be used in categorisation of the necessary preconditions for each type.

Definition 3.3.6. *Instruction Types:* In the defined LTS where $M = (S, R, V)$, the following sets can be formally defined:

- *Sequential:* $SQ = \{i | Decode(i) \in A \wedge (s, s') \in R_i \wedge [i]s'.pc = s.pc + 1\}$;
- *Direct Jump:* $DJ = \{i | Decode(i) \in A \wedge (s, s') \in R_i \wedge [i]s'.pc = w \in_{emb} i\}$;

- *Conditional Branch:* $CB = \{i | Decode(i) \in A \wedge (s, s') \in R_i \wedge [i]s'.pc = w \in_{emb} i \vee s'.pc = s.pc + 1\}$;

- *Indirect Jump:* $IJ = \{i | Decode(i) \in A \wedge (s, s') \in R_i \wedge [i]s'.pc = s.Reg(r_s)\}$.

It can be stated that $A = SQ \cup DJ \cup CB \cup IJ \cup \{halt\}$ and $SQ \cap DJ \cap CB \cap IJ \cap \{halt\} = \emptyset$. Hence to reason about the properties of code execution the validity of the arguments for the four sets that impact the execution path can be verified and in this way all possible transfer of the flow of execution will be covered. The three sets DJ, CB, and IJ are specifically designed to change the flow of execution beyond the normal sequential flow whereas the set SQ contains all the other instructions. Calling library functions in dynamically linked executables and returns require indirect jump instructions as the target destination is unknown at compile-time and needs to be calculated at run-time.

The notion of a program as stored in memory must be distinguished from the program execution. The *finite computation sequence* of a program α as defined

Table 3.3: PDL expressions of machine instructions

Relation	PDL expression	Propositions
R_{nop}	$[nop]p_1$	$p_1 \equiv s'.pc = s.pc + 1$
R_{add}	$[add\, r_d, r_s, r_t]p_1 \wedge p_2$	$p_1 \equiv s'.Reg(r_d) = s.Reg(r_s) + s.Reg(r_t)$
		$p_2 \equiv s'.pc = s.pc + 1$
R_{addi}	$[addi\, r_d, r_s, w]p_1 \wedge p_2$	$p_1 \equiv s'.Reg(r_d) = s.Reg(r_s) + w$
		$p_2 \equiv s'.pc = s.pc + 1$
R_{movi}	$[movi\, r_d, w]p_1 \wedge p_2$	$p_1 \equiv s'.Reg(r_d) = w$
		$p_2 \equiv s'.pc = s.pc + 1$
R_{ld}	$[ld\, r_d, r_s(w)]p_1 \wedge p_2$	$p_1 \equiv s'.Reg(r_d) = s.Mem(s.Reg(r_s) + w)$
		$p_2 \equiv s'.pc = s.pc + 1$
R_{st}	$[st\, r_d(w), r_s]p_1 \wedge p_2$	$p_1 \equiv s'.Mem(s.Reg(r_d) + w) = s.Reg(r_s)$
		$p_2 \equiv s'.pc = s.pc + 1$
R_{jd}	$[jd\, w]p_1$	$p_1 \equiv s'.pc = w$
R_{bgt}	$bgt\, r_s, r_t, w \equiv$	
	$[(?\varphi; jd\, w) \cup (?\neg\varphi; nop)]p_1$	$p_1 \equiv s'.pc = w \vee s'.pc = s.pc + 1$
		$\varphi \equiv s.Reg(r_s) > s.Reg(r_t)$
R_{jmp}	$[jmp\, r_s]p_1$	$p_1 \equiv s'.pc = s.Reg(r_s)$
R_{ret}	$[ret\, r_s]p_1$	$p_1 \equiv s'.pc = s.Reg(r_s)$
R_{halt}	$[halt]p_1$	$p_1 \equiv \bot$

in [115] and denoted as $CS(\alpha)$ is the set of all possible sequences of atomic steps in the execution of the program α. Using this definition logical statements can be made about any partial sequence that would belong to this set. The term *computation sequence*, which is by itself a program in the context of PDL, is used to refer to the order of execution of instructions that are not necessarily located in contiguous memory locations. To refer to the program as stored, the term *memory sequence* is used. As a simple clarification of the goal here for instance a successful code injection attack would involve a memory sequence that does not belong to the memory sequence of the program α. For an exploitable program the execution of the program would allow the injection of the code and transfer of the flow of execution to the injected code. Since it is possible to execute the injected code, it also belongs to the possible computation sequences of the program regardless of whether it was anticipated by the programmer. When the sequence of atomic instructions of a program in execution is used in PDL expressions, it refers to all of its possible computation sequences. This is specially useful when logical statements are made about the execution of the attacker's intended computation sequence as part of the execution of an exploitable program.

3.4 Attack Model

An attack model describes the capability of an adversary and allows for clear assumptions which will further clarify the analysis of any proposed protective measure. Various models have been discussed in the literature which in general can be defined as arbitrary memory write access by the adversary. These attack models come with certain restrictions such as Data Execution Prevention (DEP, $W \oplus X$, or NX) [116] and separation of code and data memory. Depending on the proposed method there are additional memory spaces that are inaccessible to the adversary which are generally defined as part of the assumptions or premises of the work. A shadow stack [1, 30], and safe memory for code pointers [117] are examples of such memory

spaces. Another important aspect of defining an attack model which is directly connected to the preventive methods is the adversary's exploitation technique. That is how the adversary uses the described arbitrary memory write to exploit the target machine. From this aspect the exploitation can be broadly categorised as *Code Injection* and *Code Reuse* attacks where a standalone code injection under the DEP assumption is prevented. The term standalone is used as it may still be possible to combine a code reuse attack with code injection where code reuse is used to load and then execute the adversary's injected code. To demonstrate the link between the exploitation technique and the protective measure two different types of Control Flow Graph (CFG) are defined:

- A *fully precise dynamic CFG* of an executable is the ideal CFG which can be defined as the intended execution path of the code by the programmer under the benign execution. The full precision only allows one target for any change in the flow of execution and the dynamism allows the run-time calculation of the address.

- A *fine-grained dynamic CFG* of an executable is defined as the intended execution path of the code by the programmer, under the benign execution for specified function calls or a list of valid targets according to a pointer analysis for each function pointer.

The three types of attack described by Carlini et al. [96] are used to classify the attacks as follows.

1. *Control Data Attack* leading to control flow hijack: The adversary changes the flow of execution to a target that violates the fully precise dynamic CFG. This type includes both code injection and code reuse. The key point is that the execution path has never been part of any benign execution of the program.

2. *Non-control Data Attack* leading to control flow bending which in this thesis is divided into two types.

- Type I: The adversary changes the flow of execution to a target that does not violate the fully precise dynamic CFG. This includes any data corruption that could influence the decision points in flow of execution (decision parameters of the conditional branch instruction) or change of parameters passed to a valid function (e.g. an edge of the fully precise dynamic CFG to `execve()` with corrupted parameters).
- Type II: The adversary changes the flow of execution to a target that does not violate a fine-grained dynamic CFG. The adversary can choose from a list of valid targets that are generated for a particular function pointer due to the imprecision of the pointer analysis.

3. *Information Leakage Attack*: The adversary performs a non-control data attack that does not violate the fully precise dynamic CFG that leads to disclosure of sensitive information.

To summarise: the first type of attack involves the use of an invalid edge of a fully precise dynamic CFG of the program; the second and fourth types make use of the precise edge, but with corrupted input which results in either confined code execution or information disclosure [96]; and the third type uses one of the valid targets that are generated due to imprecision of pointer analysis for functions.

In order to discuss countermeasures, first the attack needs to be formally defined. In general terms if a successful attack is expressed as proposition p for the exploitable executable α and the adversary's intended computation sequence β, then the fact that at least one execution of α results in successful exploitation (hence exploitable) can be expressed in PDL as: $\langle \alpha; \beta \rangle p$. Measures are proposed that would result in all executions of the exploitable program α satisfying the proposition $\neg p$ expressed as: $[\alpha; \beta]\neg p$. The proposition satisfiability depends on the type of the attack and the protective measure. The focus of this chapter is the first and third types of attack that involve the use of an invalid edge or a valid target from a list, whereas the second and fourth types cannot be defended against even with the Ideal CFI enforced. The control flow hijack attack is formally defined as follows.

Definition 3.4.1. Let the exploitable program α be comprised of the memory sequence of atomic actions $a_1; a_2; \ldots; a_n$, the adversary's program β be the computation sequence $b_1; b_2; \ldots; b_m$, and $CS(\alpha)$ the set of all possible computation sequences of the program α, given the program α is exploitable then there exists a partial computation sequence α_1 that leads to the execution of the adversary's intended program β and $\alpha_1; \beta \in CS(\alpha)$. The proposition $p :=$ *"successful control flow hijack"* is considered to be true for the computation sequence $\langle \alpha_1; \beta \rangle p$ if and only if a_k is the last instruction in α_1 and b_1 the first instruction of the adversary's intended computation sequence β and $b_1 \neq a_x$ where $a_x \in \alpha$ the next action under benign execution of α. The successful control flow hijack is simply expressed as $\langle a_k; b_1 \rangle p$ focusing on the transition from the benign execution to the adversary's computation sequence.

The distinction between computation sequence and memory sequence is clear when comparing the user program α with the adversary's sequence β that for instance would be scattered over much larger memory sequences in case of a heap spray, or scattered over the code of the program α as ROP gadgets, or be an injected memory sequence on the overflown stack, crafted to be executed as the given sequence $b_1; b_2; \ldots; b_m$. Definition 3.4.1 expresses that if a program is exploitable then there exists a computation sequence of the program that transfers the flow of execution to the intended computation sequence of the adversary.

The countermeasure against the control flow hijack is formally defined by focusing on the transition of the flow of execution to the first instruction of the adversary summarised as $a_k; b_1$ where $1 \leq k \leq n$. To satisfy the proposition $p :=$ *"successful control flow hijack"* in the state transition $(s_k, s_{k+1}) \in R_{a_k}$ the instruction a_k must belong to one of the following sets (Definition 3.3.6):

1. $a_k \in SQ$
2. $a_k \in DJ$
3. $a_k \in CB$
4. $a_k \in IJ$

5. $a_k \in \{halt\}$

If the proposed protective measures are expressed as the proposition ψ as a precondition to the execution of the exploitable program α comprised of atomic actions $a_1; a_2; \ldots; a_n$ under attack with the adversary's intended computation sequence β comprised of atomic actions $b_1; b_2; \ldots; b_m$, then the notion of preventing the successful exploitation can be expressed as $\psi \implies [\alpha; \beta]\bot$. That is the transition from exploitable code to the intended code of the adversary will fail in all executions of α. The next section formally defines the required precondition(s) ψ to satisfy the aforementioned expression.

3.5 Protective Measures

To protect the integrity of the flow of execution at run time, it is necessary to add the required controls that assure the flow of execution follows the intended path by the programmer. The classic CFI [82] adds unique labels to all of the targets of indirect jumps, and then enforces the policy using in-line reference monitors that precede indirect jumps to verify whether the label of the target of that indirect jump is valid. Function call and return are both handled with the indirect jump instruction. The policy is enforced through labelling. This however creates problems with dynamic library functions as these functions can be called from different executable code and the same function may be called from different points making multiple paths available for return. Figure 3.1 clarifies this point where two different executable code (`foo` and `bar`) call the same function (`f2`) from a dynamically linked library.

The static CFG of the executable `foo` has six edges (numbered 1 to 6 in Figure 3.1), however by execution time not all edges are valid paths at all times, which is why even a fully precise static CFG is more permissive than the intended execution path. This is due to the fact that a return instruction of a function creates an edge to all of the call sites to that function, but only the edge to the most recent call is valid at run-time. For example in Figure 3.1 edges 3 and 6 for return from function

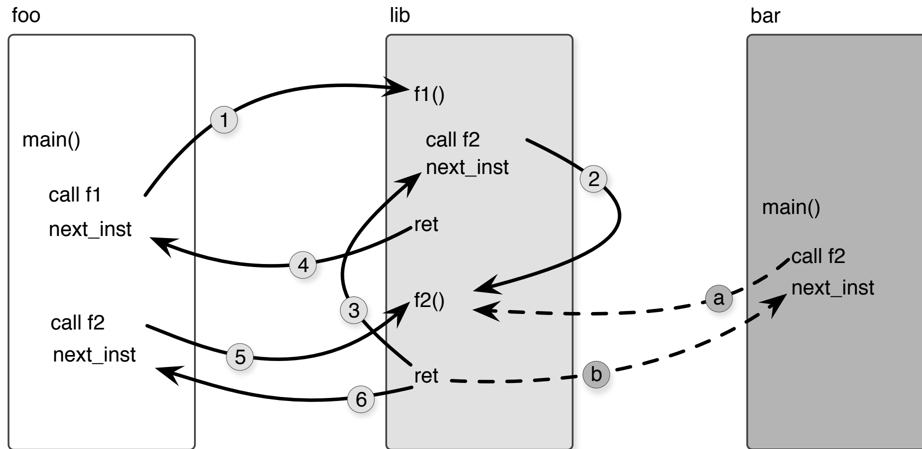

Figure 3.1: Control Flow Graph in dynamically linked executable

`f2` in library `lib` are both valid edges in CFG, but edge 3 is only valid when the most recent call is within the function `f1`. Edge 6 is only valid when the most recent call is from executable `foo`. The validity of the edges depend on information that is only available at run-time. For the library functions, as the same code is available to multiple executable code, the edges are only valid in the execution context of each code e.g. the edge b is not valid in the context of the executable `foo`. In the above example the execution context of the program `foo` at run-time will contain the addresses of the functions in the shared library as well as the return address for each function call at any given state of the program during the execution. These addresses are control metadata of the program in execution. The execution context then can be defined as the relevant control metadata of the benign execution of the program α at a given state. In an abstract model this can simply be referred to as the *intended execution path* without too much concern for data structures that implement the concept. The required information to protect this intended path will be discussed in the following sections. In the approach the forward and backward edges of the control flow, to and from a function, are treated accordingly using two different machine instructions that require different verifications.

3.5.1 Forward Edge

An edge in a CFG is defined using both starting point and end point. To protect an edge, both points must be considered. In the case of a forward edge, the start is the offset of the jump instruction; the end is the called function. If the start point is neglected in the provided protective method, then the CFG will be reduced to a list of targets where any of the targets can be a valid destination for an indirect jump. To assure that both ends of an edge are verified in the current execution context it is required that the *offset* of the indirect jump instruction and the destination of the call be recorded at compile time. The function can be a library and dynamically linked or a local function. For analysis it is assumed that all function calls whether local or dynamically linked are all translated to an indirect jump instruction. That is the call to local functions, although translatable to direct jump, are treated similar to library functions and translated to an indirect jump instruction by compiler. To simplify the notation in provided definitions the called function is referred to as $f_x \in \alpha \vee \alpha'$. The notation $f_x \in \alpha$ represents a local function. The notation $f_x \in \alpha'$ represents a library function where $\alpha' \equiv \bigcup_{0 \leq j \leq m}^{lib_j}$ expresses the union of all libraries that are accessible to the executable α. At run-time the addresses will be adjusted to physical addresses which is commonly supported in hardware. This allows the code to be position independent. For reasons that are explained later only forward edges are translated to indirect jump instruction. For returns from function calls a dedicated return instruction is used.

To protect the forward edges, the valid destination for each forward indirect jump instruction is uniquely specified for the executable α comprised of an atomic instruction sequence $a_1; a_2; \ldots; a_n$ and called functions $f_1; f_2; \ldots; f_m$ in its set of Authentic Calls (AC) as follows.

Definition 3.5.1. *Set of Authentic Calls for Executable α using Libraries α':*
$AC \stackrel{def}{=} \{(\kappa, f_x) \mid a_\kappa \in \alpha, Decode(a_k) = jmp\ r_s \wedge f_x \in \alpha \vee \alpha'\}$ where $\alpha' \equiv \bigcup_{0 \leq j \leq m}^{lib_j}$

Recording the indirect jump instruction address precisely specifies where within the body of the code a destination is reachable. In other words an indirect jump

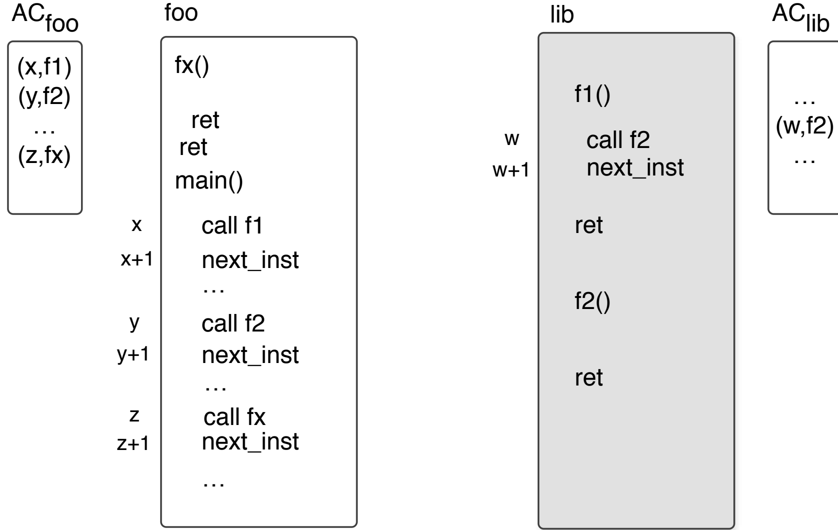

Figure 3.2: Set of Authentic Calls for the executable `foo` and library `lib`.

instruction can be executed if and only if it is located at a pre-recorded address and its destination is also a registered address for this unique indirect jump. This will be added as a precondition to the jump instruction. Figure 3.2 shows the set of authentic calls for the executable `foo`.

As it will be shown, the theorem proves the execution will follow the defined precise edge if the provided precondition is satisfied. The goal is to provide precisely one valid edge for each indirect jump instruction. However as it has been discussed in the literature [89, 99, 100, 104] there are circumstances where multiple valid destinations exist at compile time. One such case is the use of a function pointer. In this case the elements of the *AC* set can be defined as 3-tuples *(condition, offset, target)* where the first element specifies a verifiable condition for the target to be valid. To make this point clear, the set of Authentic Calls for the program α comprised of atomic actions $a_1; a_2; \ldots; a_n$ can be redefined as follows.

Definition 3.5.2. *Set of Authentic Calls*
$AC \stackrel{def}{=} \{(c_j, \kappa, t_j)\}$ such that the following conditions hold:

- $c_j \in \{T, F\}$
- $a_\kappa \in \alpha \wedge Decode(a_k) = jmp\ r_s$

- $t_j \in (\alpha \vee \alpha')$ where $\alpha' \equiv \bigcup_{0\leq j\leq m}^{lib_j}$

where $c_j = T \implies Valid(t_j)$ and the $Valid()$ function expresses that the target t_j is a valid target for the condition c_j.

The condition c_j has the value T for the normal function calls where the destination virtual address is known at compile time or is in the form $\overline{c_1 c_2 \ldots c_{j-1}} c_j \overline{c_{j+1} \ldots c_q}$ for the target t_j and the fixed *offset* κ. The condition c_j is comprised in such a way that it only allows one valid destination at run-time.

The key point is that if it is not possible to generate a precise list of valid destinations $(t_1, t_2, \ldots, t_q)$ with verifiable conditions $(c_1, c_2, \ldots, c_q)$ that uniquely identifies the valid target for a particular indirect jump (fixed κ) at run-time, then the theorem can only guarantee that the execution path will be as precise as the provided list, which would be fine-grained but not fully precise. It also means that the program is non-deterministic where there exists a point in the program that at least two destinations are unconditionally reachable. In such case performing a control flow bending attack not only includes passing corrupted input to a reachable function (Type I) but also choosing a destination from a valid list (Type II).

3.5.2 Backward Edge

Each function $f_x \in \alpha \vee \alpha'$, $(\alpha' \equiv \bigcup_{0\leq j\leq m}^{lib_j})$, starting at a defined offset in $\alpha \vee \alpha'$, has at least one or more associated return point(s). To record the location of each return instruction within the body of a function that can be used to verify a backward edge, the set of associated return point(s) is defined as follows.

Definition 3.5.3. *Set of Return Point(s)* for executable α calling library functions $f_j \in \alpha \vee \alpha'$ where $\alpha' \equiv \bigcup_{0\leq j\leq m}^{lib_j}$:
$RP \stackrel{def}{=} \{(f_j, \xi)\}$ such that the following conditions hold:

- f_j is the function logical address

- $a_\xi \in f_j \wedge Decode(a_\xi) = ret\ r_s$ is a valid return point for function f_j

The set of associated return point(s) specifies the valid return instructions of a called function. Registering the offset of these instructions will prevent the execution of a return opcode that appears in the middle of other instructions in complex architectures by verifying the offset at run-time. The emphasis is on protecting both ends of a valid backward edge to prevent such attacks. The association of return point(s) for each function of executable `foo` and library `lib` is shown in Figure 3.3.

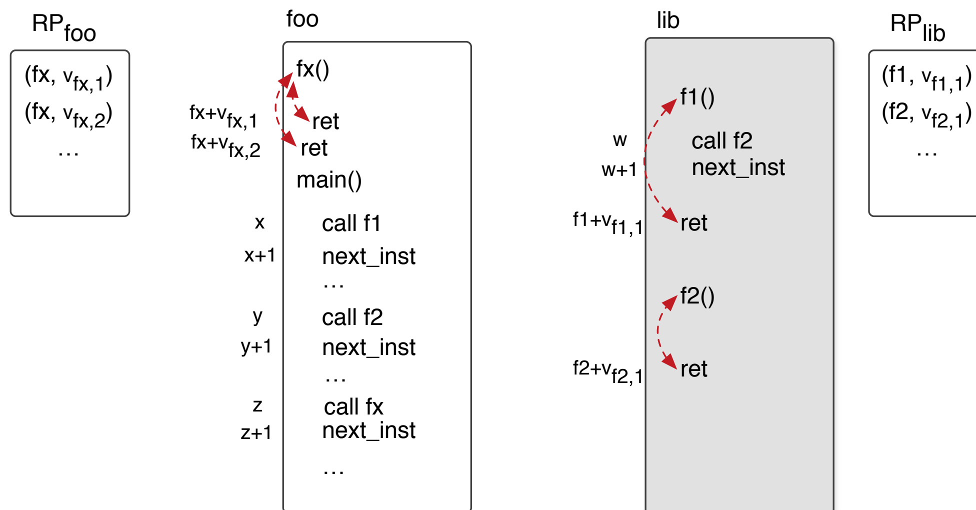

Figure 3.3: Functions' entry points and their associated return point(s)

The forward edge requires checking whether the tuple (instruction address, function address) belongs to the set of authentic calls. The forward verification is shown in Figure 3.4.

The return however requires two checks: (i) whether the return is an associated return point for this function; and (ii) whether the destination address is authentic. One way of performing these two verifications is by creating a run-time mapping, at the time of the call, that maps the two tuples. The first tuple belongs to the set AC which contains: (instruction address, authentic function x). The second tuple belongs to the corresponding set RP which contains: (function x, return point(s)). These two tuples have a shared element, the called function, which can be used to form a 3-tuple containing: (return address, function x, associated return point(s) of function x). This 3-tuple will uniquely identify the valid target address to return

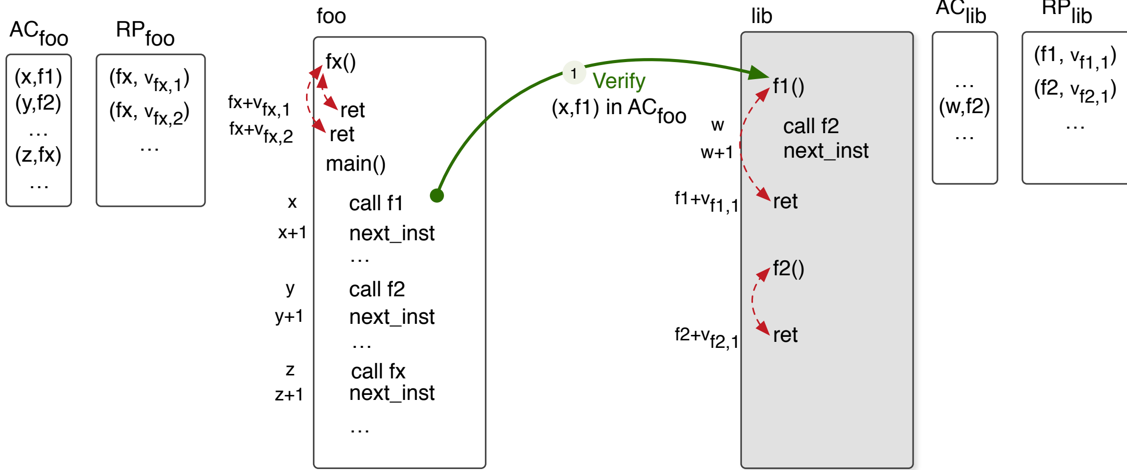

Figure 3.4: Verifying an authentic call (forward edge)

to as well as a valid return instruction. Given a function may have multiple return points, at run-time multiple mappings can be created, however only one of the return instructions will be executed and all the corresponding mappings will be removed before executing the return instruction. Alternatively, in implementation of this model, the function can only have one return instruction and all other return points can be converted to direct jumps to the address of the one return instruction.

The *Set of Runtime Mapping* which associates the return point(s) of a function defined in executable code $\alpha \vee \alpha'$ (local or library) and called within the executable code α at the time of the call (run-time) is formally defined as follows.

Definition 3.5.4. Set of Runtime Mapping for executable α is a per call sequence (with order and repetition): $RM \stackrel{def}{=} \{(\kappa{+}1, f_j, \xi)\}$ such that the following conditions hold:

- $(\kappa, f_j) \in AC_{\alpha \vee \alpha'}$
- $(f_j, \xi) \in RP_{\alpha \vee \alpha'}$

The elements of this set are 3-tuples. The first element is the address to return to in the calling code α pointed to by $pc + 1$. The second element is $Reg(r_s)$ pointing to the start of the called function f_j in $\alpha \vee \alpha'$. The third element is the associated return point for the called function f_j recorded in $RP_{\alpha \vee \alpha'}$. For each return instruction it can be then verified, at the time of return, if there exists a 3-tuple in

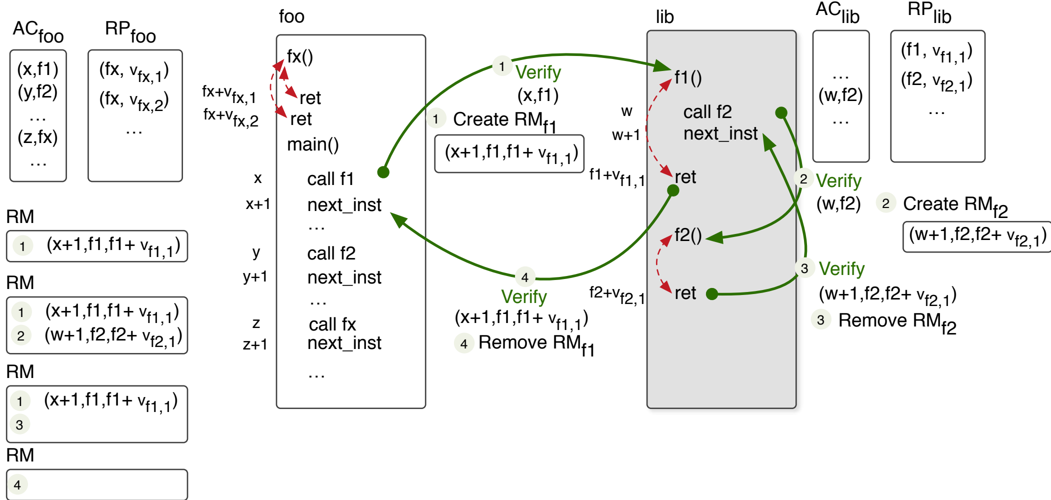

Figure 3.5: Run-time mappings of associated return point(s) of authentic function calls

the set *RP* where the target address matches the first element, the address of this function matches the second element, and the address of current return instruction matches the third element (Figure 3.5). This mapping entry is created per function call (Steps 1 and 2 in Figure 3.5) and will be removed if a match is found at the time of the authentic corresponding return (Steps 3 and 4 in Figure 3.5). The mapping creates the necessary execution context which allows achieving a fully precise dynamic (context-sensitive) and depth independent enforcement of the CFI policy. The context-sensitivity is achieved by uniquely identifying the return associated with the call. This distinguishes the valid edge from all available edges of a static CFG. The depth-independence is achieved as the run-time mapping does not limit the number of consecutive function calls before execution of any return instruction. Consecutive function calls are simply added as new elements to the set and are removed at the time of their corresponding return. In the literature, the shadow stack is considered a necessary condition for enforcing a fully precise CFI [96] whereas in this abstract model the data structure used for implementation could be in the form of a control stack, a shadow stack, or any other structure as long as it stores the specified elements used in verification and policy enforcement. If a rogue return instruction is defined as the opcode of return within the opcode of another

instruction, then it can be observed that this model can prevent the execution of a rogue return instruction. However, a shadow stack can only protect against change in the return address and is theoretically vulnerable against execution of a rogue return instruction.

3.6 Theorems of ICFI

The theorems of the Ideal CFI specify the **necessary** and **sufficient** conditions for an abstract machine to prevent the control flow hijack of a vulnerable program. Before the theorems are stated, the premises of this model are discussed in the following section.

3.6.1 Premises

The first assumption which is a required precondition is expressed in the literature as non-executable data or code and data memory separation. The thesis, however, expresses this condition as integrity of the code of an executable which is a stronger requirement. This condition is necessary with an adversary with arbitrary memory write capability. To denote the integrity property of the executable code α comprised of the atomic instruction sequence $a_1; a_2; \ldots; a_n$, the notation $Int(\alpha)$ is used as a logical proposition with true or false values. The premises of the ICFI model are defined as follows.

(A1) The precondition $Int(\alpha) \wedge_{j=1..m} Int(lib_j)$ states that the code of the program α and the libraries called by the program α cannot be changed by the adversary (the integrity of the code is intact). For clarity this precondition is expressed as $Int(\alpha) \wedge Int(\alpha')$ where $Int(\alpha') \equiv \wedge_{j=1..m} Int(lib_j)$.

(A2) The adversary cannot directly change the content of the program counter. This is represented as the post conditions on the actions (machine instructions) which specify how the content of *pc* is affected as part of the logical semantic of the defined atomic actions.

(A3) The attacker cannot modify the following sets belonging to any executable code: AC (set of Authentic Calls), RP (set of associated Return Point(s), and RM (set of Run-time Mappings). The integrity of these sets for instance could be protected with signatures, some form of secure storage, or memory bounds. We can represent this for any executable code α as $Int(RP_\alpha) \wedge Int(AC_\alpha) \wedge Int(RM_\alpha)$.

3.6.2 Theorems

The first theorem states that the integrity precondition is a necessary and sufficient condition to prevent control flow hijack attacks, defined in Section 3.4, for a program that only contains sequential, direct jump, and conditional branch type of instructions.

Theorem 3.6.1. *For the exploitable program α with memory sequence of atomic instructions: $a_1; a_2; \ldots; a_n$ and with an adversary's intended computation sequence of β comprised of atomic actions $b_1; b_2; \ldots; b_m$, for a computation sequence of α that includes the partial computation sequence $a_k; b_1$ then $Int(\alpha) \implies [a_k; b_1]\bot$, where $1 \leq k \leq n$ and $a_k \in SQ \cup DJ \cup CB$.*

Proof: For execution sequence $a_k; b_1$ there will be one of the following (A2):

1. $a_k \in SQ$ which always satisfies $s_{k+1}.pc = s_k.pc + 1$:
   - for $k < n$: a_k and b_1 are in consecutive memory locations (SQ property) which implies $b_1 \neq a_{k+1} \in \alpha$ (violates $Int(\alpha)$ A1)
   - for $k = n$: $a_n = halt$ and $a_n \notin SQ \cup DJ \cup CB \wedge [halt; b_1]\bot$ (A4 and semantic of *halt* instruction: $[halt]\bot$)
2. $a_k \in DJ$ which always satisfies $s_{k+1}.pc = w'$:
   - $w' \neq w \in_{emb} a_k$ (a_k is overwritten to point to b_1 violates $Int(\alpha)$ A1)
   - $w' = w$ hence $s_k.Mem(w) = b_1 \neq a_y \in \alpha$ (a_y in benign execution of α is overwritten with b_1 violates $Int(\alpha)$ A1)

3. $a_k \in CB$ which always satisfies $s_{k+1}.pc = w' \vee s_{k+1}.pc = s_k.pc + 1$ and implies $a_{k+1} = b_1 \vee s_k.Mem(w') = b_1$:

   - $k < n$ implies a_k and b_1 are in consecutive memory locations and $b_1 \neq a_{k+1} \in \alpha$ (violates $Int(\alpha)$ A1)
   - $k = n$, $a_n = halt$ and $a_n \notin SQ \cup DJ \cup CB \wedge [halt; b_1]\bot$ (A4 and semantic of $halt$ instruction: $[halt]\bot$)
   - $s_k.Mem(w') = b_1$:
     - $w' \neq w \in_{emb} a_k$ (a_k is overwritten to point to b_1 violates $Int(\alpha)$ A1)
     - $w' = w$ hence $s_k.Mem(w) = b_1 \neq a_y \in \alpha$ (a_y in benign execution of α is overwritten with b_1 violates $Int(\alpha)$ A1)

The second theorem states that if a program contains an indirect jump instruction the integrity property is necessary but not sufficient to prevent a control flow hijack attack. Additional checks must be performed to prevent such attacks.

Theorem 3.6.2. *For the exploitable program α with memory sequence of atomic instructions: $a_1; a_2; \ldots; a_n$ with an adversary's intended computation sequence of β of atomic actions $b_1; b_2; \ldots; b_m$, a computation sequence of α that contains the partial sequence of $a_k; b_1$, and $Int(\alpha') \equiv \wedge_{x=1..m} Int(lib_x)$ then:*
$\Big(Int(\alpha) \wedge Int(\alpha')\Big) \wedge \Big((k,\epsilon) \in AC_{\alpha\vee\alpha'} \vee (z,\epsilon,\xi) \in RM\Big) \implies [a_k; b_1]\bot$, *where* $1 \leq k \leq n$ *and* $a_k \in IJ$.

Proof: For execution sequence $a_k; b_1$ where $a_k \in IJ$ which always satisfies $s_{k+1}.pc = s_k.Reg(r_s)$, there will be one of the following (A2):

1. $Decode(a_k) = jmp\ r_s$ then:

   - $s_k.Reg(r_s) \neq f_j$ violates $(k, f_j) \in AC_{\alpha\vee\alpha'}$ (protected by A3)
   - $s_k.Reg(r_s) = f_j$ implies $b_1 \neq a_y \in \alpha \vee \alpha'$ (a_y the beginning of f_j is overwritten by b_1 violates $Int(\alpha) \wedge Int(\alpha')$ A1)

   Hence: $Int(\alpha) \wedge (k, f_j) \in AC \implies [jmp\ r_s; b_1]\bot$.

2. $Decode(a_k) = ret\ r_s$ then:

   - $(s_k.Reg(r_s), f_j, s_k.pc) \notin RM$ which implies either $(s_k.Reg(r_s), f_j) \notin AC_{\alpha \vee \alpha'}$, (unauthentic call to local or library function) or $(f_j, s_k.pc) \notin RP_{\alpha \vee \alpha'}$, (unregistered return from local or library function)

   - $(s_k.Reg(r_s), f_j, s_k.pc) \in RM$ implies $b_1 \neq a_z \in \alpha \vee \alpha'$ (a_z belonging to benign execution of α is overwritten with b_1 violates $Int(\alpha) \wedge Int(\alpha')$ A1)

   Hence: $A1 - 3 \wedge (x+1, f_j, \xi) \in RM \implies [ret\ r_s; b_1]\bot$.

Table 3.4 summarises the required preconditions for each instruction to prevent control flow hijack attacks.

3.7 Summary

A control flow hijack attack is a form of exploitation where the adversary changes the flow of execution from the programmer's intended path to an injected or existing block of code that violates the benign execution of the program. One way to prevent such exploitation is to prevent the change of the flow of execution from the intended path. This chapter discussed a formal model to protect against control flow hijack attacks. It proposed the necessary conditions to protect all forms of change in the flow of execution by categorising the machine instructions based on their effect on the PC register that controls the execution path of the program. The instructions are divided into four types: sequential, direct jump, conditional branch, and indirect jump. The required conditions are expressed in two theorems. The first theorem specifies that the integrity property of an executable code can protect against these attacks as long as all instructions are of type sequential, direct jump, or conditional branch. This is due to the fact that for the aforementioned instructions, change of the flow of execution from the intended path requires adversarial code content change that would violate the integrity of the code. The second theorem states

that the integrity property of the executable code is a *necessary* condition but is not *sufficient* if the code contains an indirect jump instruction for which additional checks must be performed. The additional check for an indirect jump is required as the change in flow of execution can happen without changing the code content. To protect the flow of execution, it is necessary to verify if the destination address which is provided as the content of a register is an authentic address, or in other words, is the intended destination by the programmer. The check for return needs to verify that two conditions are satisfied. The first condition is whether the return instruction belongs to a function such that the function was entered through an authentic call. The second condition is whether the destination address is next to the authentic call. This is done by creating a dynamic map at the time of the call that links the registered return point(s) of the called function to the return address after the authentic call.

In a control flow bending attack the adversary changes the flow of execution by corrupting non-control data, which influences the flow of execution but does not violate the programmer's intended path. For instance the adversary corrupts a variable that affects a conditional branch instruction. Another example of such attack is when the adversary changes the parameters passed to a function where the call to the function belongs to a benign execution of the program, for instance passing "`/bin/sh`" to `execve()` function instead of any other intended executable file. The proposed ICFI model cannot prevent these types of attack as the benign execution path is not violated. To protect against these types of attacks, non-control data variables that could affect the flow of execution must be protected from corruption. In Chapter Four a memory model that can prevent these types of attacks is discussed. Another protective measure is proposed to protect against memory address leakage that is used by the adversary to craft control flow attacks.

Table 3.4: Instruction preconditions to prevent control flow hijack attack

Instruction	PDL expressions
$nop \in SQ$	$Int(\alpha) \implies [nop]p_1$ $p_1 \equiv s'.pc = s.pc + 1$
$add \in SQ$	$Int(\alpha) \implies [add\, r_d, r_s, r_t]p_1 \wedge p_2$ $p_1 \equiv s'.Reg(r_d) = s.Reg(r_s) + s.Reg(r_t)$ $p_2 \equiv s'.pc = s.pc + 1$
$addi \in SQ$	$Int(\alpha) \implies [addi\, r_d, r_s, w]p_1 \wedge p_2$ $p_1 \equiv s'.Reg(r_d) = s.Reg(r_s) + w$ $p_2 \equiv s'.pc = s.pc + 1$
$movi \in SQ$	$Int(\alpha) \implies [movi\, r_d, w]p_1 \wedge p_2$ $p_1 \equiv s'.Reg(r_d) = w$ $p_2 \equiv s'.pc = s.pc + 1$
$ld \in SQ$	$Int(\alpha) \implies [ld\, r_d, r_s(w)]p_1 \wedge p_2$ $p_1 \equiv s'.Reg(r_d) = s.Mem(s.Reg(r_s) + w)$ $p_2 \equiv s'.pc = s.pc + 1$
$st \in SQ$	$Int(\alpha) \implies [st\, r_d(w), r_s]p_1 \wedge p_2$ $p_1 \equiv s'.Mem(s.Reg(r_d) + w) = s.Reg(r_s)$ $p_2 \equiv s'.pc = s.pc + 1$
$jd \in DJ$	$Int(\alpha) \implies [jd\, w]p_1$ $p_1 \equiv s'.pc = w$
$bgt \in CB$	$Int(\alpha) \implies bgt\, r_s, r_t, w \equiv [(?\varphi; jd\, w) \cup (?\neg\varphi; nop)]p_1$ $p_1 \equiv s'.pc = w \vee s'.pc = s.pc + 1$ $\varphi \equiv s.Reg(r_s) > s.Reg(r_t)$
$jmp \in IJ$	$Int(\alpha) \wedge Int(\alpha') \implies [(?\varphi; jmp\, r_s) \cup (?\neg\varphi; halt)](p_1 \wedge p_2) \vee \bot$ $\varphi \equiv \big(s.pc, s.Reg(r_s)\big) \in AC_{\alpha \vee \alpha'}$ $p_1 \equiv s'.pc = s.Reg(r_s)$ $p_2 \equiv RM \cup \{(s.pc + 1, s.Reg(r_s), rp_i)$ where $s.Reg(r_s)$ *points to* f_j and $(f_j, rp_{1..x}) \in RP_{\alpha \vee \alpha'}$
$ret \in IJ$	$Int(\alpha) \wedge Int(\alpha') \implies [(?\varphi; ret\, r_s) \cup (?\neg\varphi; halt)](p_1 \wedge p_2) \vee \bot$ $\varphi \equiv (s.Reg(r_s), f_j, s.pc) \in RM$ $p_1 \equiv s'.pc = s.Reg(r_s)$ $p_2 \equiv RM - \big(s.Reg(r_s), f_j, rp_{1..x}\big)$ where $(f_j, rp_{1..x}) \in RP_{\alpha \vee \alpha'}$
$halt$	$Int(\alpha) \implies [halt]p_1$ $p_1 \equiv \bot$

Chapter 4

Memory Integrity Model

4.1 Introduction

Memory corruption is the underlying cause of malicious code execution where the adversary affects the flow of execution by either changing the control or non-control data. In this chapter a formal model to protect the integrity of the memory against memory corruption leading to exploitations is provided. The goal of this formal model is to prevent the adversary controlling a corrupted memory cell that will affect the flow of execution. The model can then prevent the attacks defined in the previous chapter as non-control data attacks leading to control flow bending as well as the other control flow hijack attacks.

A confidentiality model is then discussed to protect against information leakage attacks, aimed particularly to prevent the leakage of memory addresses that can be used in crafting control flow attacks. Sections 4.2 and 4.3 provide a review of the literature regarding memory corruption and the countermeasures. The memory integrity model is defined in Section 4.4 which provides protection against memory corruption leading to all forms of change in flow of control. The memory confidentiality model that prevents leaking of memory addresses is then discussed in Section 4.5. The chapter is concluded in Section 4.6.

4.2 Memory Corruption

Szekeres et al. [118] describe the memory corruption attacks through a 6-step process. The first step is to make a pointer invalid which happens when a pointer: (i) goes out of bounds; or (ii) becomes a dangling pointer. A dangling pointer is a pointer that is pointing to a deleted object. Dereferencing an invalid pointer will cause an error which can be classified as:

1. a *Spatial Error*: happens when an out of bound pointer is dereferenced;
2. a *Temporal Error*: happens when a dangling pointer is dereferenced.

To make a pointer go out of bounds an attacker can use various exploitation techniques such as [118]:

- triggering an unchecked allocation failure which could make a pointer `Null` and exploitable in kernel-space [119];
- manipulating an array pointer without bounds checking in a loop which results in buffer overflow or underflow;
- manipulating the index to an array with bounds checking vulnerability which could result in array pointer pointing to an arbitrary location (e.g. integer overflow, sign manipulation or truncation);
- corrupting another pointer as a primary step in achieving one of the aforementioned techniques.

To make a pointer a dangling pointer an attacker can exploit an incorrect exception handler responsible for deallocating an object but which does not reinitialise the pointer to the object as part of the deallocation process. The temporal memory errors are also known as *use-after-free* vulnerability [118] as the pointer is used after returning the memory it was pointing at to the memory management process. An exploited pointer that is under the attacker's control can escape the locality of the exploited pointer and be used with a global range [118]. In the second step the

attacker may read or write the exploited pointer to achieve one of the following outcomes:

- to modify a data pointer, to make another pointer go out of bounds;
- to modify the code section of the executable, leading to a code corruption attack;
- to modify a code pointer, which leads to a control flow hijack attack;
- to modify a data variable, leading to a control flow bending attack;
- to modify an output data variable, leading to information leakage.

4.3 Memory Safety

Various methods have been proposed to provide memory safety to protect against spatial or temporal errors or both. In the following section these protective measures are discussed.

4.3.1 Spatial Memory Safety

Nagarakatte et al. [120] propose a pointer bounds checking technique to provide spatial memory safety. Their goal is to achieve source code compatibility, completeness, and separate compilation. They classify other solutions in two broad classes: (i) Object-based; and (ii) Pointer-based.

In the object-based approach, bounds information is associated with objects rather than the pointer to the object. The disadvantages of this method are: (i) the out of bounds pointer requires special treatment; (ii) verifying the bounds of a pointer to an object becomes a range lookup which is implemented using a low performance splay tree data structure; (iii) the implementation generally does not capture all spatial memory violations and is incomplete.

In the pointer-based approach, the base and bound information of each pointer is tracked with the pointer itself referred to as *fat pointer* representation. The

pointer-based approach addresses the issue of multiple pointers pointing to the same object of the object-based approach as each pointer has its own base and bound information. This method can provide complete spatial memory safety, however the existing solutions such as CCured [23] require modification to the source code where due to the changes of the memory layout these modifications require programmer intervention.

To achieve spatial memory safety SoftBound [120] uses disjoint metadata for each pointer that contains the base and bound of that pointer. The code is then instrumented to verify the bounds of the pointer at each read and write. The disjoint metadata does not change the memory layout which provides the compatibility of the object-based approach. The association of the base and bound information per pointer, provides the completeness of the pointer-based approach.

In SoftBound the C code is translated to a generic intermediate form, where for every pointer value the corresponding base and bound intermediate values are created. To provide pointer checking, for each dereference of the pointer, code is inserted that verifies the base and bound of that pointer (Listings 4.1 and 4.2).

Listing 4.1: Pointer check function [120]

```
void check(ptr, base, bound, size) {
  if ((ptr < base) || (ptr + size > bound)) {
    abort();
    }
  }
```

Listing 4.2: Pointer dereference [120]

```
  check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
  value = *ptr;  // original load
```

To address the two ways of pointer creation in C, SoftBound inserts code at each `malloc()` call-site to set the values of base and bound for the pointer that receives the returned value for the allocated memory (Listing 4.3). If the returned value is `NULL` the bound is also set to `NULL`.

Listing 4.3: Pointer check for dynamically allocated memory [120]

```
ptr = malloc(size);
ptr_base = ptr;
ptr_bound = ptr + size;
if (ptr == NULL) ptr_bound = NULL;
```

For the global and stack-allocated objects, SoftBound inserts code to set the base of the pointer and the bound is set to one byte past the size of the object which is known in advance (Listing 4.4).

Listing 4.4: Pointer check for stack-allocated pointers [120]

```
int array[100];
ptr = &array;
ptr_base = &array[0];
ptr_bound = ptr_base + sizeof(array);
```

In pointer assignments, which may also involve pointer arithmetic including array indexing, the left-hand side receives the base and bound of the original pointer (Listing 4.5).

Listing 4.5: Verification for pointer assignment and arithmetic [120]

```
newptr = ptr + index;  // or &ptr[index]
newptr_base = ptr_base;
newptr_bound = ptr_bound;
```

To avoid internal object overflow, SoftBound allows narrowing the bounds of pointers which however could result in false violation for certain C idioms.

Davietti et al. [121] have designed HardBound, a hardware supported method of bound checking for pointers that adopts the principle of fat pointers without changing the memory layout. In this method each memory cell and register is virtually considered to be a triple (*value, base, bound*). A one-bit *tag* per memory cell will distinguish non-pointer variables from pointers, where the former does not require bound checking. The base and bound information for each pointer is stored in a shadow memory space and every pointer operation is verified against these values.

The design allows the tag and bound information to be cached to reduce the required memory access for each bound check operation. The compiler will generate the required `setbound` instruction for each pointer at the time of definition of the pointer which will initialise the base and bound of that pointer in the associated shadow storage. For each pointer operation the compiler will generate the required instructions to verify whether the pointer is within its defined bounds. The HardBound method does not require change in the C program, does not change the memory layout, and achieves better performance than the software methods such as CCured [121].

In the Intel Memory Protection Extensions (MPX) four new 128-bit registers and eight instructions are introduced to provide bounds checking for pointers [122]. The new Intel MPX instructions check the lower and upper bounds of a pointer against its current value and throw an exception if the bounds are violated. As four registers are not sufficient to provide bounds checking for all pointers the bounds registers can be spilled. The spilling bounds can also be used to pass pointers as function call parameters. The MPX-enabled processors allow the four bounds registers, two configuration registers, and one status register to be saved during context switches. This requires operating system support. The new MPX instructions are treated as `nop` instructions for the processors that do not support these instructions allowing the compilation for the new instruction set to be backward compatible. In 64-bit mode the MPX processor allows a 2 Gigabyte Bound Directory where each entry points to a 4 Megabyte Bounds Table. Each Bounds Table entry contains the lower bound, upper bound, pointer value, and a reserved metadata field. The Bound Directory is allocated from virtual memory as a Demand Zero page and is only allocated when a page fault occurs. To use the features of Intel MPX processors, compilers must implement the following [122]:

- For each pointer, the bounds must be calculated according to the C/C++ standard.
- Before each pointer dereference the bounds must be verified.

- For each pointer, the Bounds Table entry must be maintained.
- Assembler and linker must be modified accordingly to support new Intel MPX instructions.

The MPX instructions provide a low overhead, hardware-assisted spatial memory safety implementation tool that could remove the out-of-bounds use of pointers given that the operating system and compiler are MPX enabled and the user code is properly instrumented.

4.3.2 Temporal Memory Safety

Dangling pointers and double-free errors are created due to manual memory management and could lead to crashes, data corruptions, and security vulnerabilities. There are various methods aiming at identifying these types of memory errors. The temporal checking methods can be divided into two broad categories: (i) location-based; and (ii) identifier-based. In location-based approaches the status of each memory location is recorded in an auxiliary data structure and maintained using allocation and deallocation code, `malloc()` and `free()` respectively. This approach however fails to detect dangling pointers for reallocated memory locations [123]. In other similar methods a tree structure or a shadow-space is used for all of the allocated regions of the memory which leads to slow lookup or high memory overhead respectively [123].

In identifier-based approaches, a unique key which is never reused is assigned to each allocated memory region and associated with each pointer. This will assure that the identifier will persist even if the memory is freed. The implementation of identifier-based approach can be classified into two classes: (i) per-pointer metadata; and (ii) set-based. One method of implementation using per-pointer metadata is using the concept of a fat pointer through change of the structure of the pointer. This will result in change of memory layout and compatibility issues. The set-based approach requires that the identifier be added to a set for the allocations

and removed for deallocations. This is generally implemented using hash tables and could result in significant run-time overheads.

Most of these methods are incomplete and cannot find all temporal errors [123]. To address this issue and overhead in terms of memory and performance, Nagarakatte et al. [123] propose an approach based on two techniques: (i) each allocated memory region is associated with a unique key; and (ii) the unique identifier is stored disjoint to the memory region avoiding memory layout change. The Compiler-Enforced Temporal Safety for C (CETS) stores the disjoint metadata to keep the memory layout unchanged and uses compiler-based instrumentation and directly-accessed data structures to avoid performance degradation. The CETS lock and key approach stores two word fields for each pointer: (i) a unique allocation key; and (ii) a lock address pointing to a lock location. The lock location is verified for each temporal access. To protect against calling `free()` on pointers not returned by `malloc()` and double-free errors, CETS maintains a map from keys to pointers that can be freed.

4.3.3 Complete Memory Safety

The SoftBound and CETS combined provide complete memory safety with 116% overhead [118, 123]. Using the Intel MPX instruction set may significantly improve the performance of the spatial memory safety approaches. Both methods provide formal proofs using the Coq automated theorem prover for a fragment of C and in a single-thread environment [120, 123].

4.3.4 Tag-based Architecture

Various forms of security policies have been proposed in the literature as in-line reference monitors such as software compartmentalisation (Sandboxing), Control-Flow Integrity, and memory bounds checking. Amorim et al. [124] describe a formal method to define such policies using a tag-based architecture where the architecture can accommodate required tags for the defined policy. To formally prove their work

they propose a high-level symbolic machine and verify four micro policies namely: (i) dynamic sealing to protect cryptographic keys and specify encrypted data in memory; (ii) compartmentalisation; (iii) control-flow integrity; and (iv) memory safety. In this work the specification of the micro policy is defined using an ideal abstract machine with a PUMP-like architecture [125] where the instruction semantics have a built-in information flow policy. A symbolic machine is then used for each of the information-flow polices to dynamically express the required mechanisms for that policy. Then a concrete machine implements the specified mechanism in symbolic machine with a software controller that interacts with these low-level mechanisms. The formal work proves the correctness of the transitions from the abstract machine to the proposed implementation in the concrete machine. The result however relies on the accuracy and security of the specified information-flow policy. For instance the CFI policy in the proposed tag-based architecture implements the equivalent policy of Abadi et al. [30] which suffers from inaccuracy of equivalent classes. The flexibility of the proposed tag-based architecture comes with the price of overlapping policies with similar goals. For instance the CFI policy and memory safety are both targeting the different aspects of vulnerability and exploitation. A complete and secure implementation of memory safety would make the CFI policy redundant. Dhawan et al. [125] propose an architecture that can enforce any policy that can be expressed using a defined tag-based rule structure. This tag-based architecture, called PUMP, is used in the formal work of Amorim et al. [124] as a generic policy enforcement in the form of an in-line reference monitor.

4.3.5 Type-based Non-interference Languages

Volpano and Smith [126] propose a type-based approach to enforce confidentiality and integrity of the program data flow. The general idea is that all program inputs and outputs are classified with appropriate security labels. To prevent the program from leaking sensitive data, a classified output of the program at some level cannot change due to change only in the input at a higher level. This would otherwise mean

the information at a higher security level is leaked by observing the change in the lower security level output. This policy enforcement is defined based on data types for security labels that identify the sensitivity of data types. The operations allowed using the security-typed data is then specified by various rules that when followed guarantees the policy enforcement. The type system is then formally proved to be sound and complete [126]. The non-interference property however can be too strong for practical implementation. For instance in the login process, the user provides a password which has a higher security level than user provided data. For the login process to succeed, the information provided by the user needs to be compared with the stored password, which violates the non-interference policy rules [127]. To address this issue, Li and Zdancewic [127] propose a generalised framework for downgrading strong non-interference policies that can be enforced in practice using mechanisms such as type systems. In this method the non-interfering program expressed as $f(h, l)$ is factored into two parts: (i) $f_H(h, l)$ represents the high security part; and (ii) $f_L(l)$ represents the low security part. The observation is that the low security part does not rely on high security input. In this approach the proof of non-interference is provided by transforming the original program to the aforementioned structure for a low security part that does not rely on high security input and a downgrading policy function for a part that depends on both high level and low level security input [127].

4.4 Memory Model

This section describes a memory model inspired by Biba's integrity model [128] and Bell-LaPadulla's confidentiality model [129]. The integrity model aims to prevent memory corruption by assigning different levels of integrity based on the trustworthiness of the variables. The confidentiality model can prevent memory leaks by assigning a higher confidentiality level to memory cells that contain sensitive program internal values. The goal of both models is to protect the program in execution against attacks rather than protecting user provided values. The integrity model

and the required preconditions are discussed first, followed by the changes to the instruction set that enforce these requirements. The theorem of this model with its proof is stated in a separate subsection. The confidentiality model is then defined with a similar structure.

4.4.1 Memory and Register Representation

To accommodate the new requirements to protect memory from corruption and unauthorised disclosure of content the structure of memory and its representation is redefined. In this new model each memory cell is represented as a 4-tuple:
(Address, Content, Integrity Level, Confidentiality Level).
Each register is also represented as a 4-tuple:
(Register Number, Content, Integrity Level, Confidentiality Level).
The memory definition is amended as follows for an l-bit architecture with two levels for each of the integrity and confidentiality levels of memory cells and registers.

Definition 4.4.1. *Memory*:
$Mem : \{0,1\}^l \times \{0,1\}^l \times \{0,1\} \times \{0,1\}$

Definition 4.4.2. *Register file*:
$Reg : \{0,1,\ldots,31\} \times \{0,1\}^l \times \{0,1\} \times \{0,1\}$

Before discussing the requirements of the memory model in terms of conditions and checks built into machine instructions, various high level abstract concepts need to be linked to their representations in the more concrete machine model. The following section creates this link between these concepts in high level programs and their corresponding representations in the machine model.

4.4.2 Preliminary Definitions for Memory Integrity Model

User input in a vulnerable program provides the opportunity of exploitation to the adversary. That is a program that does not interact with user in any way cannot be exploited even if it is vulnerable. In the concept of programming languages,

the interaction with user is performed through the use of variables. The variables however serve a broader purpose than interaction with the user. They store the state of the program during execution. To distinguish between different types of variables based on their purpose, the variables are categorised. The relationship between the higher abstraction level that is the variables and the lower abstraction that is the memory cells assigned to these variables can then be formally defined. Two general categories of variables are defined in the model:

1. User provided, which defines variables provided by user and includes any form of input such as standard input/output, file, and network;
2. Program internal, which defines variables that are used by the programmer as part of the internal state of the program.

The integrity level then will be assigned to each category based on the level of trust. Variables that are used internally by a program will be treated as containing trusted values; variables provided by the user will be treated as untrusted. To allow the flow of information between user input and program internal when necessary, in the proposed approach, the integrity level can be changed by the use of explicit instructions, conditional on passing an explicitly defined test when the change is from a lower integrity level to a higher integrity level. These instructions and their semantics are discussed in more detail in a later section.

First two sets of variables are defined: (i) those that require high integrity; and (ii) those that require low integrity. To make sure that the two sets will partition the set of all variables of the executable code α and that all variables will have a defined integrity level, all variables that must be treated as low integrity are declared explicitly by the programmer. All of the other variables are assigned to have a high level of integrity. Using this approach, user provided variables can be declared by the programmer as low integrity variables and untrusted at the beginning of the code execution.

Definition 4.4.3. *Set of Variables* of the executable code α:
$V_\alpha \stackrel{def}{=} \{v | Variable(v) \in \alpha\}$

Definition 4.4.4. *Set of Low Integrity Level Variables* of the executable code α:
$LV_\alpha \stackrel{def}{=} \{v | v \in V_\alpha \wedge IntegrityLevel(v) = Low\}$

Definition 4.4.5. *Set of High Integrity Level Variables* of the executable code α:
$HV_\alpha \stackrel{def}{=} V_\alpha - LV_\alpha$

Based on the given definition it holds that $V_\alpha = HV_\alpha \cup LV_\alpha$ and $HV_\alpha \cap LV_\alpha = \emptyset$. Assigning a high integrity level to all the variables that are not explicitly excluded is justified based on the fact that if a variable is not provided by the user, it is by principle outside direct user control. The goal of the integrity model is to protect the internal state of the program from unauthorised modification that results in change in the flow of execution. Thus, it would be logical to define the boundary between user provided input and the program internal by including all the variables used in the program that are not provided by users in the high integrity set regardless of their effect on flow of execution. This is a conservative approach in classifying variables which means all of the variables that contain sensitive values such as pointers, array indices, variables that are part of an expression for a conditional branch, or a loop will be set to the high integrity level and will be protected against corruption.

The user input, however cannot be completely isolated as the program often needs to process user data and the flow of execution may change depending on the provided input. To make sure a user provided variable will change from low integrity to high integrity when a condition is met, the programmer must define an explicit test so that when it is satisfied the variable can be trusted. This will allow the transition from an untrusted state of a user provided variable to trusted at a later point during the execution whenever such a transition would be needed. To clarify, a transition from low integrity to high integrity is required whenever the user provided variable would affect the flow of execution. As mentioned before, a low integrity variable can be classified as high when it passes its explicitly defined test. The test must be done after the variable is provided by the user. The transition

of the integrity level must happen before reading another variable from the user to remove the opportunity of corruption after a successful test. The test can be as simple as one instruction or as complex as a program. The test however must return either True or False for any given variable v. The security of the model relies on the correctness of the classification test for any variable v. It would depend on the properties of the variable and its use. For instance if a variable is of type string, the test of the length of the variable may be enough to satisfy its trustworthiness or it may require testing the content as well as the length to avoid certain characters, patterns, or checking against a black or white list. As another example for integers, the test could involve a range of acceptable values. To associate such test with its corresponding variable, a set of *Classification Tests* is defined such that the elements of this set are 2-tuples representing the defined classification test and the low integrity level variable.

Definition 4.4.6. *Set of Classification Tests:*
$CT_\alpha \overset{def}{=} \{(\varphi, v) | v \in LV_\alpha \wedge \varphi(v) \in \alpha \wedge \varphi(v) = True \text{ if } v \text{ can be trusted}\}$

Since the classification test is done on user provided value it will always involve a low integrity element. To protect the integrity of the test itself the untrusted value must be compared to a high integrity element. The result of a successful test then can elevate the integrity level of the variable associated with that test. Using this approach, before a conditional branch is used, all user provided values involved in the conditional branch can be tested and elevated to high integrity if successful. It also allows the flexibility for more complex tests on logical expressions that can be left for the compiler.

Given the integrity level of a variable is decided by the programmer and a variable is a high level language concept that will be translated and associated to a memory address or a range in the generated low level machine code, it is necessary to express this relationship formally. The concept of Type is used to bridge the association of variables and assigned memory cells, as the type-based approach is a well studied and understood topic. Without loss of generality, the compiler of a

high level programming language can be assumed to produce correct code for the type of a variable. The correct number of memory cells will be assigned to each variable of any defined type. This mapping of high level program variables to memory addresses for the executable code α (in machine language) can be expressed as an abstract function for which concrete instructions can be generated by a compiler. The association of program variables to memory cells according to the variable size plays a major role in the proposed memory integrity model. The assumption is that the compiler generates correct code for every usage of the variable within the body of the code, meaning the size of the variable is tightly controlled in the generated code.

Definition 4.4.7. $Map(v) : Mem_v[m, m + l - 1]$ where $v \in V_\alpha \wedge Type(v) : \tau \wedge Size(\tau) : l$

Since a variable may require more than one memory cell depending on the type size, a new notation is introduced to represent a memory range associated with a given variable. The expression $Type(v) : \tau$ specifies that variable v is of type τ; the expression $Size(\tau) : l$ specifies that the type τ requires l memory cells. The memory range is expressed as $Mem_v[m, m + l - 1]$ in the definition of the abstract function $Map(v)$. This range can grow from lower addresses to higher addresses or vice versa. It has no effect on the formal representations if the same policy applies to all variables in the same context. If memory grows from higher addresses to lower addresses the range can be expressed as $Mem_v[m, m - l + 1]$. The former representation is used in the remainder of this chapter, however as stated before the results will hold if the same policy is followed and address calculations are applied accordingly. A mixture of both policies can also be used, as long as it can be determined from the context which policy is applicable for a given variable to memory mapping. The required rules of the integrity model can then be enforced in the related range in that context.

All variables that are provided by the user must be treated as untrusted, unless an explicit test clears that variable for transition to higher integrity. Following this

principle, if a variable is provided by the user more than once throughout the program, its integrity level must be reset to low prior to the new reading. Although implementations of such models are not discussed in detail in this thesis, as part of enforcement of this model it seems necessary to require that library functions that deal with user input use low integrity level instructions as defined in the following section. This will allow catching variables that are not set properly by the programmer or are being reused, that is, are being read from the user for a second time after classification.

The variables are classified into two broad groups: trusted and untrusted. It must be emphasised that in the context of the memory integrity model, there is a difference between terms *valid* and *trusted*. In this model a variable may contain a value that is not valid, but the variable can be trusted because it is under the programmer's control. On the other hand a variable may contain a valid value and not be trusted because it is under a user's control. The memory integrity model does not aim to eliminate the programming mistakes or provide correctness guarantees for the program, rather it provides a means to prevent users gaining control of memory outside the boundaries defined by the programmer throughout the program and corrupting memory that may affect control flow. To clarify the point, for instance a dangling pointer may cause the program to crash but it would also prevent the user controlling the value of that pointer by enforcing the rules of the memory integrity model.

To provide formal definitions and required protective measures against memory corruption there is a need for a formal representation of the corruption itself. The memory corruption can be considered to be at least one unauthorised memory write. Since the attacker's access to memory is through user provided input, this can be expressed as a write for the variable that is being misused to an address not within its associated range. To express this formally, first a simple abstract memory write for a variable to its associated address is defined as follows.

Definition 4.4.8. $Write(v) \stackrel{def}{=} Mem_v[m, m+l-1] \leftarrow Value(v)$ where $v \in V_\alpha \wedge Type(v) : \tau \wedge Size(\tau) : l$

Variables are high level programming concepts which are represented in the machine level with assignment of memory cells. A variable may require more than one memory cell to store its value. The abstract memory write function for a variable expresses that correct instructions will be generated by the compiler for the correct memory range assigned to the variable by the $Map(v)$ program. The $Map(v)$ program uses the type and size of a variable to assign a range of memory cells to the variable. The $Write(v)$ program reinforces the fact that the compiler generates correct code for every memory write operation of a variable. The $Value(v)$ function expresses the content of the variable v that is being written to its associated memory cells. The address calculation in the proposed machine model is of the form: *base + offset*.

An abstract $Read_{IO}()$ function is defined for a variable, to express the distinction of the input/output operations from other memory accesses. This will help in formal definition of the rules for the memory integrity model. The abstract $Read_{IO}(v)$ for variable v expresses the process of transferring user provided input stored in a memory mapped range of addresses, referred to as Mem_{IO}. The formal expression omits the process of reading the user provided input from the IO device to Mem_{IO} as this operation can be flexible and independent of the variable size. As it will be discussed the memory IO addresses will always be set to low integrity level.

Definition 4.4.9. $Read_{IO}(v) \stackrel{def}{=} Mem_v[m, m+l-1] \leftarrow Mem_{IO}[m_{io}, m_{io}+l-1]$ where $v \in V_\alpha \wedge Type(v) : \tau \wedge Size(\tau) : l$

The $Read_{IO}(v)$ program would be comprised of appropriate number of `load` and `store` instructions depending on the size of the variable. After discussing in more detail the instructions of the machine model, the definition will be amended (Definition 4.4.30).

A memory corruption then would be either manipulation of the parameters of address calculation involved in one or more memory writes, or a direct corruption

of a pointer that is dereferenced for various purposes. The corruption would then include any of the following scenarios: when base, offset or both parameters used in calculating the target address of a memory write are out of their intended bounds or when a register that contains the target of an indirect jump, another memory write, or a memory read is corrupted. It will be shown that if the requirements of the memory integrity model is followed the aforementioned cases will be detected and prevented.

4.4.3 Requirements of Memory Integrity Model

To protect against memory corruption the integrity model must meet certain requirements. The first requirement is quite straightforward and is based on the classification of the variables and the trustworthiness of each of these classes. To express this notion formally, there needs to be a formal definition of the set of user provided variables as one of the discussed classes.

Definition 4.4.10. The *Set of User Provided Variables* for executable code α:

$UV_\alpha \stackrel{def}{=} \{v \mid v \in V_\alpha \wedge v \text{ provided by user } \}$

The first Memory Integrity Requirement (MIR) states that all user provided variables must be assigned to low integrity level defined by the set of low integrity variables for the executable code α.

$\forall v \in UV_\alpha \implies v \in LV_\alpha$

This will assure all user provided input to the program will initially be considered untrusted. The notion of variables however is a high level programming language concept. To express it in more concrete form, variables need to be bound to their associated memory cells which are actually used in code execution. The notation $Mem(m)$ is used to express retrieving memory content at address m. To express retrieving the integrity level of the memory cell at address m, the notation $MemIL(m)$ is used.

An abstract link is provided between variables and their associated sizes, in terms of memory cells in Definition 4.4.7 using the abstract function $Map()$. Here

the link between the integrity level of a variable and the integrity level of the memory cells assigned to the variable is expressed. Without loss of generality, it is assumed that the compiler will produce the correct machine language code regarding the size of variables in terms of memory cells. This will set the proper integrity level for all of the associated memory cells to each variable and the proper level for the machine language instructions generated to process the variables according to their level. The latter point will become clearer during discussion of the enforcement of the integrity requirements with machine instructions in the following section. The amended definition of variable to memory mapping is as follows.

Definition 4.4.11. *Set of Variable to Memory Map* for executable α:
$MV_\alpha \stackrel{def}{=} \{Mem_v[m, m+l-1] \mid v \in V_\alpha \wedge Type(v) : \tau \wedge Size(\tau) : l \;\; \wedge \;\; \exists! \; m \in Mem \wedge MemIL_v[m, m+l-1] = IL_v\}$

In the above definition IL_v expresses the integrity level of the the variable v. For brevity, $MemIL(m) = IL_v \wedge MemIL(m+1) = IL_v \wedge \cdots \wedge MemIL(m+l-1) = IL_v$ is expressed in a range form for variable v as $MemIL_v[m, m+l-1] = IL_v$.

The sets of low and high integrity memory associated with low and high integrity variables (respectively) for the executable code α can be defined as follows.

Definition 4.4.12. *Set of Low Integrity Variable to Memory Map* for executable code α:
$MVL_\alpha \stackrel{def}{=} \{Mem_v[m, m+l-1] | Mem_v[m, m+l-1] \in MV_\alpha \wedge MemIL_v[m, m+l-1] = Low\}$

Definition 4.4.13. *Set of High Integrity Variable to Memory Map* for executable code α:
$MVH_\alpha \stackrel{def}{=} \{Mem_v[m, m+l-1] | Mem_v[m, m+l-1] \in MV_\alpha \wedge MemIL_v[m, m+l-1] = High\}$

This definition also enforces the second requirement of the integrity model which simply states that all memory cells associated with a variable will be set to the same integrity level as that variable.

So far in the classification of required memory of an executable code only variables defined by the programmer are considered. The code in execution, however, may require other memory spaces not associated with any variable within the code. To represent this notion the process memory is defined for the executable code α. To keep the definition consistent with defined sets for program variables, the same memory range notation is used for the process memory. The defined ranges for user provided variables in the formal model, however are associated with the abstract variable name. For consistency, the set of execution variables is defined to associate abstract variable names with the allocated memory during execution.

Definition 4.4.14. *Set of Execution Variables* for the executable code α:
$EV_\alpha \stackrel{def}{=} \{Mem_v[m, m + l - 1] | Abstract(v) \wedge Size(v) : l\}$

This set simply represents any memory that is required during execution of α that is not associated with any defined variable within the program. The function $Abstract()$ will assign a unique random name to such variables which is not required for any implementation, rather only for formal representations. The set of process memory for the executable code α then can be defined as a set where the elements would represent the associated memory cells with variables which are either defined by the programmer or required during execution and represented abstractly.

Definition 4.4.15. *Set of Process Memory* for the executable code α:
$PM_\alpha \stackrel{def}{=} MV_\alpha \cup EV_\alpha$

For the set of all memory space assigned to the process of the executable code α, the subsets of low integrity and high integrity are defined.

Definition 4.4.16. *Set of Low integrity Process Memory* for the executable code α:
$PML_\alpha \stackrel{def}{=} \{Mem_v[m, m+l-1] | Mem_v[m, m+l-1] \in PM_\alpha \wedge MemIL_v[m, m+l-1] = Low\}$

Definition 4.4.17. *Set of High integrity Process Memory* for the executable code α:

$PMH_\alpha \stackrel{def}{=} \{Mem_v[m, m+l-1] | Mem_v[m, m+l-1] \in PM_\alpha \wedge MemIL_v[m, m+l-1] = High\}$

The notion of setting the integrity levels of memory cells assigned to all user provided variables to low can now be expressed as the first requirement of the memory integrity model.

MIR 1. $\forall v \in UV_\alpha \implies Mem_v[m, m+l-1] \in PML_\alpha$

The second requirement, which is a logical assumption, states that all memory cells assigned to a variable are set to the integrity level of that variable. This requirement is expressed formally in the definition of the low and high integrity process memory and expressed in general as follows.

MIR 2. $\forall v, Mem_v[m, m+l-1] \in PM_\alpha : MemIL_v[m, m+l-1] = IL_v$

The two sets of low and high integrity process memory will partition the set of process memory. That is $PML_\alpha \cup PMH_\alpha = PM_\alpha$ and $PML_\alpha \cap PMH_\alpha = \emptyset$. The sets of memory associated to low and high integrity variables are subsets of the low and high integrity process memory respectively: $MVL_\alpha \subset PML_\alpha$ and $MVH_\alpha \subset PMH_\alpha$.

Before the code execution starts, all process memory that are not explicitly set to low integrity level must be set to high integrity level. The same policy must be applied for the set EV_α during execution. This shapes the third requirement of the memory integrity model.

MIR 3. $\forall v \in V_\alpha - UV_\alpha : Mem_v[m, m+l-1] \in PMH_\alpha$

It is worth mentioning that the size of program variables are determined by their types, whereas the size of the required memory during execution which is outside the programmer's control is determined by either the compiler or operating system. In either case the integrity level of this type of process memory can be set explicitly by the compiler or operating system. In formal expression this type of memory is represented by abstract variable names to keep the defined sets compatible. In

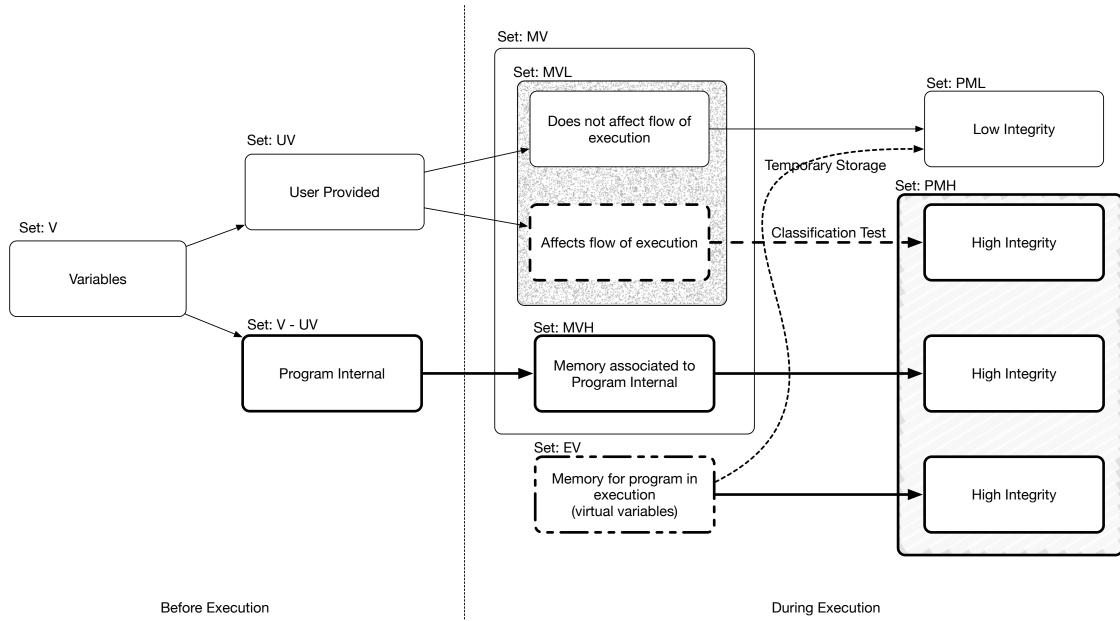

Figure 4.1: Relations between defined sets for program variables, assigned memory and process memory

other words the generated code is correct for the purpose of the execution of α regardless of the formal representation of the memory used during execution. From an implementation point of view this means that the integrity level of memory used during execution must be set to high by default. The integrity level must only change when explicitly required, for instance in case of temporary storage of a low integrity register. A diagram of the relations between defined sets can be found in Figure 4.1.

Here the general idea of a memory corruption based on Definition 4.4.8 is described. The enforcement of the integrity requirements using the added checks to machine instructions are discussed in detail in the following section. Given the abstract function $Write()$ is supposed to store the value of a variable in its associated memory cells, it will contain at least a `store` instruction in its machine code. A memory corruption can be defined as at least a `store` instruction outside the defined boundaries of a variable. The following definition formally expresses this general idea.

Definition 4.4.18. *Memory Corruption*:
$\forall v \in PM_\alpha$ where $\big(Type(v) : \tau \wedge Size(\tau) : l\big) \vee \big(Abstract(v) \wedge Size(v) : l\big)$,

for the partial program $Write(v)$ with atomic actions $a_1; a_2; \ldots; a_n$
$\exists a_k \equiv st\, r_d(w), r_s$ where $m_x = s_k.Reg(r_d) + w \notin Mem_v[m, m + l - 1]$

It is assumed that $m_x \in PM_\alpha$, as the process address space can be tightly controlled with virtual addressing or segmentation. Based on the definition, if $m_x \notin Mem_v[m, m + l - 1]$ then m_x would belong to another range such as $m_x \in Mem_u[m', m' + l' - 1]$ associated with the variable u (defined by the programmer or abstractly) and will have the following cases: (i) $u \in PMH_\alpha$; or (ii) $u \in PML_\alpha$. That is the integrity level of the memory cell pointed to by m_x is either set to high or it is set to low. The corruption is done using user provided input. This means the variable v will have a low level of integrity. The integrity level of the memory cells assigned to the variable v will consequently be set to low. If the integrity level of the memory cell pointed to by m_x is set to high a mismatch of integrity will occur. This is due to the $Write(v)$ program performing low integrity memory writes for the low integrity variable v. It will be shown in the following section that the atomic `store` instruction(s) belonging to the $Write(v)$ program can only access low integrity memory cells.

If m_x points to an address outside the associated range for variable v but the integrity level of the memory cell is set to low, the corruption will occur. This however is inconsequential for a control flow attack as all memory cells that will affect the flow of execution are set to high integrity or have to pass an explicit test. This means corruption of low integrity memory cells will either corrupt a cell that has no effect on flow of execution or will be tested before being elevated to high integrity. There must be no opportunity to corrupt a low integrity level variable that has passed its classification test. The elevation of the associated memory cells must be done before reading another variable from the user. This forms the fourth requirement of the integrity model.

MIR 4a. $\forall v_1, v_2 \in UV_\alpha \wedge \exists(\varphi, v_1) \in CT_\alpha \wedge Classify(v_1) : a_1; a_2; \ldots; a_n$

1. "if $\varphi(v_1)$ then $Classify(v_1)$ else halt()" $\stackrel{C}{\to} \Big(\big(?\varphi(v_1); Classify(v_1)\big) \cup \big(?\neg\varphi(v_1); halt\big) \Big)$

2. $\forall a_k \in Classify(v_1) \implies a_k \notin Read_{IO}(v_2)$

The $Read_{IO}(v)$ formally defined in Definition 4.4.9 is a program that reads the user provided value from Mem_{IO}, and stores it in its corresponding location in memory for a variable v. $Classify(v)$ is a program that represents the change of the integrity level of the variable v. The requirement states that for any two user provided variables v_1 and v_2 if there exists a classification test for variable v_1 such as $\varphi(v_1)$ and it passes the test successfully expressed as "$\varphi(v_1) = True$", then the transition of the integrity level of the variable v_1 happens before reading the variable v_2 from the user. This is demonstrated using the notation $\stackrel{C}{\rightarrow}$ that expresses the notion of "is compiled to". The requirement is that the classification test is enforced using the high level programming construct `if-then-else` for the variable v_1 and it is translated to the given PDL formula. The formula then is constructed using the test and sequence operator as $?\varphi(v_1); Classify(v_1)$ expressing that there are no other instructions between the action sequence of the classification test and the action sequence of the classification of the variable. This is reinforced by the choice operator conditioned on unsuccessful test followed by the `halt` instruction. The second requirement expresses that for all atomic instructions of the program $Classify(v_1)$, none belong to the program $Read_{IO}(v_2)$. This is required to remove the opportunity of corruption that may be available by reading the second variable before elevating the first variable that has passed its associated classification test. After defining the machine instructions, the requirement above will be discussed in more detail (Section 4.4.5).

The last requirement of the model states that the compiler will produce instructions according to the integrity level of the memory or registers involved. That is each instruction has the capability of enforcing the requirements of the memory integrity model and the compiler will use the instructions accordingly. This will assure that low integrity and high integrity content will not be mixed together without explicit instruction(s) to increase or decrease the integrity level when necessary and according to the rules preventing user-controlled corruption of high integrity memory.

The general PDL expression in MIR 5 specifies a test φ that must be satisfied for successful execution of the instruction and the logical statement p that must be satisfied in all possible executions of the provided action sequence $(?\varphi; a_k) \cup (?\neg\varphi; halt)$. In all cases the p statement only expresses the condition related to the rules of integrity model that must be satisfied when the test $\varphi \equiv \top$. The case where $\varphi \equiv \bot$ and logical statements regarding the semantics of the instructions are omitted. The expressions $\varphi \equiv \top$ and $p \equiv \top$ express that the instruction has no precondition or postcondition for memory integrity to satisfy.

MIR 5. $\forall a_k \in \alpha \implies \big[(?\varphi; a_k) \cup (?\neg\varphi; halt)\big]p$

- $a_k = (nop \vee jd\, w \vee movi\, r_d, w \vee halt) \implies \varphi \equiv \top \wedge p \equiv \top$

  The no operation, jump direct, move immediate operand, and halt instructions do not require any verification for memory integrity.

- $a_k = add\, r_d, r_s, r_t \implies \varphi \equiv s_k.RegIL(r_d) = s_k.RegIL(r_s) = s_k.RegIL(r_t)$

- $a_k = addi\, r_d, r_s, w \implies \varphi \equiv s_k.RegIL(r_d) = s_k.RegIL(r_s)$

  The arithmetic and logic operations can be done on registers with same integrity levels. The result can only be stored in a register with the same integrity level.

- $a_k = ld\ r_d, r_s(w) \implies \varphi \equiv s_k.RegIL(r_s) = H \wedge p \equiv s_{k+1}.RegIL(r_d) = s_k.MemIL(s_k.Reg(r_s) + w)$

  The integrity level of the register that receives the content of a memory cell will be set to the integrity level of that memory cell. The integrity level of the register used for address calculation will be high.

- $a_k = st\ r_d(w), r_s \implies \varphi \equiv s_k.RegIL(r_d) = H \wedge s_k.RegIL(r_s) = s_k.MemIL(s_k.Reg(r_d) + w) \wedge p \equiv \top$

  The `store` *instruction must verify that the target memory location has the same integrity level of the register that is being stored. The compiler will explicitly set the integrity level of the target memory before executing this instruction in the case of temporary storage, or it will be the same if the store*

logically follows an earlier load. The register used for address calculation must have a high integrity level.

- $a_k = bgt\, r_s, r_t, w \implies \varphi \equiv s_k.RegIL(r_s) = H \wedge s_k.RegIL(r_t) = H \wedge p \equiv \top$

  The conditional branch instruction affects the flow of execution based on the content of two registers. The integrity levels of these registers must be high.

- $a_k = (jmp\, r_s \vee ret\, r_s) \implies \varphi \equiv s_k.RegIL(r_s) = H \wedge p \equiv \top$

  The indirect jump and return instructions use the content of a register as the target address, hence the integrity level of the register must be high.

A more detailed expression of the last requirement for each instruction will be provided in the following section. New instructions that are necessary to explicitly change the integrity level of memory cells or registers are also discussed.

The informal summary of the requirements can be stated as follows:

1. All user provided variables must be declared explicitly by the programmer and the integrity level of the assigned memory cells to these variables must be set to low.

2. The integrity levels of all memory cells associated with a variable will be set to the integrity level of that variable.

3. The integrity levels of all other memory cells used in the program must be set to high, except memory cells used for temporary storage of low integrity level variables or registers during execution.

4. If a low integrity variable will be used in a decision that will influence the flow of control in any way, it must have a classification test and be elevated as soon as it passes the test (before reading the next variable from the user).

5. The part of the code dealing with low integrity operations, will use low integrity instructions operating on registers and memory locations set to low integrity

levels. This is enforced as part of the semantic of the atomic instructions and by appropriate instruction generation of the compiler.

The enforcement of the last two requirements through machine instructions is discussed in the following section.

4.4.4 Instructions Enforcing Memory Integrity Model

The main goal of memory integrity model is to limit what can be corrupted by the attacker to the user provided values that have no effect on the flow of execution and are already under the user's or adversary's control. This will ultimately make the attack ineffective. This is achieved in three main steps. The first step is to classify memory into two broad classes and mark the memory cells accordingly. The second step is to define rules for the flow of information from the lower category to the higher. The third step is to enforce the defined rules at atomic instruction execution to prevent unauthorised flow of information. The previous sections have discussed the first two steps. This section deliberates more on the requirements of each atomic instruction, which falls under the third step of achieving the goal of the memory integrity model.

Before discussing the requirements of instructions in more detail, a summary of the notation used to express the semantics of each instruction is provided. To represent the register content the notation $Reg(r_{Regnum})$ is used, which is the same as the notation in the previous chapter. $RegIL(r_{Regnum})$ is used to represent the the integrity level of a register. Similarly $Mem(x)$ is used for memory content at address x; $MemIL(x)$ is used for the memory integrity of the cell at address x. The notation summary can be found in Table 4.1.

The instruction set prior to the required checks of the integrity model from the previous chapter is summarised in Table 4.2. For clarity the preconditions of Ideal Control Flow Integrity (ICFI) will be removed in the remainder of this chapter. The enforcements in atomic instructions related to ICFI will be simply referred to as ICFI Conditions.

Table 4.1: Notation summary

Notation	Semantic
$\leftarrow$	Assignment as $target \leftarrow value$
$Mem(x)$	Content of memory at address x
$Mem_v[m, m+l-1]$	Memory cells assigned to variable v of size l starting at address m
$MemIL(x)$	Integrity Level of memory at address x
$Reg(r_x)$	Content of register r_x in the register file
$RegIL(r_x)$	Integrity Level of register r_x in the register file
IL_v	Integrity Level of variable v
pc	Content of the program counter
$\in_{emb}$	Embedded as an immediate operand as $w \in_{emb} i_x$
dot / .	Partial element of the state e.g. $s.pc$: content of pc in state s
s	Current state
s'	Next state (after execution of the current instruction)

Since the focus of the integrity model is on the flow of information and there are various instructions that have no effect on memory or registers, these instructions can be excluded from the discussion. These instructions are: `nop`, `halt`, and `jd` which do not require any changes as these instructions do not change the content of memory or any register in the register file. The jump direct instruction uses an immediate operand as an address where a successful exploitation would require corruption of the code memory. As the corruption would need a memory write to code memory it can be prevented either under the read only memory for code, the memory integrity model, or as discussed in the previous chapter under the integrity precondition of the executable code for ICFI. The `movi` instruction can change the content of a register, however as the operand and the register are both embedded in the instruction itself and similar to direct jump cannot be changed during execution with read only memory or integrity assumption of the code, this instruction does not require any change under the integrity model. To distinguish the instructions that require additional verification under the memory integrity model from their previous version, the first letter of the instruction is changed to a capital letter. For instance the new `load` instruction will be expressed as $Ld\, r_d, r_s(w)$ which will be constructed

Table 4.2: Summary of instruction semantics

Instruction Semantics
$[nop]p_1$ $p_1 \equiv s'.pc = s.pc + 1$
$[add\, r_d, r_s, r_t]p_1 \wedge p_2$ $p_1 \equiv s'.Reg(r_d) = s.Reg(r_s) + s.Reg(r_t)$ $p_2 \equiv s'.pc = s.pc + 1$
$[addi\, r_d, r_s, w]p_1 \wedge p_2$ $p_1 \equiv s'.Reg(r_d) = s.Reg(r_s) + w$ $p_2 \equiv s'.pc = s.pc + 1$
$[movi\, r_d, w]p_1 \wedge p_2$ $p_1 \equiv s'.Reg(r_d) = w$ $p_2 \equiv s'.pc = s.pc + 1$
$[ld\, r_d, r_s(w)]p_1 \wedge p_2$ $p_1 \equiv s'.Reg(r_d) = s.Mem(s.Reg(r_s) + w)$ $p_2 \equiv s'.pc = s.pc + 1$
$[st\, r_d(w), r_s]p_1 \wedge p_2$ $p_1 \equiv s'.Mem(s.Reg(r_d) + w) = s.Reg(r_s)$ $p_2 \equiv s'.pc = s.pc + 1$
$[jd\, w]p_1$ $p_1 \equiv s'.pc = w$
$bgt\, r_s, r_t, w \equiv [(?\varphi; jd\, w) \;\cup\; (?\neg\varphi; nop)]p_1$ $p_1 \equiv s'.pc = w \vee s'.pc = s.pc + 1$ $\varphi \equiv s.Reg(r_s) > s.Reg(r_t)$
$[(?\varphi; jmp\, r_s) \cup (?\neg\varphi; halt)](p_1 \wedge p_2) \vee \bot$ $\varphi \equiv$ ICFI Condition $((s.pc, s.Reg(r_s)) \in AC_{\alpha \wedge \alpha'}$ $p_1 \equiv s'.pc = s.Reg(r_s)$ $p_2 \equiv$ ICFI Condition $(s'.RM_\alpha = s.RM_\alpha + (s.pc + 1, s.Reg(r_s), rp_{1...m}))$
$[(?\varphi; ret\, r_s) \cup (?\neg\varphi; halt)](p_1 \wedge p_2) \vee \bot$ $\varphi \equiv$ ICFI Condition $((s.Reg(r_s), f_j, rp^{f_j}_{1...m}) \in s.RM_\alpha$ $p_1 \equiv s'.pc = s.Reg(r_s)$ $p_2 \equiv$ ICFI Condition $(s'.RM_\alpha = s.RM_\alpha - (s.Reg(r_s), f_j, rp^{f_j}_{1...m}))$
$[halt]p_1$ $p_1 \equiv \bot$

using its previous semantic with added verifications. This will also demonstrate the relation and changes between the two versions of these instructions.

Considering the integrity model at its core is dealing with the content of memory and registers, instructions that update the content of a memory cell or register are considered first. The required rules for `load` and `store` instructions can be defined as follows.

Definition 4.4.19. *Load Instruction*:

$Ld\, r_d, r_s(w) \equiv \Big[\big(?\varphi; ld\, r_d, r_s(w)\big) \ \cup \ \big(?\neg\varphi; halt\big)\Big](p_1 \wedge p_2 \wedge p_3) \vee \bot$ such that:

- $\varphi \equiv s.RegIL(r_s) = H$
- $p_1 \equiv s'.RegIL(r_d) = s.MemIL(s.Reg(r_s) + w)$
- $p_2 \equiv s'.Reg(r_d) = s.Mem(s.Reg(r_s) + w)$
- $p_3 \equiv s'.pc = s.pc + 1$

The requirement for the new `load` instruction is that the integrity level of the register used for the address of the memory cell must be high. Since the target register would represent the content of the addressed memory cell, it will be set to the same integrity level as the source memory cell. The integrity requirement of the address is enforced by specifying condition φ in the PDL expression of the instruction that when True the `ld` instruction will be executed and the `halt` instruction otherwise. This is expressed as $\varphi \equiv s.RegIL(r_s) = H$ which specifies the integrity level of register r_s used in address calculation, as $Reg(r_s) + w$ must be high. This is due to the fact that the integrity level of all memory addresses must be high regardless of the integrity level of the cell they are pointing to. That is all memory addresses are considered as the program internal state and must never be controlled by the user. The outcome of the execution of the `load` instruction is expressed as the three logical statements p_1, p_2, and p_3. The first statement expresses that in the new machine state the integrity level of the destination register r_d is set to the same level as the the memory cell at address $Reg(r_s) + w$, where w represents the offset provided as an embedded value within the `load` instruction itself. The second logical predicate expresses the semantic of the `load` instruction. In the new machine state the destination register will contain the content of the memory cell addressed as $Reg(r_s)+w$. The last logical outcome of the `load` instruction in the new machine state is that the program counter is incremented by one.

The `store` instruction writes the content of a register into memory, and from this aspect it would provide the opportunity for memory corruption.

Definition 4.4.20. *Store Instruction*:

$St\, r_d(w), r_s \equiv \Big[\big(?\varphi; st\, r_d(w), r_s\big) \;\cup\; \big(?\neg\varphi; halt\big)\Big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.MemIL(s.Reg(r_d) + w) = s.RegIL(r_s) \wedge s.RegIL(r_d) = H$
- $p_1 \equiv s'.Mem(s.Reg(r_d) + w) = s.Reg(r_s)$
- $p_2 \equiv s'.pc = s.pc + 1$

Similarly for the new `store` instruction, the integrity level of the register used in address calculation of the target memory must be high. The `store` instruction differs from the `load` in the sense that the destination register in `load` always represents the memory cell and must also represent its integrity level, whereas in `store` there are circumstances that must be checked whether the integrity levels of the register and the target memory match to assure that high integrity memory is not corrupted by low integrity content. This can be enforced by adding another condition to the `store` instruction that verifies whether the source register and the target memory cell have the same integrity level. This is expressed in PDL as a choice between the sequence of a test φ and `st` instructions, and the sequence of the test $?\neg\varphi$ and `halt` instructions. The φ condition then specifies two requirements for a successful execution of the `store` instruction. The first defined as $s.MemIL(s.Reg(r_d) + w) = s.RegIL(r_s)$ expresses the condition that the integrity level of the memory cell at address $Reg(r_d) + w$ and the integrity level of the r_s register are equal. The second expressed as $s.RegIL(r_d) = H$ enforces the requirement of high integrity levels for address calculation. The outcome of the successful execution of the `store` instruction is expressed as the two logical statements p_1 and p_2. The first statement expresses the semantic of the `store` instruction as $s'.Mem(s.Reg(r_d) + w) = s.Reg(r_s)$. This specifies that in a new machine state the memory cell at the address calculated as the content of r_d plus the embedded value w will contain the content of the register r_s. The second statement simply specifies that the program counter will be incremented by one in the new machine state.

More complex instructions such as `push` and `pop`, which are not discussed in the model, can be constructed using the `load` and `store` instructions, reserved registers for stack pointers, and an instruction that unconditionally sets the integrity level of the memory cell at the top of the stack according to the integrity level of the register that is being pushed. The unconditional setting of the memory cell to low integrity only applies when the target memory cell is used as temporary storage and is guaranteed to be free. This can be achieved by limiting the required memory writes for user provided input to one variable at a time and setting the associated memory to low integrity, as discussed in the previous section. Depending on the architecture, the logic used to deal with function calls may differ. One strategy is to use the current stack frame to store the content of the register file. The integrity of the stack registers can be guaranteed by setting the integrity level of these registers to high. This will assure that when these registers are pushed to the top of the stack, the integrity level of the assigned memory cells will be set to high. As the attacker will only have access to low integrity memory cells the stack pointers cannot be corrupted. When the stack pointers can be trusted then the `push` and `pop` instructions can be trusted to explicitly set the integrity level of the memory cell pointed to by the stack pointers according to the integrity level of the register that is being pushed to the top of the stack. The `pop` instruction would behave similar to `load`, as the target register will be overwritten in process of restoring a previous state of the program in execution. A stack buffer overflow attack takes advantage of the vulnerabilities that involve the `store` instruction and an incorrect memory address (base or offset) pointing to memory cells in the stack rather than `push` and `pop` instructions. In this scenario a `store` instruction for a low integrity register cannot be executed if the integrity level of the memory cell with the calculated target address is high. This will stop an unchecked `store` instruction within a loop as soon as the instruction tries to perform a write into a memory cell with the high integrity level. The corruption then will be limited to memory cells set to low integrity. This will only include memory associated with user provided

variables that have not been read from the user or do not influence the flow of execution.

The conditional branch instruction affects the flow of execution based on a condition, hence it can only use registers that are set to the high integrity level as the condition of the branch. This will assure that low integrity content cannot affect the flow of execution. For user provided values to be used in changing the flow of execution, they must first pass the classification test and be changed to high integrity. This is discussed as part of the conditional change of integrity levels of registers and memory cells further in this chapter.

Definition 4.4.21. *Branch Greater Than Instruction*:
$Bgt\, r_s, r_t, w \equiv \left[\Big(?\varphi_1; \big((?\varphi_2; jdw) \cup (?\neg\varphi_2; nop)\big)\Big) \cup \Big(?\neg\varphi_1; halt\Big)\right] p_1 \vee \bot$ such that:

- $\varphi_1 \equiv s.RegIL(r_s) = H \wedge s.RegIL(r_t) = H$
- $\varphi_2 \equiv s.Reg(r_s) > s.Reg(r_t)$
- $p_1 \equiv s'.pc = w \vee s'.pc = s.pc + 1$

The requirement of the conditional branch is enforced by constructing the `Bgt` instruction as a choice between the sequence of a test φ_1 and the semantic of the conditional branch and the complement of the test and `halt` instruction. The test φ_1 verifies whether the integrity levels of both registers used in deciding the flow of execution are high. If this test fails the execution will halt. In case that the test φ_1 succeeds the test φ_2 combined with a choice decide the flow of execution. If φ_2 succeeds, then the direct jump instruction `jd` with the embedded address w is executed; if it fails, the no operation instruction `nop` will be executed. The successful execution of the conditional branch is expressed as logical statement p_1, where in the new machine state the program counter is either containing the address w as embedded in the `Bgt` instruction or is incremented by one.

Indirect jump and return instructions use the content of a register as an address to jump or return to, and clearly affect the flow of execution based on a run-time value. This value must also be of high integrity and outside the reach of corruption

by user provided input. This is achieved by requiring both indirect jump and return instructions to use only registers set to high integrity.

Definition 4.4.22. *Indirect Jump Instruction*:
$Jmp\, r_s \equiv \big[(?\varphi; jmp\, r_s) \;\cup\; (?\neg\varphi; halt)\big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.RegIL(r_s) = H \wedge ICFIconditions$
- $p_1 \equiv s'.pc = s.Reg(r_s)$
- $p_2 \equiv$ *ICFI conditions*

For indirect jump this is shown in PDL as a sequence of a test φ and the `jmp` instruction or the sequence of the test $\neg\varphi$ and the `halt` instruction. The test is constructed using the condition that the integrity level of the register used in address calculation, r_s, must be high. The ICFI conditions can be enforced in a similar manner and are discussed in detail in the previous chapter. The program counter in the new machine state will contain the content of the register r_s after successful execution of the indirect jump instruction. Similarly, to enforce the ICFI requirements the ICFI conditions can be expressed as part of the semantic of the indirect jump, which has been discussed in detail in the previous chapter.

The return instruction, similar to indirect jump instruction can only use a high integrity register for return address.

Definition 4.4.23. *Return Instruction*:
$Ret\, r_s \equiv \big[(?\varphi; ret\, r_s) \;\cup\; (?\neg\varphi; halt)\big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.RegIL(r_s) = H \wedge ICFIconditions$
- $p_1 \equiv s'.pc = s.Reg(r_s)$
- $p_2 \equiv$ *ICFI conditions*

These two instructions are similar in the context of the memory integrity model however differ in the context of ICFI.

To avoid accidental mixture of low integrity content with high integrity content which could lead to memory corruption, the arithmetic operations can only be performed on registers with the same integrity level.

Definition 4.4.24. *Add Instruction*:
$Add\, r_d, r_s, r_t \equiv \big[(?\varphi; add\, r_d, r_s, r_t) \ \cup \ (?\neg\varphi; halt)\big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.RegIL(r_d) = s.RegIL(r_s) = s.RegIL(r_t)$
- $p_1 \equiv s'.Reg(r_d) = s.Reg(r_s) + s.Reg(r_t)$
- $p_2 \equiv s'.pc = s.pc + 1$

To enforce the aforementioned requirement, the `Add` and `Addi` instructions are both expressed as a sequence of a test and the corresponding instruction, or the complement of the same test and the `halt` instruction. In the `Add` instruction the involved registers must all be of the same integrity level. This is stated in PDL as the logical statement $s.RegIL(r_d) = s.RegIL(r_s) = s.RegIL(r_t)$ specifying that registers r_d, r_s and r_t, all have the same integrity level.

The outcome from the add instruction is defined using two logical statements expressing the fact that in the new machine state the register r_d contains the result of addition between the content of registers r_s and r_t, and the fact that the program counter is incremented by one.

Definition 4.4.25. *Add Immediate Instruction*:
$Addi\, r_d, r_s, w \equiv \big[(?\varphi; addi\, r_d, r_s, w) \ \cup \ (?\neg\varphi; halt)\big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.RegIL(r_d) = s.RegIL(r_s)$
- $p_1 \equiv s'.Reg(r_d) = s.Reg(r_s) + w$
- $p_2 \equiv s'.pc = s.pc + 1$

The `Addi` instruction differs from `Add` instruction in that it only requires two registers and the test is expressed as: $s.RegIL(r_d) = s.RegIL(r_s)$ for these two involved registers. The outcome is similarly expressed as the logical statements

specifying the content of the register r_d and program counter in the new machine state.

To allow increase or decrease of the integrity level when needed, according to the locality and logic of the generated executable code, instructions that can change the integrity level of registers or memory cells are needed. First the unconditional instructions to change the integrity level of a register or a memory cell are defined. Then the various requirements and conditions that must be met to avoid abuse of these instructions in memory corruption are described.

Definition 4.4.26. *Change Register Integrity Level*:
$[Intreg\ r_t, il]p_1 \wedge p_2$ such that:

- $p_1 \equiv s'.RegIL(r_t) = il$
- $p_2 \equiv s'.pc = s.pc + 1$

The outcome of the execution of this instruction is defined as two logical statements. The first statement specifies that the integrity level of the register r_t in new machine state will be equal to the embedded integrity level in the instruction il. The second statement expresses that the program counter is incremented by one in the new machine state.

Similarly, to change the integrity level of the memory at address $Reg(r_d) + w$ the memory instruction can be used. Although this instruction is a new instruction under the memory integrity model, $intmem\ r_d(w), il$ is used to express the semantic of the instruction without the required verification of the target memory address to build up the instruction that is intended. The intermediate expression $intmem\ r_d(w), il$ will not be a valid instruction by itself and will not be used in code generation.

Definition 4.4.27. *Change Memory Integrity Level*:
$Intmem\ r_d(w), il \equiv \Big[\big(?\varphi; intmem\ r_d(w), il\big) \cup \big(?\neg\varphi; halt\big)\Big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.RegIL(r_d) = H$

- $p_1 \equiv s'.MemIL(s.Reg(r_d) + w) = il$
- $p_2 \equiv s'.pc = s.pc + 1$

The memory version of changing an integrity level instruction uses a memory address. All memory addresses must use high integrity registers in address calculation. The instruction is constructed as a sequence of a test φ and an unconditional change of the integrity level of the memory cell at address $Reg(r_d) + w$, or the sequence of the complement of the test and the `halt` instruction. The result of a successful execution of the instruction is expressed as two logical statements specifying that in the new machine state the integrity level of the memory cell addressed as $Reg(r_d) + w$ will be set as il, and the program counter will be incremented by one.

Given the conditional branch instruction can only use registers with high integrity levels and the need to perform a test on user provided input, it must be considered how to define a condition which would allow the transition of the integrity level from low to high for user provided input. This can be achieved by conditioning the elevation of the integrity level of a memory cell on two essential elements of a classification test for a variable. The first essential element is the user provided value itself or some derived value from it, for instance the length or the type. This first element can be of low integrity. The second essential element is a high integrity value which represents the expected value or an acceptable value. The second element will protect the integrity of the test itself by assuring that the adversary can only control one of the two elements in any classification test and that the element is always tested against a high integrity element. The high integrity element can be an elevated value which means it has passed its own specific test and now can be trusted. The result of a successful test would be the elevation of the associated memory cell to this test. This will allow the construction of more complex tests that could verify and increase the integrity level of multiple elements in more complex expressions. Although this instruction can combine the classification test of a variable and the classification of the memory cell associated to it, both

$\varphi(v) \in CT_\alpha$ and $Classify(v)$ are considered as programs with certain requirements that will be discussed shortly.

Definition 4.4.28. *Conditional Set Memory Integrity Level*:
$Cintmem\ r_d(w), r_s, r_t \equiv \Big[\big(?\varphi; intmem\ r_d(w), H\big) \ \cup \ \big(?\neg\varphi; halt\big)\Big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.Reg(r_t) < s.Reg(r_s) \wedge s.RegIL(r_s) = H \wedge s.RegIL(r_d) = H$
- $p_1 \equiv s'.MemIL(s.Reg(r_d) + w) = H$
- $p_2 \equiv s'.pc = s.pc + 1$

The three aforementioned elements of the conditional change of integrity level for a memory cell are integrated in the construction of the instruction as part of the test φ and the result of successful execution of the instruction in the logical statement p_1. As the instruction is only used to elevate the integrity level, there is no need to provide the integrity level as an immediate operand within the instruction itself. The instruction is represented in PDL as the sequence of a test φ and the unconditional change of the integrity level of the memory cell addressed by $s.Reg(r_d) + w$ to high; or the sequence of the complement of the test and the instruction `halt`. This assures that the violation of the test will stop the execution. The test φ then is comprised of three logical statements that must be true for the test to be successful. The first statement specifies the comparison of the low integrity element of the classification test, register r_t, with the high integrity element, register r_s. The second statement assures that the high integrity element is verified by checking the integrity level of the register r_s. The third statement assures that the integrity level of register r_d, which would contain a memory address base, is high. The result of the successful execution of the instruction is defined as two logical statements expressing the elevation of the integrity level of the memory addressed by $Reg(r_d) + w$ and the increment of the program counter in the new machine state.

For completeness the conditional change of integrity level is provided for a register as well. The conditional set register integrity instruction is similar to its memory

counterpart, with the exception that the result of a successful test would be the elevation of a register.

Definition 4.4.29. *Conditional Set Register Integrity Level*:
$Cintreg\ r_d, r_s, r_t \equiv \big[(?\varphi; Intreg\ r_d, H)\ \cup\ (?\neg\varphi; halt)\big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.Reg(r_t) < s.Reg(r_s) \wedge s.RegIL(r_s) = H$
- $p_1 \equiv s'.RegIL(r_d) = H$
- $p_2 \equiv s'.pc = s.pc + 1$

This instruction for instance can be used for optimisation in case of simple data structures of the maximum length of a register. In this scenario, a user provided variable is read from input/output memory into a register. A classification test is performed to elevate the integrity level of the register. The register then will be used in a conditional branch instruction and the integrity level of the register is set back to low for repeating this procedure. Rather than elevating the memory cell to high before the branch and lowering it after, the test can be performed on the copy of the variable's value in a register. The copy can be used as a representative of the variable in the conditional branch instruction. The generated code will be correct if the memory cell will contain the final value of the variable and the integrity level of the memory cell is set to the appropriate value. The optimisation however is outside the scope of this thesis.

4.4.5 A Discussion on Compiler Requirements

Having defined the requirements of the instructions for the memory integrity model, the potential avenue of attack and the sequences of instructions that must be avoided by the compiler can be discussed in more detail. The unconditional change of memory integrity or register instructions combined with `store` have the capability to overwrite high integrity memory locations. To avoid abuse of these instructions as part of an exploitation, their combined use must be tightly controlled by the

compiler. Change of the integrity level of registers and memory hence can be done conditionally or unconditionally, depending on the locality and logic of the program to remove the opportunity of exploitation by the adversary. More precisely, the elevation of the integrity level must be limited to a specific part of the program and separate from the parts where the user and therefore the adversary have write access to memory. This is justified, as lowering the integrity level of memory or registers will not lead to memory corruption that could change the flow of execution as all registers used in address calculation require high integrity levels as part of their semantics, as well as the conditional branch instruction.

The partial programs $Map(v)$ and $Classify(v)$ require the change of integrity level for memory cells either unconditionally or conditionally. These programs may also use the `load` and `store` instructions. The $Read_{IO}(v)$ program transfers the user provided input from IO memory to memory cells assigned to variables. This program also uses the `load` and `store` instructions. By separating the classification of the associated memory cells from the write operation, the memory corruption will be limited to programming errors where the corruption cannot be controlled by the attacker. Otherwise it is part of the benign execution of the program. That is, it will happen with valid user input and cannot be considered as exploitation. The goal is to remove the opportunity of corruption from each of these partial programs.

The $Map(v)$ program uses unconditional change of integrity level instructions to set up the integrity levels of the associated memory cells for all variables. As this program is executed before the program α, it is outside the reach of the adversary since its execution occurs before any of the $Read_{IO}(v)$ programs for $\forall v \in UV_{\alpha}$. This can be formally expressed by considering a high level program A comprised of high level functions $f_i \in A \vee lib_j$ for $j = 1, \ldots, n$. Each function is compiled to a corresponding machine code comprised of (i) a function prologue; (ii) an equivalent machine code of the function; and (iii) a function epilogue. This can be formally expressed as the last memory integrity requirement as follows.

MIR 6. $\forall f_i \in (A \vee lib_j), i = 1, \ldots, n_1, j = 1, \ldots, n_2$ *then*
$f_i \stackrel{C}{\rightarrow} (\beta_i; \alpha_i; \gamma_i) \implies \forall v_{ik} \in f_i, Map(v_{ik}) \in \beta_i, k = 1, \ldots, n_3$

The notation $\stackrel{C}{\rightarrow}$ expresses the compilation process from a high level function to low level machine instructions. The action sequence β_i is the prologue for the function f_i. The action sequence α_i is the equivalent machine code of the function f_i. The action sequence γ_i is the epilogue for the function f_i. The prologue code, β_i, will perform all the required preparations for the execution of the equivalent machine code α_i for function f_i, including but not limited to associating required memory for the local variables. The function prologue can perform other tasks such as setting up the new stack frame for the called function. In this case the $Map(v)$ will be part of the prologue of function f_i where the variable v is a local variable in that function. The epilogue of a function then can perform any required clean up task, for instance resetting the integrity level of any necessary memory location that was set up by the function prologue. The MIR 6 states that the association of the memory cells to variables are done before the execution of the code starts. The requirement also expresses that the association of variables does not need to be done all at once for all variables of the program, it can be done per function as the execution progresses. However, for every function the association of memory cells to local variables is done before the execution of the equivalent machine code of that function starts.

Once $Map(v)$ sets the appropriate integrity level for the associated memory cells for each variable $v \in V_\alpha$, during the execution of the program α, the partial programs $Read_{IO}(v)$ for $v \in UV_\alpha$ will read the user provided values and store them in associated memory. An abstract definition for the $Read_{IO}(v)$ has been provided in Definition 4.4.9. Now the requirement that the $Read_{IO}(v)$ program not perform any conditional or unconditional change of integrity level instruction can be added. This can be expressed by amending the definition of $Read_{IO}(v)$ as follows:

Definition 4.4.30. For the program $Read_{IO}(v)$ comprised of atomic actions $a_1; a_2; \ldots; a_n$ where $v \in UV_\alpha$ then $\forall a_k \in Read_{IO}(v)$:

- $a_k = Ld\ r_d, r_s(w) \implies \big(s_k.Reg(r_s) + w \in Mem_{IO}[m_{io}, m_{io} + l - 1] \wedge MemIL[m_{io}, m_{io} + l - 1] = L\big)$
- $a_k = St\ r_d(w), r_s \implies s_k.RegIL(r_s) = L$
- $a_k \neq \big(Intreg\ r_t, il \vee Cintreg\ r_d, r_s, r_t \vee Intmem\ r_d(w), il \vee Cintmem\ r_d(w), r_s, r_t\big)$

The first requirement of the program $Read_{IO}(v)$ is that the integrity levels of all memory cells in the Memory IO range are set to low. Consequently all the required `load` instructions will set the temporary registers, used in transferring the user provided input from Memory IO, to associated memory cells to low integrity. The second requirement states that all `store` instructions will perform memory writes using a low integrity register as the source. The third requirement is that none of the conditional or unconditional changes of register or memory integrity level instructions are used in the $Read_{IO}(v)$ program.

When dealing with user provided variables, the change of integrity level from low to high can only happen after passing the corresponding classification test and will happen using the corresponding $Classify()$ program for that variable. The $Classify(v)$ program will control the use of conditional or unconditional change of memory integrity instructions based on the the starting address and the size of the variable v. Since the $Classify(v)$ program is only concerned with the integrity level of the variable v not its content, it does not require any `store` instruction. The registers are temporary storage and the `load` instruction keeps the integrity level of the variables intact when stored in registers. As a general rule, the $Classify(v)$ program does not require any change of register integrity as it elevates the integrity level of a variable by changing the integrity level of the associated memory cells. There may be exceptions for this rule for optimisation purposes which is outside the scope of this model. An example would be for simple data structures that can be stored in a register. The classification test and consequent elevation can all be done using the copy of the variable in the register where the variable will be declassified

right after participating in a conditional branch, for instance to be reread from the user.

This can be specified as rules for the compiler and formally expressed as follows: For the program $Classify(v)$ comprised of atomic actions $a_1; a_2; \ldots ; a_n$, generated by compiler, $\forall a_k \in Classify(v),\ a_k \neq (Intreg\ r_t, H \vee Cintreg\ r_d, r_s, r_t)$. This can be added to the fourth requirement of the memory integrity model. The fourth requirement has stated that for any two user provided variables v_1 and v_2, if v_1 has a classification test and has been provided by the user then the classification must happen before reading another user provided variable such as v_2. The requirement was expressed in two conditions for any two variables v_1 and $v_2 \in V_\alpha$ (MIR 4a). Given the corruption opportunity exists during $Read_{IO}(v_2)$ and can be achieved using the `store` instruction, the requirement can state that the $Classify(v_1)$ does not contain any low integrity `store` instruction. Adding the rule for setting the register integrity level for $Classify(v)$ the requirement can be amended as follows.

MIR 4b. $\forall v \in UV_\alpha \wedge \exists(\varphi, v) \in CT_\alpha$ then:

1. "if $\varphi(v)$ then $Classify(v)$ else halt()" $\overset{C}{\to} \Big(\big(?\varphi(v); Classify(v)\big) \cup \big(?\neg\varphi(v); halt\big) \Big)$

2. $Classify(v) : a_1; a_2; \ldots ; a_n \implies \forall a_k \in Classify(v)$:

    - $a_k \neq St\, r_d(w), r_s \wedge s_k.RegIL(r_s) = L$
    - $a_k \neq (Intreg\ r_t, H \vee Cintreg\ r_d, r_s, r_t)$

The first requirement expresses that the high level programming construct "if $\varphi(v)$ then $Classify(v)$ else halt()" is compiled to a choice dependant on the classification test without any other actions between the test and the classification action sequence.

One more condition must be specified for the $Classify(v)$. That is the change of memory integrity will not be performed outside the associated range of memory cells to variable v. This is assured as all registers used in address calculation, in this case r_d in $Intmem\ r_d(w), H$ instruction, must have the high integrity level and cannot be directly affected by user provided input. The $Read_{IO}(v)$ program for any variable v

cannot perform any high integrity memory write. Given the ICFI requirements are in place, the code of the program cannot be modified by the adversary to change the value of the embedded offset w. The code can be equally protected by setting the integrity level of the code memory to high as well as making the code memory read only. Another avenue of attack would be the length of the variable v. Since it is a user provided variable defined by the programmer, its type and consequently its length are known in advance by the compiler. Proper numbers of conditional or unconditional changes of memory integrity level instructions can be generated by the compiler, to classify the associated memory cells for the variable v. This requirement can be expressed as:

$\forall Intmem\ r_d(w), H \in Classify(v), Reg(r_d) + w \in Mem_v[m, m + l - 1]$ where $Type(v) : \tau \wedge Size(\tau) : l$.

The fourth requirement with all conditions can be expressed as follows:

MIR 4. $\forall v \in UV_\alpha \wedge \exists(\varphi, v) \in CT_\alpha \wedge Classify(v) : a_1; a_2; \ldots; a_n \wedge Type(v) : \tau \wedge Size(\tau) : l$

1. *"if $\varphi(v)$ then $Classify(v)$ else halt()"* $\xrightarrow{C} \Big(\big(?\varphi(v); Classify(v)\big) \cup \big(?\neg\varphi(v); halt\big) \Big)$

2. $\forall a_k \in Classify(v)$:

   - $a_k \neq St\ r_d(w), r_s \wedge s_k.RegIL(r_s) = L$
   - $a_k \neq (Intreg\ r_t, H \vee Cintreg\ r_d, r_s, r_t)$
   - $a_k = \big(Intmem\ r_d(w), H \vee Cintmem\ r_d(w), r_s, r_t\big) \implies s_k.Reg(r_d) + w \in Mem_v[m, m + l - 1]$

With all requirements of the memory integrity specified, the theorem and its proof can now be stated.

4.4.6 Theorem of the Memory Integrity Model

This section provides the theorem of the memory integrity model and its proof.

Theorem 4.4.1. *For the program α in execution with its set of process memory PM_α where $PM_\alpha = PML_\alpha \cup PMH_\alpha \wedge PML_\alpha \cap PMH_\alpha = \emptyset$ and the corresponding classification tests CT_α, given the rules of memory integrity model (MIR 1-6) then all computation sequences of the program α will be benign.*

Proof: A memory corruption as defined in Definition 4.4.18 is a `store` instruction that attempts to write outside the boundary of a variable, whether defined by programmer or abstract. Such a `store` instruction under the defined machine model would be $St\ r_d(w), r_s$ where $m_x = Reg(r_d) + w$ is the calculated address for the store operation. The proof is divided into two parts where the first part assumes that the `store` instruction belongs to the $Read_{IO}(v)$ program and studies the possibilities of exploitation. The second part of the proof studies whether the attacker could control the parameters of the `store` instruction that is not part of a $Read_{IO}(v)$ program, particularly when the instruction is writing a high integrity source register into a high integrity memory location. It can be argued that corruption of a low integrity location is inconsequential as either the target does not affect the flow of execution, or if it does, it has a classification test which will detect the corruption.

1. $St\ r_d(w), r_s \in Read_{IO}(v) \implies RegIL(r_s) = L$ based on Definition 4.4.30 and for the `store` instruction to be a memory corruption: $m_x \notin Mem_v[m, m+l-1] \wedge m_x \in Mem_{v'}[m', m'+l'-1]$. There can be the following cases:
   (a) v' is a defined variable hence $v' \in V_\alpha$ where there will be two possibilities:
      i. $v' \in UV_\alpha$ where using the MIR 1 rule $v' \in PML_\alpha$, where by the definition of PML_α and MIR 1-2 $MemIL(m_x) = L$ and the corruption would be inconsequential as:
         A. the variable v' has no effect on flow control and does not require passing a classification test
         B. the variable v' has an effect on flow control and requires passing a classification test which will detect the corruption

NB: If the variable v' has passed its test, then it will be classified before the attacker has the opportunity to corrupt v'.

ii. $v' \notin UV_\alpha$ where using the MIR 3 rule $v' \in PMH_\alpha \implies MemIL(m_x) = H$ and given $RegIL(r_s) = L$, the `store` instruction will halt according to the semantic of the instruction (MIR 5)

(b) v' is an abstract variable and there will be the following two possibilities:

i. $v' \in PML_\alpha \implies MemIL(m_x) = L$ and given $RegIL(r_s) = L$ the corruption will be inconsequential as the attacker can corrupt a temporary storage of a low integrity register (since v' is abstract variable)

ii. $v' \in PMH_\alpha \implies MemIL(m_x) = H$, and given $RegIL(r_s) = L$, the `store` instruction will halt according to the semantic of the instruction (MIR 5)

2. $St\ r_d(w), r_s \notin Read_{IO}(v)$. For $m_x = Reg(r_d) + w$, consider the case where $MemIL(m_x) = H \wedge RegIL(r_s) = H$ and the calculated address or the content of r_s is controlled by the attacker. This part of the proof considers the scenarios under which an attacker can control a calculated address and/or a high integrity register content.

(a) attacker changes w embedded in the `store` instruction which violates the $Int(\alpha)$ assumption under the ICFI model, or code memory read-only assumption.

(b) attacker controls r_d

(c) attacker controls r_s

For the attacker to control either r_d or r_s one the following sequences must happen:

(a) a memory location belonging to a user provided value is first corrupted and then changed to high integrity (requires $Intmem$ or $Cintmem$). The

value is then loaded from memory to a register which is used as r_d or r_s. This sequence (changing integrity level of memory cells of user provided variable) can only be part of the $Classify(v)$ program where no low integrity `store` is permitted (MIR 4).

(b) a memory location belonging to a user provided value is first corrupted and then loaded into a register. The register is then elevated to high integrity used as r_d or r_s. This sequence of instructions is not needed for any reason as elevation of integrity level for user provided variables must be performed on the memory cells assigned to the variable. The compiler must not generate such sequence which is specified as MIR 4 for instructions of the $Classify(v)$.

NB: The other possible scenario for using the unconditional change of the integrity level, for either memory cells or registers is the $Map()$ function, where as specified by MIR 6 for any given machine code of a function the $Map()$ is performed at the prologue of that function where there is no opportunity for corruption by the attacker.

4.5 Memory Confidentiality Model

The goal of the memory confidentiality model is to protect against memory leak attacks. In this model only the leaks that result in revealing the memory addresses of the process address space are considered. The leaked address can be the loading address of the executable code, shared libraries, heap, or stack. The knowledge of the memory addresses can help the adversary to craft exploits to influence the flow of execution.

4.5.1 Preliminary Definitions

This section discusses the preliminary definitions that will be used in formal expression of the memory confidentiality model. The concept of assigned memory cell(s)

to a variable expressed previously in Definition 4.4.7 is revisited first. The $Map(v)$ function associates a memory range expressed as $Mem_v[m, m+l-1]$ with the variable v, which belongs to the set of all variables of the program α expressed as V_α. The size of the associated range, l, depends on the type of the variable v expressed using the $Type()$ and $Size()$ functions.

In the memory confidentiality model the goal is to protect against leaking the memory addresses. Given only variables of pointer type contain memory addresses the scope of the type of variables can be limited to pointers. The set of all pointer variables for the program α is defined as follows.

Definition 4.5.1. *Set of all pointers for the program* α:
$PTR_\alpha \overset{def}{=} \{x | x \in V_\alpha \wedge Type(x) : ptr \wedge Size(ptr) : l\}$

Given in some high level languages it is possible to retrieve the memory address of a variable using an operator, the usage of such operator on variables must also be considered. To define this set, Ampersand (&) is used as the address retrieving operator and the retrieved address will be considered as an abstract variable as follows.

Definition 4.5.2. *Set of all ampersand operator on variables for the program* α:
$Amp_\alpha \overset{def}{=} \{x | x = \&v \wedge v \in V_\alpha\}$

The programmer may also use variables for pointer manipulations where such variables must also be declared as highly confidential to avoid the mixture of address related expressions that could lead to part of the code revealing memory addresses.

Definition 4.5.3. *Set of all address-related variables for the program* α:
$AV_\alpha \overset{def}{=} \{i | Type(i) = integer \wedge \exists x \in PTR_\alpha, i, v \in V_\alpha \text{ where } m_v = x + i \in Mem_v[m, m+l-1]\}$

A memory leak can now be defined as a `store` instruction that writes a register that contains a memory address to a memory cell within the address range of Mem_{IO}.

Definition 4.5.4. *Memory leak*:

st $r_d(w), r_s$ such that the following conditions hold:

- $Reg(r_s) = m_x \in Mem_x[m, m + l - 1] \wedge \big(x \in V_\alpha \vee Abstract(x)\big)$
- $Reg(r_d) + w \in Mem_{IO}$

The definition is comprised of two logical statements about the `store` instruction that is considered as a memory leak. The first statement expresses that the register r_s contains an address in a memory range associated with a variable of the program or an abstract run-time variable. The second statement expresses that the calculated address for the `store` instruction using the register r_d and the offset w is in the Mem_{IO} address range.

4.5.2 Requirements of Memory Confidentiality Model

To protect the variables of the pointer type, it is required that the confidentiality levels of all pointer variables be set to high. This can be expressed formally as follows.

MCR 1. $\forall x \in PTR_\alpha \cup Amp_\alpha \cup AV_\alpha \implies Mem_x[m, m + l - 1] \in MHC_\alpha$

Similar to the rules of the memory integrity model, the confidentiality levels of all memory cells assigned to a variable will be set to the same value. To express the confidentiality level of memory cells and registers, $MemCL()$ and $RegCL()$ notations are used respectively. The second requirement of the confidentiality model can be expressed as follows.

MCR 2. $\forall x, Mem_x[m, m + l - 1] \in PM_\alpha : MemCL_x[m, m + l - 1] = CL_x$

The PM_α expresses the process memory of the program α in execution.

The third requirement states that the confidentiality levels of all other variables will be set to low.

MCR 3. $\forall x \in V_\alpha - \big(PTR_\alpha \cup Amp_\alpha \cup AV_\alpha\big) \implies Mem_x[m, m + l - 1] \in MLC_\alpha$

MCR 4. *Explicit declassification:*
$(x \in PTR_\alpha \cup Amp_\alpha \cup AV_\alpha) \wedge Declassify(x) \implies MemCL_x[m, m+l-1] = L$

This requirement states that the compiler will only generate instructions that lower the confidentiality level of a memory cell or a register that contains a memory address after the initialisation only if the programmer explicitly declassifies a memory address to be used in program output.

The MCR 5 expresses the requirements of each machine instruction similar to the expressions used for integrity model.

MCR 5. $\forall a_k \in \alpha \implies \big[(?\varphi; a_k) \cup (?\neg\varphi; halt)\big]p$

- $a_k = (nop \vee jd\, w \vee movi\; r_d, w \vee halt) \implies \varphi \equiv \top \wedge p \equiv \top$

  The no operation, jump direct, move immediate operands, and halt instructions do not require any verification for memory confidentiality.

- $a_k = add\; r_d, r_s, r_t \implies \varphi \equiv s_k.RegCL(r_d) = s_k.RegCL(r_s) = s_k.RegCL(r_t)$

- $a_k = addi\; r_d, r_s, w \implies \varphi \equiv s_k.RegCL(r_d) = s_k.RegCL(r_s)$

  The arithmetic and logic operations can be done on registers with the same confidentiality level. The result can only be stored in a register with the same confidentiality level.

- $a_k = ld\ r_d, r_s(w) \implies \varphi \equiv \big(s_k.RegCL(r_s) = H\big) \wedge p \equiv \big(s_{k+1}RegCL(r_d) = s_k.MemCL(s_k.Reg(r_s) + w\big)$

  The register that receives the content of a memory cell will be set to the same confidentiality level of that memory cell. Since the register used for address calculation contains an address, it must have a high confidentiality level.

- $a_k = st\ r_d(w), r_s \implies \varphi \equiv \big(s_k.RegCL(r_d) = H \wedge (s_k.RegCL(r_s) = s_k.MemCL(s_k.Reg(r_d)+ w)\big) \wedge p \equiv \top$

  The `store` instruction must verify that the target memory location has the same confidentiality level of the register that is being stored. The compiler will

explicitly set the confidentiality level of the target memory before this instruction in case of temporary storage, or it will be the same if the store logically follows an earlier load. The register used for address calculation must have a high confidentiality level.

- $a_k = bgt\ r_s, r_t, w \implies \varphi \equiv s_k.RegCL(r_s) = s_k.RegCL(r_t) \wedge p \equiv \top$

  The conditional branch instruction must compare the values of the registers with the same confidentiality levels to assure that information is not leaked by deduction from the execution path if the compared values contain address information.

- $a_k = jmp\ r_s \implies \varphi \equiv s_k.RegCL(r_s) = H \wedge p \equiv \top$

- $a_k = ret\ r_s \implies \varphi \equiv s_k.RegCL(r_s) = H \wedge p \equiv \top$

  The indirect jump and return instructions use the content of a register as the target address, hence the confidentiality level of the register must be high.

4.5.3 Instruction Requirements for Confidentiality Model Enforcement

The goal of the memory confidentiality model is to protect addresses of the process memory from leaking to the user. At this stage, the protection of user data is not considered in the discussed model and can be explored in future work. Similar to the integrity level the `nop`, `halt`, and `jd` instructions do not require any changes with regard to the memory confidentiality model as these instructions do not change the content of memory or any register in the register file. The `movi` instruction changes the content of a register however the operand is provided as an embedded value with the instruction itself. If the code of the program can be read by the user the value of the operand is already disclosed. If the instruction is used to initialise a register with a high confidentiality level then the usage of the `movi` instruction can be considered as an explicit declassification.

The conditional branch instruction, `Bgt`, can potentially leak information through deduction if it compares registers with confidential content, as the execution path would reveal information if one of the values is known to the adversary.

Definition 4.5.5. *Branch Greater Than Instruction*:
$Bgt\, r_s, r_t, w \equiv \Big[\Big(?\varphi_1; \big((?\varphi_2; jdw) \cup (?\neg\varphi_2; nop)\big)\Big) \cup \Big(?\neg\varphi_1; halt\Big)\Big] p_1 \vee \bot$ such that:

- $\varphi_1 \equiv s.RegCL(r_s) = s.RegCL(r_t)$
- $\varphi_2 \equiv s.Reg(r_s) > s.Reg(r_t)$
- $p_1 \equiv s'.pc = w \vee s'.pc = s.pc + 1$

The new `Bgt` instruction is constructed using two tests and two choices. The first test φ_1 verifies that both registers are of the same confidentiality level. If this test fails the execution will be halted. This is expressed as the choice $\cup(?\neg\varphi_1; halt)$ and $\vee\bot$ as the outcome of the execution of the instruction. The second test φ_2 performs the comparison of the values of the registers. Based on the result, the choice decides the flow of execution which would be between the direct jump to embedded address w or the execution of the no operation instruction. This is expressed as the logical statement p_1 which shows the two possible outcomes as the content of the program counter in the next machine state.

The `jmp` and `ret` instructions use register contents as target addresses. Hence the confidentiality levels of these registers must be high.

Definition 4.5.6. *Indirect Jump Instruction*:
$Jmp\, r_s \equiv \big[(?\varphi; jmp\, r_s) \ \cup \ (?\neg\varphi; halt)\big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.RegCL(r_s) = H \wedge$ ICFI conditions
- $p_1 \equiv s'.pc = s.Reg(r_s)$
- $p_2 \equiv$ *ICFI conditions*

Definition 4.5.7. *Return Instruction*:
$Ret\, r_s \equiv \big[(?\varphi; ret\, r_s) \ \cup \ (?\neg\varphi; halt)\big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.RegCL(r_s) = H \wedge$ ICFI conditions
- $p_1 \equiv s'.pc = s.Reg(r_s)$
- $p_2 \equiv$ *ICFI conditions*

To enforce the confidentiality level some changes must be made to the instructions which are described as follows. Any instruction that updates the content of a memory cell or a register must follow the confidentiality rules. The required rules for `load` and `store` instructions are defined as part of the instruction semantic.

Definition 4.5.8. *Load Instruction*:
$Ld\, r_d, r_s(w) \equiv \Big[\big(?\varphi; ld\, r_d, r_s(w)\big) \cup \big(?\neg\varphi; halt\big)\Big](p_1 \wedge p_2 \wedge p_3) \vee \bot$ such that:

- $\varphi \equiv s.RegCL(r_s) = H$
- $p_1 \equiv s'.RegCL(r_d) = s.MemCL(s.Reg(r_s) + w)$
- $p_2 \equiv s'.Reg(r_d) = s.Mem(s.Reg(r_s) + w)$
- $p_3 \equiv s'.pc = s.pc + 1$

For the `load` instruction, the requirement is that the confidentiality level of the target register must be set to the same level of the source memory cell. The confidentiality level of the register used in address calculation must be high as it contains address information. The first requirement is expressed as part of the logical statement p_1 which will be true for all executions of the `load` instruction. The other two logical statements express the semantic of the `load` instruction. The second requirement is expressed as a test with a choice that will execute the `load` instruction if the test φ succeeds, and stops the execution if the test fails.

The `store` instruction must verify the confidentiality level of the target memory cell before storing the content of a register at the calculated memory address. The target memory cell must be of the same confidentiality level as the source register for the `store` instruction to execute successfully otherwise the execution will halt. Another requirement of the `store` instruction is that the confidentiality level of the register used in address calculation must be high.

Definition 4.5.9. *Store Instruction*:
$St\, r_d(w), r_s \equiv \Big[\big(?\varphi; st\, r_d(w), r_s\big) \;\cup\; \big(?\neg\varphi; halt\big)\Big](p_1 \land p_2) \lor \bot$ such that:

- $\varphi \equiv s.MemCL(s.Reg(r_d) + w) = s.RegCL(r_s) \land s.RegCL(r_d) = H$
- $p_1 \equiv s'.Mem(s.Reg(r_d) + w) = s.Reg(r_s)$
- $p_2 \equiv s'.pc = s.pc + 1$

The described conditions are expressed using the PDL test notation φ that when true the micro instruction $st\ r_d(w), r_s$ will be executed and when false the `halt` instruction. The micro instruction $st\ r_d(w), r_s$ is not considered a valid instruction on its own and cannot be used by the compiler in code generation.

To avoid accidental mixture of confidential content with other content, which could lead to a memory leak, the arithmetic operations can only be performed on the registers with the same confidentiality level.

Definition 4.5.10. *Add Instruction*:
$Add\, r_d, r_s, r_t \equiv \big[(?\varphi; add\, r_d, r_s, r_t) \;\cup\; (?\neg\varphi; halt)\big](p_1 \land p_2) \lor \bot$ such that:

- $\varphi \equiv s.RegCL(r_d) = s.RegCL(r_s) = s.RegCL(r_t)$
- $p_1 \equiv s'.Reg(r_d) = s.Reg(r_s) + s.Reg(r_t)$
- $p_2 \equiv s'.pc = s.pc + 1$

The PDL test φ expresses the requirement that all involved registers have the same confidentiality level, whereas the logical statements p_1 and p_2 express the semantic of the new `Add` instruction.

Definition 4.5.11. *Add Immediate Instruction*:
$Addi\, r_d, r_s, w \equiv \big[(?\varphi; addi\, r_d, r_s, w) \;\cup\; (?\neg\varphi; halt)\big](p_1 \land p_2) \lor \bot$ such that:

- $\varphi \equiv s.RegCL(r_d) = s.RegCL(r_s)$
- $p_1 \equiv s'.Reg(r_d) = s.Reg(r_s) + w$

- $p_2 \equiv s'.pc = s.pc + 1$

The `Addi` instruction requires two registers where both must have the same confidentiality level.

To allow the programmer to output the memory addresses for troubleshooting purposes, instructions to declassify a pointer variable is needed. That is to allow the memory address to be stored at a memory IO location, it must first be explicitly declassified by the programmer. The declassification then can be translated by the compiler to a change memory or register confidentiality level. These instructions can also be used by the $Map(x)$ program to set up the appropriate confidentiality level for pointer variables at the beginning of the execution of the program α. For any given high level programming function, the prologue and epilogue machine code (similar to the discussion of the integrity model) will perform the required preparation and clean up tasks before the execution of a function starts and after it ends. That is, in a more concrete scenario the $Map()$ function will perform the required memory association for local variables of called functions progressively as the program execution continues.

Definition 4.5.12. *Change Memory Confidentiality Level*:
$[confmem\ r_s(w), cl]p_1 \wedge p_2$ such that:

- $p_1 \equiv s'.MemCL(s.Reg(r_s) + w) = cl$
- $p_2 \equiv s'.pc = s.pc + 1$

This instruction unconditionally changes the confidentiality level of the memory location addressed by $Reg(r_s) + w$ to the embedded confidentiality level. This is expressed as the logical statement p_1 whereas the second logical predicate states that the instruction is a sequential instruction that increments the program counter by one.

Definition 4.5.13. *Change Register Confidentiality Level*:
$[confreg\ r_s, cl]p_1 \wedge p_2$ such that:

- $p_1 \equiv s'.RegCL(r_s) = cl$
- $p_2 \equiv s'.pc = s.pc + 1$

This instruction changes the confidentiality level of a given register to the embedded confidentiality level cl.

4.5.4 Theorem of Confidentiality Model

Theorem 4.5.1. *For the program α in execution with its sets of low and high confidentiality memory MLC_α and MHC_α where $PM_\alpha = MLC_\alpha \cup MHC_\alpha \wedge MLC_\alpha \cap MHC_\alpha = \emptyset$ and given the explicit declassification rule and the requirements of memory confidentiality model (MCR 1-5) then all computation sequences of the program α will not reveal any memory address if no variable is explicitly declassified.*

Proof Sketch: Using the definition of the memory leak there would be a `store` instruction as $St\ r_d(w), r_s$ where r_s contains a memory address which is expressed as $Reg(r_s) = m_x \in Mem_v[m, m + l - 1]$. Since the confidentiality level of the Mem_{IO} address range is set to low it holds that $MemCL\big(Reg(r_d) + w\big) = L$. The `store` instruction has built-in checks that verifies the confidentiality level of the target address with the confidentiality level of the register r_s. For the `store` instruction to succeed one of the following scenarios must happen:

1. given the register r_s contains a memory address then the confidentiality level of this register has been explicitly changed to low;
2. the memory cell that contains an address is explicitly changed to low confidentiality and the value is loaded into register r_s using a `load` instruction.

As there is no need for explicit change of the confidentiality level for the program to perform its benign execution and that the explicit declassification is only used when the purpose of the program is to output the memory address, then in either of the above scenarios the goal of the program is to expose the memory address. On the other hand when no explicit declassification instruction is used there is

simply no path in the execution of the program that would allow the change of the confidentiality level of a memory cell or register from high to low. The `store` instruction that is part of the memory leak would not successfully execute.

Proof: Using the definition of the memory leak there is at least a `store` instruction $St\ r_d(w), r_{x1}$ such that:

1. $Reg(r_{x1}) = m_x \in Mem_x[m, m+l-1] \wedge \big(x \in V_\alpha \vee Abstract(x)\big)$, the register r_{x1} contains an address belonging to a variable of the program or an abstract variable and all address variables will have high confidentiality level, hence $MemCL(m_x) = H$.

2. $m_y = Reg(r_d) + w \in Mem_{IO}$, the target of the `store` instruction is an address in memory IO.

By the definition of memory IO, the confidentiality levels of all memory cells in $Mem_{IO}[]$ range are set to low, hence: $MemCL(m_y) = L$. For the `store` instruction to succeed (semantic of the instruction) the confidentiality level of the r_{x1} register must also be low: $RegCL(r_{x1}) = L$. Given that the register r_{x1} contains an address, its confidentiality level in the current state is low, and the memory cell from which the register is loaded contains an address and must have been given the high confidentiality level (MCR 1), one of the following sequences of instructions must belong to a computation sequence of the program α for a memory leak to succeed:

1. $Ld\ r_{x1}, r_{x2}(w); confreg\ r_{x1}, L; St\ r_d(w), r_{x1}$ where the `load` instruction loads the address into a register r_{x1}, the `confreg` instruction changes the confidentiality level of the register r_{x1} and then the `store` instruction writes the content of the r_{x1}, which contains an address, into memory IO.

2. $confmem\ r_{x2}(w), L; Ld\ r_{x1}, r_{x2}(w); St\ r_d(w), r_{x1}$ where the `confmem` instruction changes the confidentiality level of the memory cell addressed at $Reg(r_{x2}) + w$, the `load` instruction transfers the content of the memory cell addressed at $Reg(r_{x2}) + w$ into r_{x1} which will set the confidentiality level of the r_{x1} register

to low and the `store` instruction will write the content of that register into memory IO.

Given there is no explicit declassification for the program α, both the `confreg` and `confmem` instructions are unnecessary and must not be generated by the compiler after the initialisation of the program.

4.6 Summary

Memory corruption can be used by the adversary to perform malicious code execution attacks. The vulnerability can be used to corrupt control data to hijack the flow of execution to blocks of code intended by the attacker. It can also be used to corrupt non-control data to bend the flow of execution such that malicious intent is achieved while not violating the benign execution path of the program. Other forms of memory corruption attacks can disclose sensitive information such as memory addresses that can be used in crafting malicious code execution exploitations. In Chapter Three, a formal model was proposed that can prevent the control flow hijack attacks. The proposed ICFI model however cannot prevent control flow bending attacks and memory address leakage attacks where the benign execution flow is not violated by the adversary.

This chapter defined two models for memory architecture to protect against memory corruption that would otherwise lead to control flow attacks and memory address leakage. The first model aims to hinder the adversary's ability to corrupt memory cells that will affect the flow of execution. This will effectively prevent any type of control flow hijack or control flow bending attack. To achieve this goal the model formally defines two classes of variables: (i) user provided variables considered as untrusted; and (ii) program internal variables considered as trusted. These classes of trusted and untrusted, or low and high integrity, will partition the process memory of a program in execution. Formally defined rules of the integrity model will make sure that low integrity values will not affect the flow of the execution. This is

mainly ensured by separating the instructions that affect the integrity of memory cells from the instructions that change the memory content. More specifically by limiting user's write access only to low integrity memory cells where corruption will be inconsequential. The result of the proposed model is expressed as a theorem with a formal proof.

The memory confidentiality model will prevent the attacker from leaking memory addresses that could be used in crafting other exploitations. Similar to integrity model two classes of variables are defined: (i) memory address and address-related variables considered as confidential; and (ii) all other variables considered as non-confidential. The two sets will partition the process memory of the program in execution where the confidential set will be protected from leakage. This is achieved by classifying memory IO address range as non-confidential and limiting the write access to this range from only non-confidential registers. Formally defined rules will assure that the classified memory cells as confidential cannot be written in the memory IO address range without being explicitly declassified. The explicit declassification will require the programmer to use high level programming language qualifiers to specify where the compiler can generate instructions that will declassify variables containing memory addresses.

In Chapter Five, the potential realisation of code memory integrity as a precondition to the ICFI model is discussed. The implementation adds a page-by-page authentication mechanism to a Linux kernel that is triggered by the page fault exception. A potential realisation of the integrity model, that reduces the cost of memory architecture by dividing the process address space, is also proposed in the following chapter. The address division will require changes to the micro operations and semantics of the machine instructions. Another alternative memory model is the combination of integrity and confidentiality for various types of variables. This is discussed in detail in Chapter Five. The proposed model also protects the memory of the operating system against malicious user processes in addition to the combined protection of confidentiality and integrity for the process variables.

Chapter 5

Realisation of Code Memory Authenticity and Alternative Memory Models

5.1 Introduction

In the previous chapters two main theoretical models were proposed to prevent successful exploitation of a vulnerable system. In Chapter Three, an Ideal Control Flow Integrity (ICFI) model was described, where the model could resist against control flow hijack attacks by verifying both forward and backward edges of the transfer of the flow of execution. A required precondition of the ICFI model is the integrity of the code, which will protect against misuse of the direct jump and conditional branch instructions as well as overwriting of the code of the program by the attacker. A realisation of this precondition is proposed in this chapter as a form of code memory authentication in a demand paging system for a Linux kernel.

In Chapter Four, a memory model was discussed with focus on two properties of memory cells: integrity and confidentiality. The first property is protected in the memory integrity model. It prevents corruption of user-provided variables that could lead to any form of unauthorised transfer of the flow of execution, including

control flow bending attacks. A potential realisation of this model to reduce the cost of memory is described in this chapter, as an address space division for the integrity model. The second property is protected in the memory confidentiality model. It aims to prevent leakage of memory addresses that can be used in crafting other forms of exploitation. The two models require extra bits for memory cells and registers to represent the level of integrity and confidentiality. The instructions enforce the required policy according to various rules. A combined model to protect both properties is discussed in this chapter. It provides protection of the operating system memory against malicious user processes.

In this chapter a realisation of the code memory integrity precondition of the ICFI is provided. The implementation, discussed in Section 5.2, performs a page-by-page authentication of the code for a Linux kernel. A formal representation of signature-based authentication and the effects of demand paging on the ICFI model are discussed. This representation is followed by the overall design and required changes to the kernel, to achieve this proposed code memory authenticity. An analysis of a vulnerable high level language program under each of the ICFI and memory integrity models is discussed in Section 5.3 to further clarify the requirements and potential of each model. In Section 5.4 an alternative realisation of the memory integrity model is discussed, including the required changes in micro operations of each instruction to enforce the policy in the proposed solution. The cost of the alternative is decreased in terms of required memory space. A combined memory model to protect both integrity and confidentiality properties for different types of variables is discussed in Section 5.5. This alternative model can also protect the memory of the operating system against malicious user processes. The required changes to the instructions to enforce the combined policies are discussed in detail. The chapter concludes with a summary in Section 5.6.

5.2 Code Memory Integrity

A precondition to the Ideal Control Flow Integrity, as discussed in the third chapter, is the integrity of the executable code. How this can be achieved is discussed from a formal point of view, using cryptographic methods on the content of the executable code in Section 5.2.1. The effect of on-demand paging on the ICFI model and the provided proof is discussed in Section 5.2.2. An implementation of one of the proposed approaches is also discussed in Section 5.2.3. A short review of literature related to code integrity verification for the Linux operating system followed by the analysis of the implementation is discussed in Section 5.2.4. One of the potential outcomes of the implementation for access control is discussed in Section 5.2.5.

5.2.1 A Signature-based Code Memory Integrity

The integrity property can verify whether the code has been modified since a given point in time, but cannot determine the authenticity or the identity of the origin or producer of the code. The non-executable data memory or mutually exclusive Write and eXecute access to memory ($W \oplus X$) property makes the executable memory pages not writable and the writable pages not executable. To some extent this property can be considered as providing the integrity property of the executable code. This however lacks certain security properties. Firstly, all memory pages, including pages that will be used for executable code, have to be writable first. The permission is then changed for the executable pages to read-only and for data pages to non-executable. Hence there is a mechanism for changing the permission on any page. If this mechanism is exploitable, then the integrity property can be bypassed. Secondly, integrity is assured as long as the correct page is loaded from the correct file and cannot give any assurance about the code content. To address these issues, code authentication is used as it provides stronger control based on the content of the executable code, which assures only authentic code is loaded into memory.

To assure authenticity and integrity of the code that reaches the execution state, a signature scheme formally defined by Goldreich [130] is used. The signature scheme

can be symmetric or asymmetric. This allows the use of more efficient implementations of these algorithms. The two conditions are combined to create a signature scheme (symmetric or asymmetric) with the security condition as follows [130]:

Definition 5.2.1. The triple (G, S, V) of all *probabilistic polynomial-time algorithms*, is a *Secure Signature Scheme* that satisfies the following conditions:

1. On input, 1^n, algorithm G (the key generator) outputs a pair of bit strings (s, v) as a signing and verifying pair where n is the security parameter.

2. For every pair (s, v) in the range of $G(1^n)$ and for every executable code $\alpha \in \{0,1\}^*$, algorithms S (Signing) and V (Verification) satisfy:
   $Pr[V_v(\alpha, \beta) = 1] = 1$
   where $\beta = S_s(\alpha)$ is the signature using the signing key s, $V_v(\alpha)$ is the verification algorithm using the corresponding verifying key v, and the probability is taken over the internal coin tosses of algorithms S and V.

3. For every executable code $\alpha \in \{0,1\}^*$ and all probabilistic polynomial-time oracle machines M and their set of queries on input x denoted as $Q_M^O(x)$, and the corresponding output denoted as $M^O(x)$, every positive polynomial P and all sufficiently large n it holds that:

   - in the Private-key signature scheme:
     $Pr[V_v(\alpha, \beta) = 1 \text{ and } \alpha \notin Q_M^{S_s}(1^n) \text{ where } (s, v) \leftarrow G(1^n) \text{ and } (\alpha, \beta) \leftarrow M^{S_s}(1^n)] < \frac{1}{P(n)}$.
   - in the Public-key signature scheme:
     $Pr[V_v(\alpha, \beta) = 1 \text{ and } \alpha \notin Q_M^{S_s}(v) \text{ where } (s, v) \leftarrow G(1^n) \text{ and } (\alpha, \beta) \leftarrow M^{S_s}(v)] < \frac{1}{P(n)}$.

As the producer of the code is explicitly trusted in this proposed model, the authenticity of an executable code must be checked by tracing it back to its producer. To achieve this an identification value will be used to uniquely identify the code producer. Defining a trust model or discussing properties that would make an

executable code trustworthy are outside the thesis scope. This thesis defines a *Trusted Entity* as an entity that is the producer of an executable code and explicitly trusted within the system not to have malicious intent. The set T is defined as the set of *l-bit* values that uniquely identifies each *Trusted Entity* within the system.

Definition 5.2.2. T=$\{x \mid x \leftarrow \{0,1\}^l$ is a unique identifier of a Trusted entity$\}$

The signature scheme binds the executable code to the trusted entity, by first associating a key pair with the *id* of the trusted entity and then using the signing key to produce a signature for the executable code. Each Trusted Entity has a (s, v) *key pair* which is bound to its unique identifier *id* by the *KeyVault* function that is defined as follows.

Definition 5.2.3. $KeyVault = \{(s, v, id) \mid (s, v) \leftarrow G(1^n)$ of (G, S, V) *Secure Signature Scheme* and $id \in T\}$

As authenticity provides authentication in addition to integrity for an executable program α: $Auth(\alpha, id) \implies Int(\alpha)$, then the notation $Auth(\alpha, id)$ can be used in place of $Int(\alpha)$. The signature verification can be done for the entire executable code or page-by-page as required by the flow of execution. In the following section, there is a discussion of the formal requirements of page-by-page verification of authenticity for executable code.

5.2.2 Memory Blocks and Paging

Paging is a memory management technique that is used in modern operating systems to provide a more efficient means of managing the memory allocated to programs in execution. Regardless of the technique used, this mechanism, which is generally supported in hardware, provides a way of translating a logical address to a physical address and removes the requirement of loaded blocks of the executable code being in contiguous memory locations. From an abstract point of view, the theorem results hold in such architectures, as long as the paging mechanism is implemented correctly and is not exploitable by itself. To demonstrate this point the memory blocks of the

executable code α, comprised of an atomic instruction sequence $a_1; a_2; \ldots; a_n$, are defined as a sequence of fixed-length non-overlapping blocks $P_1; P_2; \ldots; P_m$ where $m = \frac{n}{page\ size}$. Each of the memory blocks P_i can be considered as a program α_i for which the two theorems will hold with some adjustments. An adjustment is only needed where the boundaries of the memory blocks are traversed. In other words an adjustment in expressing the theorems and proofs is needed in the state transition from user code to the adversary code $a_k; b_1$ in the following scenarios:

1. $a_k \in SQ$ and a_k is the last instruction of the page P_i where either a_k is the very last instruction of the program α which needs to be *halt* and $a_k \notin SQ$, or the next instruction is the first instruction of page P_{i+1} and $b_1 \neq a_{k+1} \in P_{i+1}$ which then violates $Int(\alpha)$. The integrity in the paging scenario is discussed later in this section.

2. $a_k \in DJ$ and $w \in_{emb} a_k$ and there will be one of the following scenarios:

   - $a_w \in P_i$ (that is the direct jump is to the same page);
   - $a_w \in P_x$ (the jump is to another page);

   in either case $b_1 \neq a_y \in (P_i \vee P_x)$ violates $Int(\alpha)$.

3. $a_k \in CB$ this is the combination of the previous two scenarios (four combinations):

   - a_k is the last instruction of the page P_i and the condition is not true in which case the first instruction of the page P_{i+1} needs to be executed ($b_1 \neq a_{k+1} \in P_{i+1}$ violates $Int(\alpha)$);
   - a_k is not the last instruction and the condition is true and jump is to the same page P_i ($b_1 \neq a_x \in P_i$ violates $Int(\alpha)$);
   - the condition is true in which case regardless of whether a_k is or is not the last instruction of the page P_i, the direct jump happens; the jump could be to the same page P_i ($b_1 \neq a_x \in P_i$ violates $Int(\alpha)$) or another page P_x ($b_1 \neq a_y \in P_x$ violates $Int(\alpha)$).

4. $a_k \in IJ$ in which case the second theorem requires the verification of the destination address based on the sets AC_α and RM_α and the address conversion determines the validity of the destination.

For the integrity of the program α in a paging system the following definition can be used.

Definition 5.2.4. *Program Integrity in Demand Paging:*
For the program α of size n, and comprised of a sequence of fixed-length non-overlapping blocks $P_1; P_2; \ldots; P_m$ where $m = \lceil \frac{n}{page\ size} \rceil$, (and P_m padded with zero when $m \neq \frac{n}{page\ size}$), then:
$Int(\alpha) \equiv Int(P_1 P_2 \ldots P_m) \iff Int(P_1) \wedge Int(P_2) \wedge \cdots \wedge Int(P_m)$.

To take advantage of the efficiency of the paging mechanism in verification of the integrity of the programs in execution, the integrity property can be used on the individual pages. As shown in the first three scenarios the requirement comes down to the integrity of the individual page (P_i, P_{i+1}, or P_x). Assuming that the paging mechanism by itself is not exploitable then the state transition to the adversary code is prevented. In the fourth scenario the integrity of the individual page is necessary but not sufficient as the destination is determined by the content of a register which could be what the adversary intends. The integrity of the target page would still be intact. This issue which is the same problem without paging is addressed by use of the destination address verification where the paging mechanism is considered in address translations when dealing with AC_α and RM_α sets. In this case only the integrity of the pages on the execution path is necessary, not the entire code of the program α. To map the code authenticity, as discussed in previous section, to the paging mechanism and integrity of each page a set of *Block Signatures* is defined for the executable α.

The *Block Signatures* for each executable code α comprised of pages $P_1; P_2; \ldots P_m$ produced by a trusted entity with $id \in T$, with associated signature key pair (s, v) where $(s, v, id) \in KeyVault$ is defined as follows.

Definition 5.2.5. $\forall P_k \in \alpha$:

$BlockSignatures \stackrel{def}{=} \{(k, Sig_k) \mid Sig_k = Sign(s, P_k) \; \forall k \in 1..m$ where $(s, v, id) \in KeyVault\}$

5.2.3 Implementation of Page-by-Page Verification

To provide a proof of concept a page-by-page verification scheme for a Linux kernel is implemented. The Linux kernel used is Version 3.8.2 of the Ubuntu 12.04 distribution.

Authenticated Code Execution in Demand Paging System

In the Linux operating system all executable code is memory mapped for execution. A page fault exception will occur when the requested address is within a page that has not been loaded into memory by the OS. To prevent loading unverified code into system memory in a demand paging system, the verification of the content of a code page must happen before the page is loaded into memory or some time before the execution of the first instruction in that page. A page-by-page verification scheme can verify the authenticity of a code page at the time of the page fault exception handling procedure. This will assure that only the part of the code required for execution is verified before the flow of execution is transferred to the loaded page. If a page does not pass the verification, the process execution will be aborted by the OS. To achieve a better performance the verification uses a type of Message Authentication Code (MAC) for each page.

The `__do_fault()` function in `memory.c` in the Linux kernel performs the operation of loading a page that has caused a page fault exception. In this implementation the kernel is compiled with a method of loading an initial RAM disk which detects the required device drivers and other kernel modules, and loads them into memory along with the final kernel image. To support this method of boot using the initial RAM disk or `initrd`, a two stage verification is done. Two separate passwords are used to implement a password-based MAC method used for both stages. In the

first stage, a small body of the code that is necessary to load drivers as part of the `initrd` procedure is verified using a key derived from the chosen `initrd` password. The SHA256 of this password is used as the key that is concatenated with the page content for all pages of the executable files contained in the `initrd` image. This key is only used for the pages necessary to load the code up to the point where a password can be entered by user during booting. Once the password is entered, the remainder of the code including the code required for the remainder of the `initrd` procedure will use the entered password by the user. The password and the SHA256 digest of the password is stored in a memory structure (listing A.9[1]) defined in the `memory.c` file (listing A.8).

A global variable (listing A.10) specifies whether the page-by-page verification must use the `initrd` key or the system-wide key derived from the user-provided password during the boot process. The `initrd_key` variable (listing A.3) will contain the provided key through the boot command line option which is defined and allocated in the `main.c` file (listing A.1) under the `init` folder in Linux kernel source.

The `__do_fault()` function in the `memory.c` file is modified to verify the authenticity of the code. Two functions, `do_verify_page_mac()` and `sha256_mac()`, are added to perform the code page authentication.

The first function, shown in Algorithm 5.1, performs the verification of the requested page for the current process by passing a pointer to the filename of the current process and a pointer to the page structure. For clarity, the memory allocation and error handling is omitted from the algorithm. The C code of the function is provided in Listing A.18.

Once the appropriate key is chosen the name of the directory where the MAC of the page is stored is generated. The SHA256 hash value is generated for the concatenation of the secret value and the content of the page. The `sha256_mac()` function (listing A.17) in the modified `memory.c` file performs this operation. The message digest of the secret value concatenated with each page for all executable

[1]Listings labelled as A.n are provided in the Appendix.

Algorithm 5.1: Verify Page MAC

```
Function verify page mac
    Data: filename, page
    Result: Page MAC verification succeeded or failed
    if the integrity password is set then
        key ← key derived from user-provided password
    else
        key ← initrd key
    fnsha2 ← sha256 hash(filename)
    pmac ← "/hashes/" || hexadecimal string(fnsha2)
    page mac ← sha256 mac(key, page)
    pmac ← pmac || hexadecimal string(page mac)
    if file(pmac) not found then
        return with verification failure
    return without error
```

code and libraries within the system is generated and stored under the `/hashes` folder. The overall logic of generating message authentication code for each page of an executable code is shown in Figure 5.1.

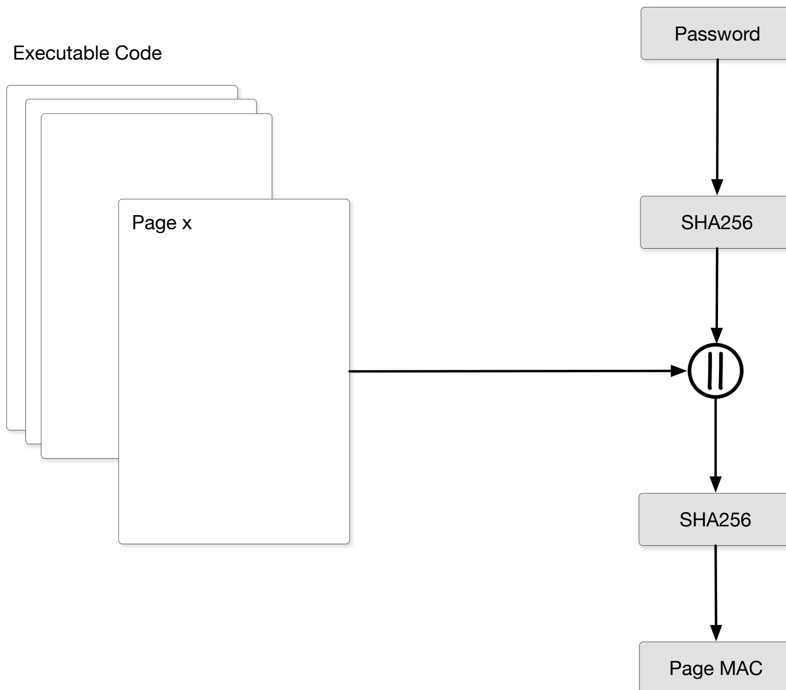

Figure 5.1: Page message authentication code generation process.

To make the content of the folder more manageable, the message digests for each executable code are stored under the same folder. The name of the folder containing all of the page MACs is chosen as the hash of the name of the executable

code including its path within the file system. The process of generating the folder name for page MACs is shown in Figure 5.2.

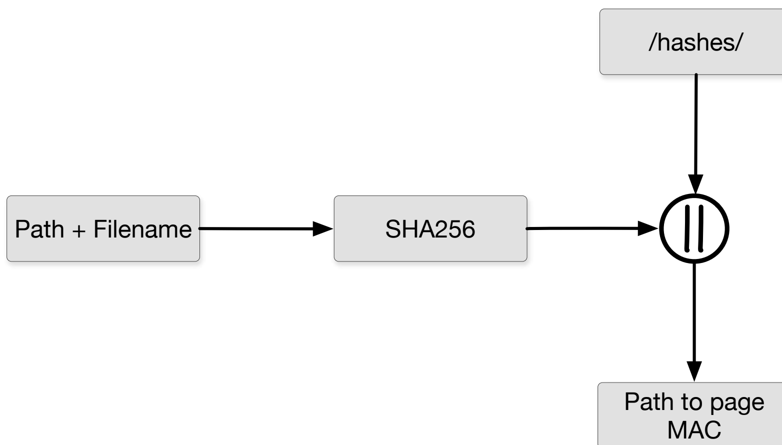

Figure 5.2: Folder name generation for page message authentication code.

To make the verification process more efficient each message digest is used as a filename with no content. Using this approach the verification succeeds if the file is found under the appropriate folder for the given page at run time. If a file is not found the verification will fail.

The Algorithm 5.2 shows the logic of the modifications made to the `__do_fault()` function. The C code is provided in Listing A.19.

Algorithm 5.2: Logic of modification added to __do_page_fault()

```
if page is executable then
    if page is associated with a file then
        filename ← full path and name of executable file
        if verify page mac(filename, page) failed then
            kill the process
```

The first `if` statement verifies that the page is an executable page. The second `if` statement verifies that the page is memory-mapped with a file. The full path of the executable file is stored in a variable to be passed to the MAC function. The MAC function is then called passing the filename and page structure as parameters. If the verification succeeds the execution will continue normally and if it fails a

jump is executed which will send the `KILL` signal to the `current` process. To avoid triggering a kernel `oops`[2], the `__do_fault()` returns `0` as a normal outcome.

Two command line boot options are added to pass the two required MAC keys. The first key is required for verifying the pages of executable files included in `initrd`. The `initrd_key` boot option is read and evaluated to its 256-bit binary value as the `initrd_key`. The second key is derived from a system-wide password to verify the pages of the files that are executed. The system-wide password will be read from the user during boot. The `encrypted_ipass` boot option contains the SHA1 of the password encrypted using the Advance Encryption Standard (AES) 128-bit key in the Cipher Block Chain (CBC) method. To generate the key and Initialisation Vector (IV) for the AES-CBC mode of operation, the password is passed through SHA256. The first 128-bit is used for the key; the second 128-bit as the IV. During booting after the user enters a password, it can be verified whether the correct password is entered by decrypting the boot option and comparing the result with SHA1 of the entered password. The required code to read the boot options from the boot command and the password from the user is added to the `main.c` (listing A.1) file under the `init` directory of the kernel source.

5.2.4 Code Integrity Verification in Linux

This section provides a brief review of the literature for various methods of code integrity or authenticity verification implemented in Linux. A discussion on advantages of the approach and comparisons to the previous work is then provided. A security and efficiency analysis for the approach is also discussed.

Related Work

The first code integrity verification for the Linux operating system is Tripwire, where the message digest of the executable codes are generated and stored in a database [131]. The message digests are then verified periodically. If the verification

[2]A kernel `oops` is a deviation from correct behavior of the Linux kernel.

fails the system administrator will be alerted. The original work had major shortcomings in that the executable codes are vulnerable between the verification periods and that the hash values are generated using the Cyclic Redundancy Code which does not contain cryptographically strong hash functions. Van Doorn et al. [132] proposed the first attached signature verification for Executable and Linkable Format (ELF) binaries, by adding the signature as a custom header to the executable file. This method uses public key encryption to sign the executable files. Catuogno and Visconti [133] proposed a method in which the executable code handler will verify the integrity of the code. The two approaches differ in how they verify the dynamically linked libraries. Another proposed technique is the `digsig` approach [134] where the public key signature of the executable code is verified before the execution starts. A kernel module is then responsible for verification of the signature. This approach relies on the Linux Security Module (LSM) framework that provides a modular security policy architecture for SE Linux [134]. The signature verification is triggered by the `security_file_mmap` LSM hook. If the verification is successful, the result will be cached. If the executable file is opened with write access which can be detected whenever the `security_inode_permission` hook is triggered, the cache entry for the executable will be removed. The problems with this method are the cost of public key signature verification in terms of required processing time and that the kernel module can be unloaded using a vulnerability in the system.

Other methods use similar concepts with differences in techniques for authenticity verification and implementation. For instance a detached signature verification approach using a symmetric algorithm that uses the `execve()` system call as the trigger for verification has been implemented in [135]. To achieve a better performance, the Cipher Block Chaining (CBC) with the Advanced Encryption Standard (AES) is used on the SHA256 of each executable code. The `do_execve()` kernel function is modified to verify the encrypted hash of the executable code. A password is used to generate the symmetric key and Initialisation Vector (IV) for the CBC-AES-128 algorithm. The first 128 bits of the SHA256 of the password is used as

the key; the second 128 bits are used as the IV. The advantage of this approach compared to other methods is that the signature is detached which makes the approach independent of the type of executable code. Another advantage is the efficiency of the symmetric algorithm compared to public key methods. One disadvantage of this approach is that given the libraries are not directly executed and are loaded by the dynamic loader, the code of the dynamic loader must also be modified to verify the authenticity of the library file. Another disadvantage is that the entire code of the binary executable is verified.

Security and Efficiency Analysis

The page-by-page verification has the advantage of using the same method for both executable code and the libraries, as all executable code in the Linux kernel are loaded using the memory map technique. Another advantage of this approach is that only the requested pages are verified in contrast to the entire code in other approaches. In addition, as long as the page is in memory, no page fault will occur. This acts as a form of cache without any extra implementation effort. If the verification does not succeed, then only the time to verify a single page is required to reject the execution of the unauthorised code.

As the page-by-page verification only checks the authenticity of required pages, it is expected that the total verification time of the executable code and library pages will be less than the method where the entire executable code and all of the libraries are verified. A time measurement experiment has been conducted using an Intel Core i5 650 processor at 3.2 GHz with 8GB of DDR3 RAM and an Intel 120GB SSD hard drive. As the time measurements fluctuate with each execution, a single execution time measurement is performed and the time for each page verification is recorded. To compare the page-by-page method with the full content verification method, a C program performs a similar operation, `SHA256(Key || file content)`, on the executable code. The execution time of the full content verification is recorded for the executable file and all of the required libraries. When the libraries were the

same for different executable files, in the case of the full content verification method the same time measurement was assumed in calculating the total verification time. Table 5.1 shows the recorded time for four executable code instances and their required libraries.

Table 5.1: Time Measurements for the Page-by-Page Verification Method versus the Entire Content Verification Method (μ seconds)

File name	Page-by-Page	# executed pages	Avg. per page	Entire Content
`/bin/busybox`	5484	58	94.5	24809
`/bin/tar`	8345	211	39.5	29646
`/bin/grep`	6303	157	40.1	22855
`/bin/ls`	7165	188	38.1	27926

The executable `/bin/busybox` is compiled statically and does not rely on any library functions. Tables A.1, A.2, and A.3 provided in the Appendix show the required libraries for each of `/bin/tar`, `/bin/grep`, and `/bin/ls` respectively. The verification time for each individual executable and library is also reported.

To achieve a better performance the process of generating the MAC for a page can be pipelined by reading the content from the file. As the hash functions are designed using a compression engine with two inputs, comprising the result of previous step and a message block, the process of generating the MAC for the page can start as soon as the first block sized data is read from the disk. This will reduce the total time of generating the hash of the page, to the time to execute the hash algorithm on a single message block. Reading from the disk will likely be slower than digesting a single message block. This however requires precise coordination between the function resolving the page fault and the registered method for reading a file from disk. This can be achieved by allocating memory to store the intermediate values accessible to the kernel's lower level file operations where these values can be generated as soon as a message block is read from disk. The final message digest will be available for the `__do_fault()` function to verify. This has not been implemented in the approach.

The security of the approach can be analysed from two aspects: (i) the type of access to the MAC storage; and (ii) the security of the MAC generation mechanism. It is assumed that the folder which contains the MACs of pages for all executable code is mounted as read only or as an immutable file system, although this aspect has not been implemented. This is a security requirement as the write access to MAC storage will give the adversary the opportunity to perform a Denial of Service attack by deleting any of the MACs stored in that file system. The adversary cannot add any authentic MAC for any unauthorised executable code without knowing the system-wide password used to generate the stored MACs, even if the file system is mounted as read/write. A system call can be added to provide a secure interface for adding new MACs for new or updated executable files which can be controlled with user privileges.

The security of the generated MACs depends on the SHA256 algorithm and the difficulty of the password. Since SHA256 is a one way algorithm the only methods available to generate a valid MAC for a page is to either perform a brute-force attack on the password or find a collision for the hash algorithm that generates the same MAC. As no collision has been found for the SHA256, the algorithm is considered cryptographically strong. Given the SHA256 of a password is concatenated with the page content, the most efficient attack would be a dictionary attack on the password. The maximum length of the password is set to 64 characters providing a large enough set of possible passwords to render the dictionary attack computationally infeasible.

5.2.5 Content-based Access Control and Authentication: A Discussion

This section proposes a method of content-based access control that provides stronger protection against execution of unauthorised executable files by users. The implementation of this proposed approach is left for future work. Only a brief overview of the design of such a method is provided.

The combination of a page-by-page verification mechanism and a key ring structure can provide a form of content-based access control. To achieve this, a key can be generated for each user within the system and the MACs of the pages of the allowed executable code for each user will be generated and stored separately. The generated keys can be stored securely in a key ring file and be loaded into memory during the boot process for faster access. In this case the folder name where the MACs for each executable file are stored can also be scrambled using a cryptographic method similar to the MAC generation process, which will simplify the management of the MACs. To perform the page-by-page verification, the `do_verify_page_mac()` function will have access to the key ring memory or file storage structure to retrieve the corresponding key according to the process or user privilege level. The access control can be enforced by generating the MACs only for executable code that a user is allowed to run within the system. In this approach even if the user has access to copies of unauthorised executable files, she will not be able to run these files on the system. Finer control can be achieved by aligning functions to pages and only signing pages that contain allowed functions. This can provide a better control than any white- or black-list approach as the control is enforced based on the content of the code rather than a name reference to the file.

5.3 x86 Sample Code Analysis

This section discusses two hypothetical scenarios where the ICFI and memory integrity models are implemented and also discusses how the exploitation would be prevented in each scenario. Given this thesis has provided the proof of the theorems in the defined abstract machine, this example is only provided to relate the concept to a widely used architecture and to demonstrate the complexity of potential implementations. The example is not an actual implementation of either of the proposed models.

5.3.1 Simple Buffer Overflow in ICFI Model

This section discusses a simple buffer overflow example in the ICFI model. The C code of this example is shown in Listing 5.1.

Listing 5.1: C code of a simple buffer overflow vulnerability

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void bof(char *str) {
  char buffer[100];

  strcpy(buffer, str);
  return;
}

int main (int argc, char *argv[]) {
  bof(argv[1]);
  return 0;
}
```

Compiling the code on a 64-bit Ubuntu operating system using the `gcc` compiler produces an ELF executable with various headers and sections. The compiled code of the `main()` and `bof()` functions are shown in Listings 5.2 and 5.3.

Listing 5.2: Assembly of the main() function

```
00000000004005db <main>:
  4005db: 55                    push   rbp
  4005dc: 48 89 e5              mov    rbp,rsp
  4005df: 48 83 ec 10           sub    rsp,0x10
  4005e3: 89 7d fc              mov    DWORD PTR [rbp-0x4],edi
  4005e6: 48 89 75 f0           mov    QWORD PTR [rbp-0x10],rsi
  4005ea: 48 8b 45 f0           mov    rax,QWORD PTR [rbp-0x10]
  4005ee: 48 83 c0 08           add    rax,0x8
  4005f2: 48 8b 00              mov    rax,QWORD PTR [rax]
```

```
4005f5: 48 89 c7               mov    rdi,rax
4005f8: e8 99 ff ff ff         call   400596 <bof>
4005fd: b8 00 00 00 00         mov    eax,0x0
400602: c9                     leave
400603: c3                     ret
400604: 66 2e 0f 1f 84 00 00   nop    WORD PTR cs:[rax+rax*1+0x0]
40060b: 00 00 00
40060e: 66 90                  xchg   ax,ax
```

A C program normally receives two arguments `argc` and `argv` for the `main()` function. The `argc` argument specifies the number of command line arguments that have been passed to the program. If no command line argument is passed, the name of the program would be the only argument that the code receives. The `rdi` register contains the `argc` parameter. The `rsi` register contains the address to the beginning of `argv` which is an array of strings specifying all command line options passed to the program. The first command line string is passed to the executable code using the address to the first byte of the string in memory. This address is stored as the second element of `argv[]` array and is passed to the `bof()` function as `argv[1]`. The following instructions prepare the `rdi` register to contain the address of the command line string. The string address is the second element of the `argv[]` array after the program name.

```
mov    rax,QWORD PTR [rbp-0x10]
add    rax,0x8
mov    rax,QWORD PTR [rax]
mov    rdi,rax
```

Once the `rdi` register is ready, the function `bof()` is called using a direct call instruction where the address of the `bof()` function is provided as an immediate operand. In the ICFI model, the thesis assumed local functions would also use indirect instructions to unify the treatment of function calls. In the proposed model the tuple: (`4005f8,400596`) would belong to the set AC_{bof}. For the call to the library function `strcpy()` in `bof()`, the target address of the direct call is the address of an entry in the Procedure Linkage Table (PLT). The PLT is a trampoline

that allows the transfer of the flow of execution to a dynamic library function. In this case the PLT entry contains an indirect call, using an address as its operand, to read the target of the call which points to an entry in `.got.plt` section of the ELF binary. This entry will be initialised with the address of the dynamic loader of the system where after the first call to the library function, `strcpy()` in this case, it will load the `C` library and resolve the address by overwriting the entry of `.got.plt` for the called function. To satisfy the requirement of the ICFI model, the tuple (`4005bf, strcpy()`) must be in the set AC_{bof}. The call must use an indirect call instruction using a register. At runtime the PC can be checked to contain `base + 4005bf`; the register to contain the address of `strcpy()`.

Listing 5.3: Assembly of the bof() function

```
0000000000400596 <bof>:
  400596: 55                      push   rbp
  400597: 48 89 e5                mov    rbp,rsp
  40059a: 48 83 c4 80             add    rsp,0xffffffffffffff80
  40059e: 48 89 7d 88             mov    QWORD PTR [rbp-0x78],rdi
  4005a2: 64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
  4005a9: 00 00
  4005ab: 48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
  4005af: 31 c0                   xor    eax,eax
  4005b1: 48 8b 55 88             mov    rdx,QWORD PTR [rbp-0x78]
  4005b5: 48 8d 45 90             lea    rax,[rbp-0x70]
  4005b9: 48 89 d6                mov    rsi,rdx
  4005bc: 48 89 c7                mov    rdi,rax
  4005bf: e8 9c fe ff ff          call   400460 <strcpy@plt>
  4005c4: 90                      nop
  4005c5: 48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]
  4005c9: 64 48 33 04 25 28 00    xor    rax,QWORD PTR fs:0x28
  4005d0: 00 00
  4005d2: 74 05                   je     4005d9 <bof+0x43>
  4005d4: e8 97 fe ff ff          call   400470 <__stack_chk_fail@plt
    >
  4005d9: c9                      leave
```

```
4005da: c3                    ret
```

In the page-by-page code authentication, if the paging mechanism is not vulnerable then the content of the code is not changed. This satisfies the integrity premise of the ICFI model. Direct calls use immediate operands. If the integrity of the code is intact and there is no indirect forward control transfer instruction within the body of the code, then the trampoline section is the only avenue of attack. If all of the addresses of the dynamic library functions are resolved and the table is protected before the execution is started, then the requirements of the forward edges are satisfied. This is due to the fact that the attacker can modify neither individual direct calls to the trampoline entries nor the entries of PLT table. If the library functions call other functions within that library, the transfer of the flow of execution must be a direct jump. The Gnu C Compiler, `gcc`, provides a compile option to disable the *lazy* loading of a dynamic library and resolution of names to addresses for the library functions. This compile option is referred to as RELocation Read Only (`RELRO`) which will resolve all of the library function names to their addresses at load time of the executable code. A method described by Di Federico et al. [136] can perform a code reuse attack on ELF binaries compiled with partial or full `RELRO` option provided that the code is not position independent. A used library for the ELF is not compiled with full `RELRO`, or the `DT_DEBUG` feature of the ELF binary is enabled to intercept events related to loading the executable. The system-wide full `RELRO` compile will have performance impact which will be a trade-off for achieved security.

For the backward edge the functions can be modified to have one return instruction. If multiple return points are necessary a direct call can be made from each of the alternate return points to the return instruction. This will simplify the backward edge verification using a control data stack or a shadow stack. Since it is necessary to verify the offset of the return instruction to avoid abuse of the return opcode that may appear within other instructions, having a single exit point for

each function requires one entry on the shadow stack per function call. In the example at the time of the call for `bof()` in `main()` the three-tuple (`4005fd, 400596, 4005da`) will be stored in RM_{bof} (or shadow stack). At the time of return it can be verified whether the return address is `base + 4005fd` and (`4005f8 - SizeOf(call instruction),400596`) $\in AC_{bof}$ and whether $PC \stackrel{?}{=}$`base + 4005da`. When compiling all of the executable code within the system including the shared libraries with the full `RELRO` option combined with the page-by-page verification of the code, a shadow stack with a single return instruction for all functions can potentially satisfy all the requirements of the ICFI model as long as the page permissions are protected and cannot be exploited and the programs are not allowed to dynamically generate code pages. A formal equivalence analysis of each of the aformentioned implementations is required to prove that all of the requirements of the ICFI model are satisfied.

5.3.2 Simple Buffer Overflow in Memory Integrity Model

This scenario assumes the `C` programming language has the qualifier `low integrity` to define variables with low integrity levels and that memory cells as well as registers have the integrity level bits. Given the command line options are provided by the user then the `argv[]` variable must be defined as low integrity. Looking at the source of the `bof.c` it can be observed that passing `argv[1]` to `bof()` function will then pass that variable to the `strcpy()` function which will try to copy the content of one variable to the other. The variable is referred to as `str` within the body of `bof.c`. The `str` variable is not defined as low integrity whereas the `argv[1]` is defined as low integrity. This will result in a compilation error hence the code can be rewritten as shown in Listing 5.4.

Listing 5.4: C code of a simple buffer overflow vulnerability under memory integrity model

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>

void bof(low integrity char *str) {
  char buffer[100];

  strcpy(buffer, str);
  return;
}

int main (low integrity int argc, low integrity char *argv[]) {
  bof(argv[1]);
  return 0;
}
```

It is worth mentioning that the pointer itself is of high integrity however the memory location that it is pointing to is of low integrity. The declaration of the `buffer[]` variable will also result in a compilation error. As in the `strcpy()` function, there is an integrity level mismatch when the content of one variable is being copied to the other. Listing 5.5 shows the corrected code.

Listing 5.5: C code of a simple buffer overflow vulnerability under memory integrity model

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void bof(low integrity char *str) {
  low integrity char buffer[100];

  strcpy(buffer, str);
  return;
}

int main (low integrity int argc, low integrity char *argv[]) {
  bof(argv[1]);
```

```
  return 0;
}
```

The compilation of the program will result in machine code that sets aside 100 bytes of memory at the top of the stack of the `bof()` function. The integrity level of these bytes will be set to low by the function prologue. The integrity levels of memory cells before and after the associated memory of the `buffer` variable will be set to high. If the `argv[1]` variable contains more than 100 bytes the `mov` instruction in the `strcpy()` function for the 101th byte will result in an exception due to an integrity level mismatch between source and destination addresses.

5.4 Address Space Division for Memory Integrity Model

To implement the memory integrity or confidentiality model in a byte addressable memory architecture there has to be one bit per byte for each security property. The cost in this case would be two bits per byte to implement both memory models or 20%. To decrease the cost of the memory model and simplify the architecture, an address space division can be implemented instead. The address space division focuses on a single property structure and rules. A brief description of a model follows where both properties are considered.

5.4.1 Division of Address Space

Recalling from the previous chapter, each memory cell was assigned a memory integrity level, low or high, by setting the value of a bit to 0 or 1. The memory cells then can be mixed in the available address space of the program in execution. The protection of memory cells from corruption is achieved using the machine instructions by enforcing various rules according to the memory integrity level. Another approach to achieve the same level of security can be done by completely separating the address space for low and high integrity levels, and enforcing the rules using the

machine instructions. Most of the memory integrity rules will stay the same, whereas some of the rules must be modified to accommodate the new memory structure.

In this method two dedicated registers specify the lower and higher bounds of the process address space in a system without the integrity model. Then to implement the model an additional register is needed to store the lower bound of the high integrity address space. This will divide the address space into two parts for each low and high integrity level. The address stored in the boundary register minus one will then specify the higher bound of the low integrity address space. The other two registers will then mark the lower bound of low integrity address space and the higher bound of high integrity address space. The register can be set by the operating system when setting up the program for execution. The option can be given to the programmer to specify how the boundary must be set.

Since the individual cells no longer have the integrity level, the change of integrity level would be implemented as a copy of memory cells from the low integrity address space to the high integrity address space. The assigned memory cells to the involved variable must also be changed for as long as the variable is set to high integrity. The same rules of the change of memory integrity level apply to this implementation. The micro operations of the $Classify()$ function, in its simplest form, would involve a `load` instruction from the low integrity address space, followed by a change in integrity level of the `load` target register from low to high and a `store` instruction in the high integrity level address space. The following section describes in more detail the necessary changes for this method.

5.4.2 Instruction Requirements for Address Space Division

First the dedicated register is defined for the memory integrity boundary. For clarity LB, HB, and IB express Lower Bound, Higher Bound and Integrity Bound respectively. Letters L and H express the Low integrity level and the High integrity level for a memory cell or a register.

Definition 5.4.1. *Memory Integrity Bound Register*:

for m_{LB} and m_{HB} the lower and higher bounds of the process address space for program α:

$\forall m \in PM_\alpha \implies m \in [m_{LB}, m_{HB}]$ then: $Reg(r_{mib}) = m_{IB}$ such that:

- $m_{IB} \in [m_{LB}, m_{HB}]$
- $\forall m \in [m_{LB}, m_{IB} - 1] \implies MemIL(m) = L$
- $\forall m \in [m_{IB}, m_{HB}] \implies MemIL(m) = H$

The value of the register can be set by a privileged instruction at the initialisation time before the execution of the program begins. As the content of the r_{mib} register does not change during the execution of the program α, in micro operation expressions the notation $Reg(r_{mib})$ is used. This indicates the content is the same in both s and s' states for each instruction. The cost of implementing integrity level for registers is low as they are low in number, however the registers can also be divided into low and high integrity groups. This approach would unnecessarily complicate the expressions, so the remainder of this section continues with using an integrity level for the register file.

Then the `load` instruction can be defined as follows.

Definition 5.4.2. *Load Instruction*:

$Ld\, r_d, r_s(w) \equiv \Big[\big(?\varphi; ld\, r_d, r_s(w)\big) \cup \big(?\neg\varphi; halt\big)\Big](p_1 \wedge p_2 \wedge p_3) \vee \bot$ such that:

- $\varphi \equiv s.RegIL(r_s) = H$
- $p_1 \equiv \big(s'.RegIL(r_d) = s.MemIL(s.Reg(r_s){+}w)\big) \equiv \big(s.Reg(r_s){+}w < Reg(r_{mib}) \implies s'.RegIL(r_d) = L)\big) \wedge \big(s.Reg(r_s) + w \geq Reg(r_{mib}) \implies s'.RegIL(r_d) = H)\big)$
- $p_2 \equiv s'.Reg(r_d) = s.Mem(s.Reg(r_s) + w)$
- $p_3 \equiv s'.pc = s.pc + 1$

Since the integrity level for registers are still in place, the φ expression remains the same. The semantic of the instruction also would be the same in this approach.

The change is made to the logical statement p_1 that expresses the outcome of the instruction for integrity level. The `load` instruction copies the integrity level of the memory cell as well as the content. In this case depending on the value of $Reg(r_s)+w$ compared to $Reg(r_{mib})$ the integrity level will be set to low or high. That is if the value of $s.Reg(r_s)+w$ is less than the integrity bound register then the memory cell belongs to the low integrity address space of the program α. If the value is greater than or equal to the r_{mib} content, the memory cell has high integrity.

The `store` instruction can similarly be defined as follows.

Definition 5.4.3. *Store Instruction*:
$St\, r_d(w), r_s \equiv \Big[\big(?\varphi; st\, r_d(w), r_s\big) \cup \big(?\neg\varphi; halt\big)\Big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv \varphi_1 \wedge \varphi_2$
  - $\varphi_1 \equiv s.MemIL(s.Reg(r_d)+w) = s.RegIL(r_s) \equiv \big(s.RegIL(r_s) = L \implies s.Reg(r_d)+w < Reg(r_{mib})\big) \wedge \big(s.RegIL(r_s) = H \implies s.Reg(r_d)+w \geq Reg(r_{mib})\big)$
  - $\varphi_2 \equiv s.RegIL(r_d) = H$
- $p_1 \equiv s'.Mem(s.Reg(r_d)+w) = s.Reg(r_s)$
- $p_2 \equiv s'.pc = s.pc + 1$

The `store` instruction changes the content of the memory. To prevent the corruption it must verify if the integrity level of the target memory cell matches that of the register being stored. In this approach the verification is changed from a comparison of the integrity levels for each register and memory cell to a comparison of the address of the target memory cell with the content of r_{mib}. Based on the result it can be determined if it matches the integrity level of the register that is being stored. This is expressed as the logical statement φ_1 as part of the required test for the `store` instruction φ.

Given there is no longer an integrity level bit per memory cell, the conditional and unconditional changes of memory integrity level instructions are not required.

The change of memory integrity level is done by copying the memory cells from low to high integrity address spaces. The change is only needed for a subset of variables, the user-provided variables that affect the flow of execution, which need to be identified by the programmer. The `load` instruction copies the integrity level of the source memory cell. To store the content at a high integrity level memory address, the integrity level of the destination register in the `load` instruction must be raised to high before being used as the source for the `store` instruction. This sequence of instructions must be limited to the $Map()$ and $Classify()$ functions.

Since the change of memory integrity level is done as part of the $Classify()$ or $Declassify()$ operations for program variables, the association of memory cells with variables must be updated after the completion of the transfer of memory cell content from one address space to another. The requirements of the $Classify(v)$ function for $v \in UV_\alpha$ must change to reflect the characteristic of this approach.

MIR 4. *$\forall v \in UV_\alpha \wedge \exists(\varphi, v) \in CT_\alpha \wedge Classify(v) : a_1; a_2; \ldots; a_n \wedge Type(v) : \tau \wedge Size(\tau) : l$, it holds that:*

1. *"if $\varphi(v)$ then $Classify(v)$ else halt()" $\stackrel{C}{\rightarrow} \Big(\big(?\varphi(v); Classify(v)\big) \cup \big(?\neg\varphi(v); halt\big)\Big)$*
2. *$\forall a_k \in Classify(v)$:*
   - $a_k \neq St\, r_d(w), r_s \wedge s_k.RegIL(r_s) = L$
   - $a_k; a_{k+1}; a_{k+2} = Ld\, r_y, r_x(w); Intreg\, r_y, H; St\, r_{x'}(w'), r_y \implies \big(s_k.Reg(r_x) + w \in Mem_v[m, m+l-1]\big) \wedge \big(s_{k+2}.Reg(r_{x'}) + w' \in Mem_v[m', m'+l-1]\big)$
   - $a_k; a_{k+1}; a_{k+2} = Ld\, r_y, r_x(w); Cintreg\, r_y, r_s, r_t; St\, r_{x'}(w'), r_y \implies \big(s_k.Reg(r_x) + w \in Mem_v[m, m+l-1]\big) \wedge \big(s_{k+2}.Reg(r_{x'}) + w' \in Mem_v[m', m'+l-1]\big)$

The first requirement expresses that there is no other instruction between the action sequence of the classification test and the classification program. This is specified using the notion of "compiled to" with notation $\stackrel{C}{\rightarrow}$ from a high level programming construct "`if-then-else`" to a choice dependent on the classification test. The second requirement is divided into two subclasses. The first subclass

specifies that none of the atomic instructions of the partial program $Classify()$ for variable v is performing the store operation for low integrity memory. The second subclass expresses that the elevation of the integrity level for any register is limited only to registers involved in transferring the content of the memory cells associated with the variable v in the low integrity address space to the associated memory cells in the high integrity address space. This is shown in two parts for unconditional and conditional change of the integrity level for the temporary register involved in a memory transfer operation. The two address ranges assigned to variable v during the execution of $Classify(v)$ are expressed as $Mem_v[m, m+l-1]$ and $Mem_v[m', m'+l-1]$ for the low and high integrity address ranges respectively. To achieve this two registers will contain the base addresses of each range, specified as r_x and $r_{x'}$ for low and high integrity levels respectively. At the completion of the $Classify(v)$ program only the high integrity address range must be associated with variable v. This requires the ability to dynamically change the associated memory to variables. Since the variables of this type are declared at the beginning of the program the appropriate machine code can be generated to enforce this requirement.

5.5 Protection of Operating System Memory

The memory integrity model can be extended to protect the Operating System (OS) memory against user processes. This can be achieved by increasing the number of integrity level bits from one to two. A combination of the confidentiality and the integrity model in this case would require four bits per memory cell which would be quite expensive. An alternate approach is to use the combination of two bits to represent a privilege level rather than integrity and confidentiality levels. At the lowest level the privilege specifies a low integrity and low confidentiality memory cell. The memory cells assigned to the lowest privilege level would be the user-provided variables. All the other program internal variables with the exception of memory addresses will be assigned to Privilege Level 1 (PL1). At the next level will be the memory addresses which will receive Privilege Level 2 (PL2). All memory

cells associated with the OS will receive Privilege Level 3 (PL3), which will include both OS internal variables as well as memory addresses and pointers. The OS can use lower level privileges to interact with users and processes.

5.5.1 Memory Privilege Model

This section describes a memory model that can be used to protect the OS memory against user processes and user processes from malicious users. Initially, the privilege level is defined as a combination of the integrity and confidentiality levels. The memory structure and register file structures are the same as in the previous chapter. To be complete, all required definitions that need not be changed compared to the previous chapter are repeated in this section with minimal explanation for brevity.

Definition 5.5.1. *Memory*:
$Mem : \{0,1\}^l \times \{0,1\}^l \times \{0,1\} \times \{0,1\}$

Definition 5.5.2. *Register file*:
$Reg : \{0,1,\ldots,31\} \times \{0,1\}^l \times \{0,1\} \times \{0,1\}$

The two-bit memory and register privilege level, expressed as RegPL and MemPL, would represent the four possible levels shown in Table 5.2.

Table 5.2: Memory Privilege Levels

Privilege Level	Associated to
Level 0	User Provided Variables
Level 1	Program Internal and Transitional User Provided Variables
Level 2	Program Addresses and Address Variables (Pointers)
Level 3	Operating System Memory

The definitions of the set of variables and the set of user-provided variables are the same as in the previous chapter.

Definition 5.5.3. *Set of Variables* of the executable code α:
$V_\alpha \overset{def}{=} \{v | Variable(v) \in \alpha\}$

Definition 5.5.4. The *Set of User Provided Variables* for executable code α:
$UV_\alpha \stackrel{def}{=} \{v \mid v \in V_\alpha \wedge v \text{ provided by user } \}$

The abstract notion of the association of memory cells with program variables is provided in the definition of the $Map()$ function.

Definition 5.5.5. $Map(v) : Mem_v[m, m+l-1]$ where $v \in V_\alpha \wedge Type(v) : \tau \wedge Size(\tau) : l$

The set of associated memory cells to variables then can be expressed as follows.

Definition 5.5.6. *Set of Variable to Memory Map* for executable α:
$MV_\alpha : \{Mem_v[m, m+l-1] | v \in V_\alpha \wedge Type(v) : \tau \wedge Size(\tau) : l \; \exists! \; m \in Mem \wedge MemPL_v[m, m+l-1] = PL_v\}$

The change in privilege level for the transitional user-provided variables must also be conditioned on a classification test. The definition of the classification test hence will be the same as in the previous chapter and is repeated here for completeness.

Definition 5.5.7. *Set of Classification Tests* for transition from PL0 to PL1:
$CT_\alpha \stackrel{def}{=} \{(\varphi, v) | v \in LV_\alpha \wedge \varphi(v) \in \alpha \wedge \varphi(v) = True \text{ if } v \text{ can be trusted}\}$

The abstract $Read_{IO}()$ and $Write()$ functions are the same as the definition in the previous chapter.

Definition 5.5.8. $Write(v) \stackrel{def}{=} Mem_v[m, m+l-1] \leftarrow Value(v)$ where $v \in V_\alpha \wedge Type(v) : \tau \wedge Size(\tau) : l$

Definition 5.5.9. $Read_{IO}(v) \stackrel{def}{=} Mem_v[m, m+l-1] \leftarrow Mem_{IO}[m_{io}, m_{io}+l-1]$ where $v \in V_\alpha \wedge Type(v) : \tau \wedge Size(\tau) : l$

Based on the given definitions the Mem_{IO} will always have PL0 when reading from and writing to the associated address range.

In the combined model the address and address-related variables will be set to PL2. To assign the appropriate privilege, the set of program pointers PTR_α, the Ampersand operator on variables Amp_α, and the set of address-related variables AV_α are defined as follows.

Definition 5.5.10. *Set of all pointers for the program* α:

$PTR_\alpha \overset{def}{=} \{x | x \in V_\alpha \wedge Type(x) : ptr \wedge Size(ptr) : l\}$

Definition 5.5.11. *Set of all ampersand operators on variables for the program* α:

$Amp_\alpha \overset{def}{=} \{x | x = \&v \wedge v \in V_\alpha\}$

Definition 5.5.12. *Set of all address-related variables for the program* α:

$AV_\alpha \overset{def}{=} \{i | Type(i) = integer \wedge \exists x \in PTR_\alpha, i, v \in V_\alpha$ where $m_v = x + i \in Mem_v[m, m + l - 1]\}$

The definitions of the set of execution variables as well as the set of process memory do not require any changes.

Definition 5.5.13. *Set of Execution Variables for the executable code* α:

$EV_\alpha \overset{def}{=} \{Mem_v[m, m + l - 1] | Abstract(v) \wedge Size(v) : l\}$

Definition 5.5.14. *Set of Process Memory for the executable code* α:

$PM_\alpha \overset{def}{=} MV_\alpha \cup EV_\alpha$

The definitions of $PM0_\alpha$, $PM1_\alpha$ and $PM2_\alpha$ are as follows.

Definition 5.5.15. *Set of Privilege Level 0 Process Memory for the executable code* α:

$PM0_\alpha \overset{def}{=} \{Mem_v[m, m+l-1] | Mem_v[m, m+l-1] \in PM_\alpha \wedge MemPL_v[m, m+l-1] = 0\}$

Definition 5.5.16. *Set of Privilege Level 1 Process Memory for the executable code* α:

$PM1_\alpha \overset{def}{=} \{Mem_v[m, m+l-1] | Mem_v[m, m+l-1] \in PM_\alpha \wedge MemPL_v[m, m+l-1] = 1\}$

Definition 5.5.17. *Set of Privilege Level 2 Process Memory for the executable code* α:

$PM2_\alpha \overset{def}{=} \{Mem_v[m, m+l-1] | Mem_v[m, m+l-1] \in PM_\alpha \wedge MemPL_v[m, m+l-1] = 2\}$

5.5.2 Requirements of the Memory Privilege Model

The first requirements of memory integrity and confidentiality models can be defined as part of the first requirement of the memory privilege model that expresses how the privilege levels are assigned to program variables.

MPLR 1. *Privilege assignment rules:*

(a) $\forall v \in UV_\alpha \implies Mem_v[m, m+l-1] \in PM0_\alpha$

(b) $\forall v \in V_\alpha - (UV_\alpha \cup PTR_\alpha \cup Amp_\alpha \cup AV_\alpha) : Mem_v[m, m+l-1] \in PM1_\alpha$

(c) $\forall x \in PTR_\alpha \cup Amp_\alpha \cup AV_\alpha \implies Mem_x[m, m+l-1] \in PM2_\alpha$

The next requirement expresses that all memory cells associated with a variable will be set to the same privilege level.

MPLR 2. $\forall v, Mem_v[m, m+l-1] \in PM_\alpha : MemPL_v[m, m+l-1] = PL_v$

The following requirement enforces the rules for the $Classify()$ function that changes the privilege level of transitional user-provided variable from PL0 to Level PL1. The rule expresses that the $Classify(v)$ function will not contain any low level store operation, does not elevate the privilege level of a register, and only elevates the privilege level of cells associated with the variable v.

MPLR 3. $\forall v \in UV_\alpha \wedge \exists(\varphi, v) \in CT_\alpha \wedge Classify(v) : a_1; a_2; \ldots; a_n \wedge Type(v) : \tau \wedge Size(\tau) : l$*, it holds that:*

1. *"if* $\varphi(v)$ *then* $Classify(v)$ *else halt()"* $\xrightarrow{C} \Big(\big(?\varphi(v); Classify(v)\big) \cup \big(?\neg\varphi(v); halt\big) \Big)$

2. $\forall a_k \in Classify(v)$*:*

   - $a_k \neq St\, r_d(w), r_s \wedge s_k.RegPL(r_s) = 0$
   - $a_k \neq (Privreg\ r_t, 1 \vee Cprivreg\ r_d, r_s, r_t)$.
   - $a_k = \big(Privmem\ r_d(w), 1 \vee Cprivmem\ r_d(w), r_s, r_t\big) \implies s_k.Reg(r_d) + w \in Mem_v[m, m+l-1]$

This requirement enforces the explicit declassification of memory addresses which will be needed for debugging purposes.

MPLR 4. *Explicit declassification of the address containing variables:*
$(x \in PTR_\alpha \cup Amp_\alpha \cup AV_\alpha) \wedge Declassify(x) \implies MemPL_x[m, m+l-1] = 0$

The following memory privilege requirement expresses that for n programs in execution $\alpha_1, \alpha_2, \ldots, \alpha_n$ the memory ranges not associated with any of the processes will belong to the OS and set to PL3.

MPLR 5. $\forall Mem[] \notin PM_{\alpha_1} \cup PM_{\alpha_2} \cup \cdots \cup PM_{\alpha_n} \implies (Mem[] \in Mem_{OS} \wedge MemPL[] = 3)$

The micro operations of each instruction will enforce the execution rules of the privilege model.

MPLR 6. *Semantics and micro operations of each instruction.*

5.5.3 Instruction Requirements for the Memory Privilege Model

To enforce the rules of the memory privilege model, any instruction that can affect the program variables whether user-provided, program internal, or abstract must follow certain rules. In this section the micro operations of such instructions are discussed. The requirements of the `load` instruction is specified first. Given the registers are temporary storage for memory cells, the destination register in the `load` instruction will be set to the same privilege level as the source memory. The register used for address calculation then must have PL2, as all addresses are considered to have high integrity and are highly confidential. The latter condition is defined as the test φ and the former as the logical statement p_1 which will be satisfied in all execution of the `load` instruction. The logical statements p_2 and p_3 express the semantic of the `load` instruction.

Definition 5.5.18. *Load Instruction*:
$Ld\, r_d, r_s(w) \equiv \Big[\big(?\varphi; ld\, r_d, r_s(w)\big) \cup \big(?\neg\varphi; halt\big)\Big](p_1 \wedge p_2 \wedge p_3) \vee \bot$ such that:

- $\varphi \equiv s.RegPL(r_s) = 2$
- $p_1 \equiv s'.RegPL(r_d) = s.MemPL(s.Reg(r_s) + w)$
- $p_2 \equiv s'.Reg(r_d) = s.Mem(s.Reg(r_s) + w)$
- $p_3 \equiv s'.pc = s.pc + 1$

For the `store` instruction, the target memory cell must have the same privilege level as the source register to avoid memory corruption or memory address leaks. The register used in address calculation must have PL2.

Definition 5.5.19. *Store Instruction*:
$St\ r_d(w), r_s \equiv \Big[\big(?\varphi; st\ r_d(w), r_s\big) \ \cup\ \big(?\neg\varphi; halt\big)\Big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv \big(s.MemPL(s.Reg(r_d) + w) = s.RegPL(r_s)\big) \wedge s.RegPL(r_d) = 2$
- $p_1 \equiv s'.Mem(s.Reg(r_d) + w) = s.Reg(r_s)$
- $p_2 \equiv s'.pc = s.pc + 1$

The privilege comparison is specified as the first of the two conditions of the test φ. The privilege level of register r_d is specified as the second.

The conditional branch instruction can use registers of the same privilege level in its comparison to change the flow of execution. Another requirement is that the privilege levels of the registers used in comparison must be at least PL1 to prevent the user-provided values being used without any test in changing the flow of execution.

Definition 5.5.20. *Branch Greater Than Instruction*:
$Bgt r_s, r_t, w \equiv \Big[\Big(?\varphi_1; \big((?\varphi_2; jdw) \ \cup\ (?\neg\varphi_2; nop)\big)\Big) \cup \Big(?\neg\varphi_1; halt\Big)\Big] p_1 \vee \bot$ such that:

- $\varphi_1 \equiv s.RegPL(r_s) = s.RegPL(r_t) \geq 1$
- $\varphi_2 \equiv s.Reg(r_s) > s.Reg(r_t)$
- $p_1 \equiv s'.pc = w \vee s'.pc = s.pc + 1$

For the indirect jump and return instructions, the register used as the target address must be of PL2. If the ICFI conditions are also enforced, the target address must be an authentic address for the indirect jump instruction.

Definition 5.5.21. *Indirect Jump Instruction*:
$Jmp\, r_s \equiv \big[(?\varphi; jmp\, r_s) \;\cup\; (?\neg\varphi; halt)\big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.RegPL(r_s) = 2 \wedge ICFIconditions$
- $p_1 \equiv s'.pc = s.Reg(r_s)$
- $p_2 \equiv$ *ICFI conditions*

For the return instruction, the register that contains the return address must have PL2. If the ICFI rules are enforced, the return must be a registered return for this function, the call to the function must be an authentic call, and the destination must be a valid destination for that authentic call.

Definition 5.5.22. *Return Instruction*:
$Ret\, r_s \equiv \big[(?\varphi; ret\, r_s) \;\cup\; (?\neg\varphi; halt)\big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.RegPL(r_s) = 2 \wedge ICFIconditions$
- $p_1 \equiv s'.pc = s.Reg(r_s)$
- $p_2 \equiv$ *ICFI conditions*

The arithmetic and logic instructions can only perform operations on registers of the same privilege level. This will prevent the mixture of values with different privilege levels that could lead to violation of integrity or confidentiality.

Definition 5.5.23. *Add Instruction*:
$Add\, r_d, r_s, r_t \equiv \big[(?\varphi; add\, r_d, r_s, r_t) \;\cup\; (?\neg\varphi; halt)\big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.RegPL(r_d) = s.RegPL(r_s) = s.RegPL(r_t)$
- $p_1 \equiv s'.Reg(r_d) = s.Reg(r_s) + s.Reg(r_t)$

- $p_2 \equiv s'.pc = s.pc + 1$

The *Addi* instruction accepts one immediate operand which does not require a privilege check, as the content of the code is protected under either code memory integrity or the mutual exclusive property of code memory for write and execute access.

Definition 5.5.24. *Add Immediate Instruction*:
$Addi\ r_d, r_s, w \equiv \big[(?\varphi; addi\ r_d, r_s, w)\ \cup\ (?\neg\varphi; halt)\big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.RegPL(r_d) = s.RegPL(r_s)$
- $p_1 \equiv s'.Reg(r_d) = s.Reg(r_s) + w$
- $p_2 \equiv s'.pc = s.pc + 1$

To change the privilege level of a register either conditionally or unconditionally, the *Cprivreg* or *Privreg* instructions are used respectively.

Definition 5.5.25. *Change Register Privilege Level*:
$[Privreg\ r_t, pl]p_1 \wedge p_2$ such that:

- $p_1 \equiv s'.RegPL(r_t) = pl$
- $p_2 \equiv s'.pc = s.pc + 1$

The conditional change of the privilege level of a register uses the comparison of a higher privilege level register with a lower privilege level register as a test. When successful, the privilege level of a third register can be set to an embedded value. Conditionally on a test, this instruction then can be used either in classification of a user-provided value for a variable or the elevation of a variable from PL1 to PL2.

Definition 5.5.26. *Conditional Change Register Privilege Level*:
$Cprivreg\ r_d, r_s, r_t, pl \equiv \big[(?\varphi; Privreg\ r_d, pl)\ \cup\ (?\neg\varphi; halt)\big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.Reg(r_t) < s.Reg(r_s) \wedge s.RegPL(r_s) \geq pl$
- $p_1 \equiv s'.RegPL(r_d) = pl$

- $p_2 \equiv s'.pc = s.pc + 1$

The memory equivalent of the change of privilege level instructions are as follows.

Definition 5.5.27. *Change Memory Privilege Level*:
$Privmem\ r_d(w), pl \equiv \Big[\big(?\varphi; privmem\ r_d(w), pl\big) \ \cup\ \big(?\neg\varphi; halt\big)\Big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv s.RegPL(r_d) = 2$
- $p_1 \equiv s'.MemPL(s.Reg(r_d) + w) = pl$
- $p_2 \equiv s'.pc = s.pc + 1$

For the unconditional change of the privilege level, the register used for address calculation must have PL2. The $privmem\ r_d(w), pl$ instruction expresses the semantic of the change of the privilege level of a memory cell without any conditions. This micro instruction will not be a valid instruction on its own and will not have a valid opcode.

Definition 5.5.28. *Conditional Change of Memory Privilege Level*:
$Cprivmem\ r_d(w), r_s, r_t, pl \equiv \Big[\big(?\varphi; privmem\ r_d(w), pl\big) \ \cup\ \big(?\neg\varphi; halt\big)\Big](p_1 \wedge p_2) \vee \bot$ such that:

- $\varphi \equiv \big(s.Reg(r_t) < s.Reg(r_s)\big) \wedge \big(s.RegPL(r_s) \geq pl\big) \wedge \big(s.RegPL(r_d) = 2\big)$
- $p_1 \equiv s'.MemPL(s.Reg(r_d) + w) = pl$
- $p_2 \equiv s'.pc = s.pc + 1$

The conditional change of the memory privilege level uses a comparison between a lower privilege level register and a higher privilege level register as a test to elevate the privilege level of a target address. This is expressed as the φ test which is comprised of three logical statements. The first statement is the comparison of the contents of the registers r_t and r_s in state s. The second statement expresses that the privilege level of the register r_s is at least equal to the embedded value of privilege

level *pl*. The third statement assures that the register used in address calculation is of PL2.

Given there are four privilege levels in this model, the various circumstances that change the privilege levels must be discussed. The first change of privilege level is needed when a user-provided variable that is used in flow of execution decisions has passed its classification test and must be elevated from PL0 to PL1. The second change of privilege level is needed when an internal variable of the program or an elevated user-provided variable is used in some form of address calculation. This type of change in privilege level could result in user-provided variables being used in address calculation which may result in exploitations. The classification of the user-provided value must also assure any further elevation of privilege level will not result in exploitation of the program. Secondary classification tests can be defined for such cases. The third type of privilege level elevation is from PL1 or PL2 to PL3, that is from the user space to the operating system space. The operating system must define classification tests for these types of elevation of privilege level of variables from the user space to the operating system space. The lowering of privilege levels must also be considered for instance from the operating system space (PL3) to the user space (PL2-PL0) to be used as user space addresses, trusted variables, or to be used as output to the user. The program can also print out addresses as part of the debugging process (PL2 to PL0). This will require explicit declassification of the memory address variables. The defined instructions have the required flexibility to address all of the aforementioned scenarios. The compiler for the user program and the operating system must ensure the proper instructions are used for each elevation of a privilege level to prevent potential exploitations.

The above cases can be summarised as the following corollaries.

Corollary 5.5.1. *For the program α in execution with its set of process memory $PM0_\alpha$, $PM1_\alpha$, and $PM2_\alpha$ where $PM_\alpha = PM0_\alpha \cup PM1_\alpha \cup PM2_\alpha$ and $PM0_\alpha \cap PM1_\alpha \cap PM2_\alpha = \emptyset$ and the corresponding classification tests (for user-provided*

variables) CT_α*, given the rules of memory privilege model (MPLR 1-6) then all computation sequences of the program* α *will be benign.*

Using the memory integrity model theorem (Theorem 4.4.6), it can be shown that:

- $PML_\alpha = PM0_\alpha$ and $PMH_\alpha = PM1_\alpha \cup PM2_\alpha$
- MPLR1a satisfies MIR1
- MPLR1b and c satisfy MIR3
- MPLR2 satisfies MIR2
- MPLR3 satisfies MIR4

Following the same logic for the proof of the memory integrity model theorem and the definition of memory corruption (Def. 4.4.18) there will be at least a `store` instruction that attempts a write outside the boundary of a variable. The instruction would be $St\ r_d(w), r_s$ where $m_x = Reg(r_d) + w$ is the calculated address for the target memory cell. Similar to the memory integrity model two cases are considered. In the first case the store operation is part of the $Read_{IO}(v)$ program; in the second case the store operation is not part of the $Read_{IO}(v)$ program.

1. $St\ r_d(w), r_s \in Read_{IO}(v) \implies RegPL(r_s) = 0$ (Def. 5.5.9 and $MemPL_{IO}[] = 0$) and based on MPLR1a and Definition 5.5.15 for this instruction to perform a memory corruption: $m_x \notin Mem_v[m, m+l-1] \wedge m_x \in Mem_{v'}[m', m'+l'-1]$, there exist the following cases:

    (a) v' is a defined variable

    i. $v' \in UV_\alpha$ where using MPLR1a: $MemPL(m_x) = 0$ and the corruption will be inconsequential (v' does not affect the flow of execution). Hence its corruption either has no effect on the flow of execution or if v' affects flow of execution it must have a classification test which will detect the corruption MPLR2,3.

ii. $v' \notin UV_\alpha$ where $MemPL(m_x) = 1 \vee MemPL(m_x) = 2$ (MPLR1b and c) and given the `store` instruction in this case is part of $Read_{IO}(v)$ then $RegPL(r_s) = 0$. Given MPLR6 for the `store` instruction (Def. 5.5.19) this will result in a program halt.

(b) v' is an abstract variable (given the `store` instruction is part of $Read_{IO}(v)$ then $RegPL(r_s) = 0$)

i. $MemPL(m_x) = 0$ the corruption of privilege 0 is inconsequential (low integrity temporary storage)

ii. $MemPL(m_x) = 1$ results in a halt (MPLR6, semantic of `store` instruction as in Def. 5.5.19)

iii. $MemPL(m_x) = 2$ the same as Privilege Level 1

2. $St\ r_d(w), r_s \notin Read_{IO}(v)$. The cases where $MemPL(m_x) = 1 \wedge RegPL(r_s) = 1$ and $MemPL(m_x) = 2 \wedge RegPL(r_s) = 2$ are considered, as any mismatch between privilege levels will result in an exception (MPLR6 for `store` in Def. 5.5.19). Also considered is how the content of either r_s or r_d can be controlled by the attacker:

(a) a memory location that belongs to a user-provided value is corrupted and then changed to PL1 or PL2 and then loaded into either r_s or r_d. The sequence of instructions that change the privilege level of a user-provided variable can only be part of $Classify(v)$ where no PL0 store operation is permitted (MPLR6). This means the corruption has happened before the classification test and is not possible after elevation of the privilege level.

(b) a memory location that belongs to a use-provided value is corrupted and loaded into a register. Then the register is elevated to PL1 or PL2 and affects the content of r_s or r_d. This sequence is not needed and must not be generated by the compiler.

A similar corollary can be defined for address leakage prevention using PL2 memory and register level.

Corollary 5.5.2. *For the program α in execution with its set of process memory $PM0_\alpha$, $PM1_\alpha$, and $PM2_\alpha$ where $PM_\alpha = PM0_\alpha \cup PM1_\alpha \cup PM2_\alpha$ and $PM0_\alpha \cap PM1_\alpha \cap PM2_\alpha = \emptyset$, given the rules of the memory privilege model (MPLR1-6), then no execution of α will leak a memory address if no explicit declassification is used.*

Using the memory confidentiality model theorem (Theorem 4.5.4), it can be shown that:

- $MLC_\alpha = PM0_\alpha \cup PM1_\alpha$ and $MHC_\alpha = PM2_\alpha$
- MPLR1c satisfies MCR1
- MPLR1a and b satisfy MCR3
- MPLR2,4 and 5 satisfy MCR2,4, and 5

Similarly the OS versus user space protection can be defined as a corollary by showing the equivalence of the sets of variables and the built-in micro operations of instructions and given rules of the memory privilege model. The OS protection can only prevent OS memory corruption. All OS memory, which includes addresses, are set to the highest privilege level that protects against unauthorised writes but not incorrect reads.

Corollary 5.5.3. *For user processes in execution $\alpha_i, i = 1, \ldots, n$ and the Operating System in execution with the following user and OS memory spaces:*

- $Mem_{User}[] = PM_{\alpha_1} \cup PM_{\alpha_2} \cup \cdots \cup PM_{\alpha_n} \implies Mem_{User}PL[] \leq 2$
- $Mem_{OS}[] = Mem[] - Mem_{User}[] \implies Mem_{OS}PL[] = 3$

Then by defining:

- $IL = L \equiv PL \leq 2 \wedge IL = H \equiv PL = 3$

- $PML = Mem_{User}[]$
- $PMH = Mem_{OS}[]$

and using the MIR1-6 and explicit classification of user memory after passing the classification test, then all computation sequences of OS will not contain memory corruptions leading to control flow attacks on the OS flow of execution.

5.6 Summary

The integrity of code memory is a precondition to the ICFI model as discussed in Chapter Three. A page-by-page verification of code memory has been implemented that assures only authenticated and authorised code will be loaded into memory. The page-by-page verification compared to previous methods has the advantage that it uses the same mechanism for executable code and shared libraries and only checks the code that is needed according to the flow of execution rather than the entire code of the executable file. The security of the proposed approach relies on immutability of the storage of page MACs and the infeasibility of dictionary attacks on the selected password. An equivalence analysis of the combination of the proposed page-by-page verification with a single return point of all functions with a shadow stack and `RELRO` compilation of all executable code could potentially satisfy all requirements of the ICFI model.

The proposed potential realisation of the memory integrity model in this chapter reduces the cost of required memory by the original model discussed in Chapter Four. This is achieved by dividing the address space between low and high integrity levels. The micro operations of instructions must change for the address space division realisation of the memory integrity model. The most significant change is the implementation of the $Classify()$ function which would require the copy of the variable content from the low integrity address space to the high integrity and the change of any further reference to the variable.

The proposed combined integrity and confidentiality memory model can protect both properties and provide an additional feature. This model can protect the OS memory against malicious user processes. The micro operations of the instructions must change to enforce the requirements of the combined model. More complex models and architectures can be built using the fundamental principles of the integrity and confidentiality memory models.

Chapter 6

Conclusion

6.1 Introduction

A computer system can have vulnerabilities which are weaknesses created due to errors during design, implementation, or use of the system. An exploitation is a method of abusing one or more vulnerabilities to gain more access to the target system that is allowed through normal communication channels. One of the widely used types of exploitation is malicious code execution where the adversary can execute crafted code on the target machine. Depending on the type of vulnerability and method of exploitation, the malicious code execution can gain various levels of control over the target system which can then be used for malicious purposes. The reported number of vulnerabilities that could lead to arbitrary code execution in the Common Vulnerability and Exposure (CVE) database indicates the significance of the problem [3]. As the number of computer devices increases with the increase in number of smart phones, other mobile devices, and Internet of Things (IoT), the number of systems that can be the target of remote code execution by the attacker will also increase unless the vulnerabilities are removed from these systems or the exploitations are prevented.

The exploitation of the target system that leads to malicious code execution can be divided into two broad categories. In the first category the adversary injects the intended malicious code in the target machine's memory and changes the flow

of execution to the crafted code. In the second category the attacker reuses the code that already exists in the target machine's memory such as library functions to achieve her goal. The second category can be further divided into two classes based on whether the benign execution flow is violated or not. Given all machine code must be loaded into memory for execution, all remote code execution attacks exploit the memory corruption vulnerability to gain some form of control over the target machine. Another form of exploitation is to gain information by leaking the content of the target machine's memory. This type of exploitation usually does not violate the flow of execution.

6.2 Contributions

For a vulnerable system to resist exploitation it must remove the requirements of a successful attack. Current modern architectures have various fundamental design flaws from a security point of view by providing too much flexibility, for example through indirect jump instructions, using a single privilege model for memory, and mixing the control and non-control data in memory. Proposing alternative architectures can help design better processors and memory structures with more flexibility in enforcing security policies without sacrificing the performance. Using formal approaches will enable proving the correctness of the solution and provide clear and provable security guarantees.

The contributions can be summarised as follows:

(C_1) *Ideal Control Flow Integrity model to prevent control flow hijack attacks*: For any remote code execution to succeed it must change the flow of execution from what was intended by the programmer to what is intended by the attacker. To address this a model, called an Ideal Control Flow Integrity (ICFI), is proposed that protects against control flow hijack attacks by enforcing three essential policies. The first policy is the integrity of the executable code which protects against overwriting the code of the executable or abusing the immediate

operands of direct jump and conditional branch instructions. The second enforced rule aims to protect the forward edges in control flow transfers which are used in calling dynamically linked library functions. The third policy protects the backward edge which is used to return from called functions. This model is the first provably secure model that allows dynamic linking and relocatable executable code. An abstract machine model proposed in the literature is used to express similar concepts with modifications required for ICFI. Propositional Dynamic Logic (PDL) is used to express logical premises and statements for proposed policy enforcements.

(C_2) *Memory Integrity model to prevent control flow hijack and control flow bending attacks*: A solution is provided to protect against non-control data memory corruption attacks where the flow of execution is not violated by the exploitation. In this solution, referred to as the memory integrity model, the memory cells and registers have an extra bit that indicates the integrity level of that memory cell or register. The required rules for the programmer, compiler, and machine instructions to prevent user-provided variables corrupting any memory cell that may influence the flow of execution are then defined. In this model the corruption opportunity is taken away from the attacker by limiting the machine instructions that read user-provided input to only low integrity memory cells. If the user-provided input will affect the flow of execution, then the programmer must specify an explicit test that can assure that the provided input can be trusted by the program. PDL is used for this model to formally express the micro operations of the instructions, and prove the correctness of the model in preventing memory corruption attacks that could lead to control flow hijack or control flow bending attacks.

(C_3) *Memory Confidentiality model to prevent memory address leakage*: To prevent memory address leakage the memory confidentiality model is defined that is similar to the integrity model in principle. The difference is that memory cells and registers containing memory addresses are considered to be confidential

compared to all other memory cells and registers. This will limit the flow of information between high and low confidential containers except when intended by the programmer.

(C_4) *An implementation of Code Memory Authenticity as a proof of concept for the memory integrity precondition of ICFI model*: A page-by-page verification of code memory for the Linux operating system implements the first step of the ICFI model in providing code memory integrity for all executable code within the system which includes the shared libraries. The implementation improves the performance by only verifying the code that is going to be executed compared to other methods where the entire code of the executable is verified. A Message Authentication approach is used that requires only hash functions and a password to achieve a form of authentication for each executable code page.

(C_5) *Address Space Separation model as a more cost-effective alternative to the memory integrity model*: A realisation of the memory integrity model is also proposed where the cost of required memory is reduced. This is achieved by dividing the memory into two separate address spaces for low and high integrity cells. The required changes compared to the formal model are discussed in detail.

(C_6) *Memory Privilege model as an alternative memory model to combine the integrity and confidentiality models*: In a combined model to protect integrity and confidentiality a solution is proposed that can protect the operating system against user processes by assigning four privilege levels to memory cells and registers. The combined model achieves protection of the operating system against user process memory, process memory addresses from leakage, and memory cells affecting the flow of execution from corruption by user-provided input.

6.3 Future Research

The forward and backward edge protection of the ICFI model can be implemented by modifying the compiler to generate the required sets of Authentic Calls and Return Point(s) for each executable code and library function. The RELocation Read Only or `RELRO` option of the Gnu C Compiler, `gcc` performs all name to address resolutions for library functions. An equivalence analysis of the above feature for dynamic loaders combined with proposed page-by-page verification can potentially satisfy the forward edge requirement of the proposed ICFI model. For backward edge protection the shadow stack with a single return point for functions can satisfy the backward edge requirement of ICFI. Further research can potentially provide the equivalence proofs of the aforementioned forward and backward edge protection to the requirements of ICFI and provide provable security against control flow hijack attacks for modern operating systems. The function pointers require further research to specify how the disjoint set of conditions that specify one valid target per condition can be defined and implemented. Another aspect of research for the disjoint set of conditions for forward edge protection is the possibility of a multi-layer scenario where the conditions are tested in a hierarchical model.

The proposed memory model is similar to the Code Pointer Integrity (CPI) work of Kuznetsov et al. [117] and DataShield proposed by Carr and Payer [137]. In CPI only a subset of code pointers are protected whereas the proposed memory models in this thesis protect all memory cells. DataShield provides a method for programmer to declare sensitive variables using a type-based mechanism where illegal reads and writes to the instances of the sensitive types will be prevented. The goals of integrity and confidentiality models in this thesis differ from DataSheild. The goal of the integrity model is only to prevent memory corruptions leading to control flow attacks. The confidentiality model aims to prevent the leakage of memory addresses. All pointer checking mechanisms such as Softbound [120], Hardbound [121], and Intel MPX [122], need to store metadata related to the pointer in some fashion which requires extra memory access whenever the pointer is dereferenced. The most efficient

solution would still require a single memory read to access the bounds information of a pointer and at least one additional instruction to verify the validity of the information for that pointer. The cost of the proposed models in this thesis will be the preparation of the memory layout and any classification of low integrity variables that will affect the flow of execution. In terms of memory the cost of the models is the additional bits per memory cells or additional registers and micro operations in case of address space separation. Once the memory cells are classified accordingly the policy enforcement has no additional cost regardless of how many times the variables are used or the pointers are dereferenced during the program execution.

Further research into the memory integrity and confidentiality model can use the proposed machine model to provide more complex instructions to address the complexity of modern architectures. Building prototypes of the proposed machine model in virtual or physical forms can enable further research into the complexity of the architecture and security/performance trade-off. Further research into required compiler rules and potential optimisations without sacrificing the achieved security is needed to avoid bypassing the required security measures.

References

[1] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, “Control-Flow Integrity: Precision, Security, and Performance,” *ACM Comput. Surv.*, vol. 50, no. 1, pp. 1–33, 2017.

[2] “Common Vulnerability and Exposure - About CVE,” https://cve.mitre.org/about/, 2017, accessed on 15-04-2018.

[3] “Common Vulnerability and Exposure - Download CVE,” http://cve.mitre.org/data/downloads/allitems.csv, 2017, accessed on 17-09-2017.

[4] E. Damiani, C. Ardagna, and N. El Ioini, “Formal methods for software verification,” in *Open Source Systems Security Certification.* Springer US, Jan. 2009, pp. 1–26.

[5] G. Klein, “Operating system verification—an overview,” *Sadhana*, vol. 34, no. 1, pp. 27–69, Feb. 2009.

[6] H. Chen and D. Wagner, “MOPS: an infrastructure for examining security properties of software,” in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002, pp. 235–244.

[7] B. Schwarz, H. Chen, D. Wagner, G. Morrison, J. West, J. Lin, and W. Tu, “Model checking an entire linux distribution for security violations,” in *Proceedings of the 21st Annual Computer Security Applications Conference*, Dec. 2005, pp. 10–22.

[8] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities." in *Symposium on Network and Distributed System Security (NDSS)*, 2000, accessed on 03-01-2018. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/2000/proceedings/039.pdf

[9] M. F. Ringenburg and D. Grossman, "Preventing format-string attacks via automatic and efficient dynamic checking," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005, pp. 354–363.

[10] K. Chen and D. Wagner, "Large-scale analysis of format string vulnerabilities in debian linux," in *Proceedings of the 2007 workshop on Programming languages and analysis for security*, 2007, pp. 75–84.

[11] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam, "Improving software security with a c pointer analysis," in *Proceedings of the 27th international conference on Software engineering.* ACM, 2005, pp. 332–341.

[12] P. T. Breuer and S. Pickin, "One million (LOC) and counting: Static analysis for errors and vulnerabilities in the linux kernel source code," in *Reliable Software Technologies–Ada-Europe 2006.* Springer Berlin Heidelberg, 2006, pp. 56–70.

[13] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[14] P. T. Breuer and S. Pickin, "Checking for deadlock, double-free and other abuses in the linux kernel source code," in *Computational Science–ICCS 2006.* Springer Berlin Heidelberg, 2006, pp. 765–772.

[15] ——, "Symbolic approximation: an approach to verification in the large," *Innovations in Systems and Software Engineering*, vol. 2, no. 3-4, pp. 147–163, Oct. 2006.

[16] ——, "Verification in the light and large: large-scale verification for fast-moving open source c projects," in *31st IEEE Software Engineering Workshop (SEW 2007)*, 2007, pp. 246–255.

[17] J. S. Foster, M. Fahndrich, and A. Aiken, "A theory of type qualifiers," *SIGPLAN Notices*, vol. 34, no. 5, pp. 192–203, 1999.

[18] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers." in *Proceedings of the 10th USENIX Security Symposium*, 2001, pp. 201–220.

[19] R. Johnson and D. Wagner, "Finding user/kernel pointer bugs with type inference." in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[20] J. S. Foster, T. Terauchi, and A. Aiken, "Flow-sensitive type qualifiers," *SIGPLAN Notices*, vol. 37, no. 5, pp. 1–12, 2002.

[21] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken, "Flow-insensitive type qualifiers," *ACM Transactions on Programming Languages and Systems-TOPLAS*, vol. 28, no. 6, pp. 1035–1087, 2006.

[22] B. Chin, S. Markstrum, and T. Millstein, "Semantic type qualifiers," *SIGPLAN Notices*, vol. 40, no. 6, pp. 85–95, 2005.

[23] G. C. Necula, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy code," *SIGPLAN Notices*, vol. 37, no. 1, pp. 128–139, 2002.

[24] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system f to typed assembly language," *ACM Transactions on Programming Languages and Systems-TOPLAS*, vol. 21, no. 3, pp. 527–568, 1999.

[25] K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic, "TALx86: A realistic typed assembly language," in *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*, 1999, pp. 25–35.

[26] G. Morrisett, K. Crary, N. Glew, and D. Walker, “Stack-based typed assembly language,” *Journal of Functional Programming*, vol. 12, no. 01, pp. 43–88, 2002.

[27] G. C. Necula, “Proof-carrying code. design and implementation,” in *Proof and System-Reliability*, ser. NATO Science Series. Springer Netherlands, Jan. 2002, vol. 62, pp. 261–288.

[28] A. Appel, “Foundational proof-carrying code,” in *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, 2001, pp. 247–256.

[29] N. Hamid, Zhong Shao, V. Trifonov, S. Monnier, and Zhaozhong Ni, “A syntactic approach to foundational proof-carrying code,” in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 89–100.

[30] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” *ACM Transactions on Information and System Security*, vol. 13, no. 1, pp. 1–40, Oct. 2009.

[31] V. van der Veen, L. Cavallaro, and H. Bos, “Memory errors: the past, the present, and the future,” in *Research in Attacks, Intrusions, and Defenses*. Springer Berlin Heidelberg, 2012, pp. 86–106.

[32] H. Orman, “The morris worm: a fifteen-year perspective,” *IEEE Security & Privacy*, vol. 1, no. 5, pp. 35–43, 2003.

[33] E. H. Spafford, “The internet worm program: an analysis,” *SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 1, pp. 17–57, 1989.

[34] “About Us -The CERT Division,” http://www.cert.org/about/, Feb. 2015, accessed on 20-12-2017.

[35] A. One, “Smashing the stack for fun and profit,” http://phrack.org/issues/49/14.html, Nov. 1996, accessed on 03-01-2018.

[36] A. mudge@l0pht.com, "How to write buffer overflows," http://insecure.org/stf/mudge_buffer_overflow_tutorial.html, 1995, accessed on 03-02-2015.

[37] S. Designer, "Linux kernel patch from the openwall project: README," http://www.openwall.com/linux/README.shtml, 1997, accessed on 03-02-2015.

[38] C. Cowan, C. Pu, D. Maier, H. Hinton, 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 USENIX Security Symposium*, 1998, pp. 346–355.

[39] A. Zabrocki, "Scraps of notes on remote statck overflow exploitation," http://www.phrack.org/issues/67/13.html#article, Nov. 2010, accessed on 11-02-2015.

[40] Bulba and Kil3r, "Bypassing StackGuard and StackShield," http://www.phrack.org/issues.html?issue=56&id=5#article, 2000, accessed on 05-01-2014.

[41] H. Etoh and K. Yoda, "Protecting from stack-smashing attacks," Technical report, IBM Research Divison, Tokyo Research Laboratory, Jun. 2000.

[42] S. Designer, "Getting around non-executable stack (and fix)," http://seclists.org/bugtraq/1997/Aug/63, Aug. 1997, accessed on 16-12-2013.

[43] Nergal, "The advanced return-into-lib (c) exploits: PaX case study," http://www.phrack.org/issues.html?issue=58&id=4#article, 2001, accessed on 05-01-2014.

[44] PaX Security Team, "Address space layout randomization," https://pax.grsecurity.net/docs/aslr.txt, 2001, accessed on 03-02-2015.

[45] H. Shacham, M. Page, B. Pfaff, E.-J. 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*, 2004, pp. 298–307.

[46] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, “Surgically returning to randomized lib (c),” in *Proceedings of the 25th Annual Computer Security Applications Conference*, 2009, pp. 60–69.

[47] S. Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique,” http://packetstorm.igor.onlinedirect.bg/papers/bypass/no-nx.pdf, 2005, accessed on 13-12-2013.

[48] 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*, 2007, pp. 552–561.

[49] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, “When good instructions go bad: generalizing return-oriented programming to RISC,” in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008, pp. 27–38.

[50] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” *ACM Transactions on Information and System Security*, vol. 15, no. 1, pp. 1–34, Mar. 2012.

[51] Blexim, “Basic integer overflows,” http://phrack.org/issues/60/10.html, Dec. 2002, accessed on 06-02-2015.

[52] O. Horovitz, “Big loop integer protection,” http://phrack.org/issues/60/9.html, Dec. 2002, accessed on 06-02-2015.

[53] gera and riq, “Advances in format string exploitation,” http://phrack.org/issues/59/7.html, Jul. 2002, accessed on 06-02-2015.

[54] C. Planet, “A eulogy for format strings,” http://www.phrack.org/issues/67/9.html#article, Nov. 2010, accessed on 11-02-2015.

[55] 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*, 2010, pp. 559–572.

[56] 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*, 2011, pp. 30–40.

[57] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting return-oriented programming malicious code," in *Information Systems Security*. Springer Berlin Heidelberg, 2009, pp. 163–177.

[58] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks," in *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, 2009, pp. 49–54.

[59] 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. Springer Berlin Heidelberg, Jan. 2011, vol. 6961, pp. 101–120.

[60] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie, "Efficient detection of the return-oriented programming malicious code," in *Information Systems Security*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2010, vol. 6503, pp. 140–155.

[61] 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.

[62] L. Davi, A.-R. 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*, 2011, pp. 40–51.

[63] 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*. ACM, 2010, pp. 195–208.

[64] 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*. ACM, 2011, pp. 353–362.

[65] 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 (SP)*, May 2012, pp. 601–615.

[66] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2011, vol. 6961, pp. 121–141.

[67] M. Conover, "w00w00 on heap overflows," http://hamsa.cs.northwestern.edu/media/readings/heap_overflows.pdf, 1999, accessed on 03-01-2018.

[68] Anonymous, "Once upon a free()..." http://phrack.org/issues/57/9.html, Nov. 2001, accessed on 06-02-2015.

[69] S. Designer, "JPEG COM marker processing vulnerability in netscape browsers and microsoft products, and a generic heap-based buffer overflow exploitation technique," http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability, Jul. 2000, accessed on 11-02-2015.

[70] M. Kaempf, "Vudo malloc tricks," http://phrack.org/issues/57/8.html, Nov. 2001, accessed on 11-02-2015.

[71] P. Phantasmagoria, “The malloc maleficarum,” https://www.securityfocus.com/archive/1/413007/30/0/threaded, 2005, accessed on 19-02-2014.

[72] Blackngel, “The malloc des-maleficarum,” http://www.phrack.org/issues.html?issue=66&id=10, Nov. 2009, accessed on 19-02-2014.

[73] huku and argp, “Pseudomonarchia jemallocum, or on exploiting the jemalloc memory manager,” http://www.phrack.org/issues/68/10.html#article, Apr. 2012, accessed on 11-02-2015.

[74] Redpantz, “Exploiting MS11-004 microsoft IIS 7.5 remote heap buffer overflow,” http://www.phrack.org/issues/68/12.html#article, Apr. 2012, accessed on 11-02-2015.

[75] huku and argp, “Exploiting VLC, a case study on jemalloc heap overflows,” http://www.phrack.org/issues/68/13.html#article, Apr. 2012, accessed on 11-02-2015.

[76] B. Hawkes, “Attacking the vista heap,” http://2008.ruxcon.org.au/files/2008/hawkes_ruxcon.pdf, 2008, accessed on 13-12-2013.

[77] P. Van Eeckhoutte, “Exploit writing tutorial part 11 : Heap spraying demystified,” https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/, Dec. 2011, accessed on 11-02-2015.

[78] ——, “DEPS – precise heap spray on firefox and IE10,” https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/, Feb. 2013, accessed on 11-02-2015.

[79] P. Ratanaworabhan, B. Livshits, and B. Zorn, “Nozzle: A defense against heap-spraying code injection attacks,” in *Proceedings of the 18th USENIX Security Symposium*, 2009, pp. 169–186.

[80] F. Gadaleta, Y. Younan, and W. Joosen, “BuBBle: A javascript engine level countermeasure against heap-spraying attacks,” in *Engineering Secure Software and Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2010, vol. 5965, pp. 1–17.

[81] J. Song, J. Song, and J. Kim, “Detection of heap-spraying attacks using string trace graph,” in *Information Security Applications*, ser. Lecture Notes in Computer Science. Springer International Publishing, Jan. 2015, pp. 17–26.

[82] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “A Theory of Secure Control Flow,” in *Formal Methods and Software Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2005, vol. 3785, pp. 111–124.

[83] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, “XFI: Software guards for system address spaces,” in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 75–88.

[84] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, “MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones.” in *Symposium on Network and Distributed System Security (NDSS)*, 2012, accessed on 04-01-2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/07_2.pdf

[85] B. Niu and G. Tan, “Monitor integrity protection with space efficiency and separate compilation,” in *Proceedings of the 20th ACM Conference on Computer and Communications Security*, 2013, pp. 199–210.

[86] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, L. Szekeres, S. McCamant, D. Song, and Wei Zou, “Practical Control Flow Integrity and Randomization for Binary Executables,” in *IEEE Symposium on Security and Privacy (SP)*, May 2013, pp. 559–573.

[87] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries." in *Proceedings of the 22nd USENIX Security Symposium*, 2013, pp. 337–352.

[88] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *IEEE Symposium on Security and Privacy (SP)*, May 2014, pp. 292–307.

[89] R. Gawlik and T. Holz, "Towards automated integrity protection of C++ virtual function tables in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 396–405.

[90] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "VTint: Defending virtual function tables integrity," in *Symposium on Network and Distributed System Security (NDSS)*, 2015, accessed on 05-01-2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/03_1_2.pdf

[91] P. Yuan, Q. Zeng, and X. Ding, "Hardware-Assisted Fine-Grained Code-Reuse Attack Detection," in *Proceedings of 18th International Symposium of Research in Attacks, Intrusions and Defenses*. Springer International Publishing, Nov. 2015, pp. 66–85.

[92] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz, "Opaque control-flow integrity," in *Symposium on Network and Distributed System Security (NDSS)*, 2015, accessed on 05-01-2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/06_3_2.pdf

[93] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-Flow Integrity," in *IEEE Symposium on Security and Privacy (SP)*, May 2014, pp. 575–589.

[94] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 401–416.

[95] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 385–399.

[96] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Security Symposium*, 2015, pp. 161–176.

[97] Zhi Wang and Xuxian Jiang, "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," in *IEEE Symposium on Security and Privacy (SP)*, May 2010, pp. 380–395.

[98] J. Pewny and T. Holz, "Control-flow restrictor: compiler-based CFI for iOS," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 309–318.

[99] D. Jang, Z. Tatlock, and S. Lerner, "SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks," in *Symposium on Network and Distributed System Security (NDSS)*, 2014, accessed on 05-01-2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/02_4_0.pdf

[100] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, . Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 941–955.

[101] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[102] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: hardware-assisted flow integrity extension," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, pp. 1–6.

[103] M. Payer, A. Barresi, and T. R. Gross, "Fine-Grained Control-Flow Integrity Through Binary Hardening," in *Proceedings of the 12th International Conference of Detection of Intrusions and Malware and Vulnerability Assessment.* Springer International Publishing, Jul. 2015, pp. 144–164.

[104] A. Prakash, X. Hu, and H. Yin, "vfGuard: Strict protection for virtual function calls in COTS C++ binaries," in *Symposium on Network and Distributed System Security (NDSS)*, 2015, accessed on 05-01-2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/11_2_2.pdf

[105] B. Niu and G. Tan, "Modular control-flow integrity," *SIGPLAN Notices*, vol. 49, no. 6, pp. 577–587, 2014.

[106] ——, "RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity," in *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014, pp. 1317–1328.

[107] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazieres, "CCFI: Cryptographically Enforced Control Flow Integrity," in *Proceedings of the 22nd ACM ACM Conference on Computer and Communications Security*, 2015, pp. 941–951.

[108] V. v. d. Veen, D. Andriesse, E. Goktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, 2015, pp. 927–940.

[109] B. Niu and G. Tan, "Per-Input Control-Flow Integrity," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, 2015, pp. 914–926.

[110] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-Grained Control-Flow Integrity for Kernel Software," in *Proceedings of 2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, Mar. 2016, pp. 179–194.

[111] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *Proceedings of the 24th USENIX Security Symposium*, 2015, pp. 177–192.

[112] J. E. Hopcroft, J. D. Ullman, and A. V. Aho, *The design and analysis of computer algorithms*. Addison-Wesley, 1975.

[113] J. van Benthem, H. van Ditmarsch, J. van Eijck, and J. Jaspars, "Chapter 6: Logic and Action," in *Logic in Action*, Feb. 2016, accessed on 05-01-2018.

[114] M. J. Fischer and R. E. Ladner, "Propositional dynamic logic of regular programs," *Journal of computer and system sciences*, vol. 18, no. 2, pp. 194–211, 1979.

[115] D. Harel, D. Kozen, and J. Tiuryn, *Propositional Dynamic Logic*. MIT Press, 2000, pp. 163–190.

[116] S. Andersen and V. Abella, "Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies," https://technet.microsoft.com/en-us/library/bb457155.aspx, Aug. 2004, accessed on 05-01-2018.

[117] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity." in *Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation*, 2014, pp. 147–163.

[118] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *IEEE Symposium on Security and Privacy (SP)*, May 2013, pp. 48–62.

[119] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kguard: Lightweight kernel protection against return-to-user attacks," in *Proceedings of the 21st USENIX Security Symposium*, 2012, pp. 459–474.

[120] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for c," *SIGPLAN Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[121] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the c programming language," *SIGPLAN Notices*, vol. 43, no. 3, pp. 103–114, 2008.

[122] R. Ramakesavan, D. Zimmerman, and P. Singaravelu, "Intel memory protection extensions (intel mpx) enabling guide," https://pdfs.semanticscholar.org/bd11/4878c6471cb5ae28546a594bf25ba5c25c6f.pdf, Apr. 2015, accessed on 05-01-2018.

[123] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," *SIGPLAN Notices*, vol. 45, no. 8, pp. 31–40, 2010.

[124] A. A. d. Amorim, M. Dns, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Micro-Policies: Formally Verified, Tag-Based Security Monitors," in *IEEE Symposium on Security and Privacy (SP)*, May 2015, pp. 813–830.

[125] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, J. Thomas F. Knight, B. C. Pierce, and A. DeHon, "Architectural Support for Software-Defined Metadata Processing," *SIGPLAN Notices*, vol. 50, no. 4, pp. 487–502, 2015.

[126] D. Volpano and G. Smith, “A type-based approach to program security,” in *TAPSOFT ’97: Theory and Practice of Software Development: 7th International Joint Conference CAAP/FASE*. Springer Berlin Heidelberg, Jun. 1997, pp. 607–621.

[127] P. Li and S. Zdancewic, “Downgrading policies and relaxed noninterference,” *SIGPLAN Notices*, vol. 40, no. 1, pp. 158–170, 2005.

[128] K. J. Biba, “Integrity considerations for secure computer systems,” MITRE CORP BEDFORD MA, Tech. Rep., Apr. 1977, accessed on 05-01-2018. [Online]. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a039324.pdf

[129] D. E. Bell and L. J. LaPadula, “Secure computer systems: Mathematical foundations,” MITRE CORP BEDFORD MA, Tech. Rep., Nov. 1973, accessed on 05-01-2018. [Online]. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/770768.pdf

[130] O. Goldreich, *Foundations of Cryptography: Volume 2, Basic Applications.* Cambridge university press, 2009.

[131] G. H. Kim and E. H. Spafford, “The design and implementation of tripwire: a file system integrity checker,” in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994, pp. 18–29.

[132] L. van Doorn, G. Ballintijn, and W. A. Arbaugh, “Design and implementation of signed executables for linux,” Technical Report CS-TR-4259, Computer Science Department, University of Maryland, Digital Repository at the University of Maryland, Tech. Rep., 2002. [Online]. Available: http://hdl.handle.net/1903/1139

[133] L. Catuogno and I. Visconti, “A Format-Independent Architecture for Run-Time Integrity Checking of Executable Code,” in *Security in Communication Networks*. Springer Berlin Heidelberg, Jan. 2003, vol. 2576, pp. 219–233.

[134] A. Apvrille, D. Gordon, S. E. Hallyn, M. Pourzandi, and V. Roy, "DigSig: Runtime Authentication of Binaries at Kernel Level." in *Proceedings of the 18th USENIX Conference on System Administration*, 2004, pp. 59–66.

[135] S. Minagar, "Efficient code verification prior execution," Monash University, Minor Thesis, Jun. 2012.

[136] A. Di Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "How the elf ruined christmas," in *Proceedings of the 24th USENIX Security Symposium*, 2015, pp. 643–658.

[137] S. A. Carr and M. Payer, "DataShield: Configurable Data Confidentiality and Integrity," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 193–204.

Appendix A

Page-by-Page Verification for Linux Kernel

A.1 Introduction

In this appendix the complete code of the implemented page-by-page verification for the selected Linux kernel is provided. The required steps to compile the kernel with modified files and the required steps to generate the secure hash values for all pages of the executable files within the system is also discussed.

A.2 Modification of Linux Kernel

In this section first an overview of the changes made to the source of the selected Linux kernel is provided. The details of each added, header file, variable, or function is then discussed in more detail in the following subsections.

A.2.1 Modification of main.c

The content of the modified `main.c` under the `init` folder in kernel source is shown in Listing A.1. For brevity and convenience part of the code that has not changed is omitted in such a way that it would be clear where the added functions are located

within the body of the code. Comments are added to help further identify the added code.

Listing A.1: C code of init/main.c

```
/*
 *  linux/init/main.c
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 *
 *  GK 2/5/95  -  Changed to support mounting root fs via NFS
 *  Added initrd & change_root: Werner Almesberger & Hans Lermen,
    Feb '96
 *  Moan early if gcc is old, avoiding bogus kernels - Paul
    Gortmaker, May '96
 *  Simplified starting of init:  Michael A. Griffith <grif@acm.org
    >
 */

#include <linux/types.h>
#include <linux/module.h>
/*
*********
*OMITTED*
*********

*/



#include <linux/file.h>
#include <linux/ptrace.h>

// Page-by-Page Verification: Modifications begin here
// added required external functions and variables
#include <linux/integrity_check.h>
// Page-by-Page Verification: Modifications end here
```

```
#include <asm/io.h>
#include <asm/bugs.h>
#include <asm/setup.h>
/*
*********
*OMITTED*
*********

*/

/*
 * Unknown boot options get handed to init, unless they look like
 * unused parameters (modprobe will find them in /proc/cmdline).
 */
static int __init unknown_bootoption(char *param, char *val, const
   char *unused)
{
  repair_env_string(param, val, unused);

  /* Handle obsolete-style parameters */
  if (obsolete_checksetup(param))
    return 0;

  /* Unused module parameter. */
  if (strchr(param, '.') && (!val || strchr(param, '.') < val))
    return 0;

  if (panic_later)
    return 0;

  if (val) {
    /* Environment option */
    unsigned int i;
    for (i = 0; envp_init[i]; i++) {
```

```
      if (i == MAX_INIT_ENVS) {
        panic_later = "Too many boot env vars at ‘%s’";
        panic_param = param;
      }
      if (!strncmp(param, envp_init[i], val - param))
        break;
    }
    envp_init[i] = param;
  } else {
    /* Command line option */
    unsigned int i;
    for (i = 0; argv_init[i]; i++) {
      if (i == MAX_INIT_ARGS) {
        panic_later = "Too many boot init vars at ‘%s’";
        panic_param = param;
      }
    }
    argv_init[i] = param;
  }
  return 0;
}

// Page-by-Page Verification: Modifications begin here
/* First define a variable for initrd key passed as boot option
 * then read the passed value as a key (256-bit) and store in
    initrd_key global vbariable
 */

char initrd_boot_option_key[65];

u8 initrd_key[32];
EXPORT_SYMBOL(initrd_key);

static int __init set_initrd_key(char *str)
{
```

```
  int i,j, err = 0;
  strncpy(initrd_boot_option_key, str, 64);
  initrd_boot_option_key[64] = '\0';
  unsigned char c1, c2;

  j = 0;
  for (i = 0; i < strlen(initrd_boot_option_key); i = i + 2) {
    c1 = hexval((unsigned char) initrd_boot_option_key[i]);
    c2 = hexval((unsigned char) initrd_boot_option_key[i + 1]);
    if ((c1 < 0) || (c2 < 0)) {
      printk("Bad parameter:%s", initrd_boot_option_key);
      err = 1;
      break;
    }
    c1 = c1 << 4;
    initrd_key[j] = c1 | c2;
    j++;
  }
  return err;
}

__setup("initrd_key=", set_initrd_key);


/* Read the boot option passed as encrypted_ipass
 * and store it in an init data variable with the same name
 * then get the password from user during boot
 * generate key and iv from the provided password
 * verify the entered password and the encrypted password provided
     via boot option
 */
char __initdata encrypted_ipass[MAX_INTEGRITY_PASSWORD_LEN+1];

static void __init set_enc_ipass(char *str)
```

```
{
  strncpy(encrypted_ipass, str, MAX_INTEGRITY_PASSWORD_LEN);
  encrypted_ipass[MAX_INTEGRITY_PASSWORD_LEN] = '\0';
  return;
}

__setup("encrypted_ipass=", set_enc_ipass);

/* gets the passphrase and verifies if it is entered correctly */
static void __init get_integrity_pass(void)
{
  char integpass[MAX_INTEGRITY_PASSWORD_LEN + 1], c, enc_pass[33],
      dec_pass[33],sha1_md[20];
  unsigned char c1,c2;
  int fd,i = 0,err, retry = 0, match = 0,j,k;
  struct termios termios;
  u8 keyiv[32];
  enc_pass[32] = '\0';
  //Reading the password from the console, since it is early using
      low level /dev/console method
  fd = sys_open("/dev/console", O_RDWR, 0);
  if (fd >= 0) {
    while ((!match) && (retry < 3)) {
      printk(KERN_NOTICE "Enter integrity password:");
      c = '\0';
      sys_ioctl(fd, TCGETS, (long)&termios);
      termios.c_lflag |= ICANON;
      //prevent the password to echo when being typed
      termios.c_lflag &= ~ECHO;
      sys_ioctl(fd, TCSETSF, (long)&termios);
      i = 0;
      while ( i < MAX_INTEGRITY_PASSWORD_LEN - 1 ) {
        sys_read(fd, &c, 1);
        if ( c == '\n') {
          integpass[i] = '\0';
```

```
      break;
    }
    integpass[i] = c;
    i++;
  }
  integpass[MAX_INTEGRITY_PASSWORD_LEN] = '\0';
  /* Calculate the SHA1 message digest of the entered password
     */
  err = sha1_hash((u8 *) integpass, (u8 *) sha1_md, i);

  /* Generate the key and IV from entered password */
  err = get_key_iv(integpass, keyiv);
  j = 0;
  for (i = 0; i < strlen(encrypted_ipass); i = i + 2) {
    c1 = hexval((unsigned char) encrypted_ipass[i]);
    c2 = hexval((unsigned char) encrypted_ipass[i + 1]);
    if ((c1 < 0) || (c2 < 0)) {
      printk("Bad parameter:%s", encrypted_ipass);
      break;
    }
    c1 = c1 << 4;
    enc_pass[j] = c1 | c2;
    j++;
  }
  /* Decrypt the provided boot option via encrypted_ipass to
     get the SHA1 message digest */
  dec_blkcipher(enc_pass, keyiv , (u8 *) dec_pass, keyiv + 16 ,
      32);

  /* Compare the two values */
  match = 1;
  for (k = 0; k < 20; k++)
    if (dec_pass[k] != sha1_md[k]) {
      match = 0;
      break;
```

```
        }
      retry++;
    }
    termios.c_lflag |= ECHO;
    sys_ioctl(fd, TCSETSF, (long)&termios);
    sys_close(fd);
  }
  if (!match)
    panic("Wrong password, kernel will crash :)\n");
  else
    set_integrity_pass(integpass);
}
//Page-by-Page Verification: Modifications end here

static int __init init_setup(char *str)
/*
*********
*OMITTED*
*********

*/



static noinline void __init kernel_init_freeable(void)
{
  /*
   * Wait until kthreadd is all set-up.
   */
  wait_for_completion(&kthreadd_done);

  /* Now the scheduler is fully set up and can do blocking
     allocations */
  gfp_allowed_mask = __GFP_BITS_MASK;

  /*
```

```
 * init can allocate pages on any node
 */
set_mems_allowed(node_states[N_MEMORY]);
/*
 * init can run on any cpu.
 */
set_cpus_allowed_ptr(current, cpu_all_mask);

cad_pid = task_pid(current);

smp_prepare_cpus(setup_max_cpus);

do_pre_smp_initcalls();
lockup_detector_init();

smp_init();
sched_init_smp();

do_basic_setup();

/* Open the /dev/console on the rootfs, this should never fail */
if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) <
   0)
  printk(KERN_WARNING "Warning: unable to open an initial console
     .\n");

// Page-by-Page Verification: Modifications begin here
//a short delay to allow the password prompt to be seen by user
ssleep(2);
preempt_disable();
get_integrity_pass();
preempt_enable();
// Page-by-Page Verification: Modifications end here

(void) sys_dup(0);
```

```
  (void) sys_dup(0);
  /*
   * check if there is an early userspace init.  If yes, let it do
       all
   * the work
   */

  if (!ramdisk_execute_command)
    ramdisk_execute_command = "/init";

  if (sys_access((const char __user *) ramdisk_execute_command, 0)
      != 0) {
    ramdisk_execute_command = NULL;
    prepare_namespace();
  }

  /*
   * Ok, we have completed the initial bootup, and
   * we're essentially up and running. Get rid of the
   * initmem segments and start the user-mode stuff..
   */
}

}
```

A.2.2 Required Changes in main.c

The modification starts with adding a header file `inetgrity_check.h` to the `main.c` The header contains the definition of the external functions and memory structures defined in `memory.c` and referenced in `main.c`. The Listing A.2 shows the content of the added header file.

Listing A.2: Code of integrity_check.h

```
#define MAX_INTEGRITY_PASSWORD_LEN 64
extern char integrity_pass[];
```

```
extern int integrity_password_is_set;
extern void set_integrity_pass(char *integ_pass);
extern int get_key_iv(char *passphrase, u8 *keyiv);
extern int dec_blkcipher(u8 *block, u8 *key, u8 *out, u8 *iv, int
    len);
extern unsigned char hexval(unsigned char in);
extern int sha1_hash(u8 *in, u8 *out, int len);
extern int sha256_hash(u8 *in, u8 *out, int len);
extern void printhex(unsigned char *in, int size);
}
```

To read the **initrd_key** boot command line option first an array of characters must be defined to contain the passed value. The binary value of the hex key must be then evaluated and stored in a global variable to be used in page verification function. The Listing A.3 shows the required code to define the **initrd_key** variable.

Listing A.3: Definition of required variables for initrd_key boot option and the corresponding global variable

```
char initrd_boot_option_key[65];
u8 initrd_key[32];
EXPORT_SYMBOL(initrd_key);
```

The function **set_initrd_key()** receives the boot option as a string to be considered as a hex value and calculates its binary value (listing A.4).

Listing A.4: C code of set_initrd_key()

```
static int __init set_initrd_key(char *str)
{

  int i,j, err = 0;
  strncpy(initrd_boot_option_key, str, 64);
  initrd_boot_option_key[64] = '\0';
  unsigned char c1, c2;

  j = 0;
  for (i = 0; i < strlen(initrd_boot_option_key); i = i + 2) {
```

```
    c1 = hexval((unsigned char) initrd_boot_option_key[i]);
    c2 = hexval((unsigned char) initrd_boot_option_key[i + 1]);
    if ((c1 < 0) || (c2 < 0)) {
      printk("Bad parameter:%s", initrd_boot_option_key);
      err = 1;
      break;
    }
    c1 = c1 << 4;
    initrd_key[j] = c1 | c2;
    j++;
  }
  return err;
}

__setup("initrd_key=", set_initrd_key);
}
```

The `__setup()` function associates a function with a given boot parameter where the associated function receives the value of the boot parameter as a variable and performs the necessary operations. The `encrypted_ipass` boot option is similar to `initrd_key` boot option in the fact that the content is considered a hexadecimal value and must be evaluated to its binary value. An initial variable is needed to store the provided boot option value which will be used in verifying the entered password by the user. Given the plaintext to the encryption algorithm is SHA1 of the password, the input length to CBC-AES is 160 bits which will be padded to two blocks of 32 bytes. Listing A.5 shows the required code to read and store the `encrypted_ipass` boot option.

Listing A.5: Definition of initialisation data to store boot option encrypted_ipass and associated function set_enc_ipass()

```
char __initdata encrypted_ipass[65];

static void __init set_enc_ipass(char *str)
{
```

```
  strncpy(encrypted_ipass, str, 64);
  encrypted_ipass[64] = '\0';
  return;
}

__setup("encrypted_ipass=", set_enc_ipass);
}
```

The provided boot option is then used to verify the entered password by the user. Listing A.6 shows the code to read the password from the low level `/dev/console` without echo. The entered password is then verified by generating SHA256 of the entered password to be used as key and IV to decrypt the binary value of passed `encrypted_ipass` as boot option. If the decrypted value matches the SHA1 of the entered password the provided password is correct otherwise the user can attempt another password up to maximum three tries.

Listing A.6: C code of get_integrity_pass()

```
static void __init get_integrity_pass(void)
{
  char integpass[MAX_INTEGRITY_PASSWORD_LEN + 1], c, enc_pass[33],
     dec_pass[33],sha1_md[20];
  unsigned char c1,c2;
  int fd,i = 0,err, retry = 0, match = 0,j,k;
  struct termios termios;
  u8 keyiv[32];
  enc_pass[32] = '\0';
  //Reading the password from the console, since it is early using
     low level /dev/console method
  fd = sys_open("/dev/console", O_RDWR, 0);
  if (fd >= 0) {
    while ((!match) && (retry < 3)) {
      printk(KERN_NOTICE "Enter integrity password:");
      c = '\0';
      sys_ioctl(fd, TCGETS, (long)&termios);
      termios.c_lflag |= ICANON;
```

```
//prevent the password to echo when being typed
termios.c_lflag &= ~ECHO;
sys_ioctl(fd, TCSETSF, (long)&termios);
i = 0;
while ( i < MAX_INTEGRITY_PASSWORD_LEN - 1 ) {
  sys_read(fd, &c, 1);
  if ( c == '\n') {
    integpass[i] = '\0';
    break;
  }
  integpass[i] = c;
  i++;
}
integpass[MAX_INTEGRITY_PASSWORD_LEN] = '\0';
/* Calculate the SHA1 message digest of the entered password
   */
err = sha1_hash((u8 *) integpass, (u8 *) sha1_md, i);

/* Generate the key and IV from entered password */
err = get_key_iv(integpass, keyiv);
j = 0;
for (i = 0; i < strlen(encrypted_ipass); i = i + 2) {
  c1 = hexval((unsigned char) encrypted_ipass[i]);
  c2 = hexval((unsigned char) encrypted_ipass[i + 1]);
  if ((c1 < 0) || (c2 < 0)) {
    printk("Bad parameter:%s", encrypted_ipass);
    break;
  }
  c1 = c1 << 4;
  enc_pass[j] = c1 | c2;
  j++;
}
/* Decrypt the provided boot option via encrypted_ipass to
   get the SHA1 message digest */
```

```
      dec_blkcipher(enc_pass, keyiv , (u8 *) dec_pass, keyiv + 16 ,
          32);

      /* Compare the two values */
      match = 1;
      for (k = 0; k < 20; k++)
        if (dec_pass[k] != sha1_md[k]) {
          match = 0;
          break;
        }
      retry++;
    }
    termios.c_lflag |= ECHO;
    sys_ioctl(fd, TCSETSF, (long)&termios);
    sys_close(fd);
  }
  if (!match)
    panic("Wrong password, kernel will crash :)\n");
  else
    set_integrity_pass(integpass);
}
}
```

The process of reading the password from console must be added to the boot sequence by adding a function call to the `get_integrity_pass()` function at an appropriate time where the necessary code exists to perform the low level read from the `/dev/console`. The call is added to the function `kernel_init_freeable()` as shown in Listing A.7.

Listing A.7: C code of kernel_init_freeable()

```
static noinline void __init kernel_init_freeable(void)
{
  /*
   *********
   *OMITTED*
```

```
     *********
     */

    do_basic_setup();

    /* Open the /dev/console on the rootfs, this should never fail */
    if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) <
        0)
      printk(KERN_WARNING "Warning: unable to open an initial console
          .\n");

    // Page-by-Page Verification: Modifications begin here
    //a short delay to allow the password prompt to be seen by user
    ssleep(2);
    preempt_disable();
    get_integrity_pass();
    preempt_enable();
    // Page-by-Page Verification: Modifications end here

    /*
     *********
     *OMITTED*
     *********
     */
}

}
```

A.2.3 Code of Modified memory.c

The content of the modified `mm/memory.c` file in kernel source is shown in Listing A.8. Similar to the `init/main.c` file for brevity part of the file content is omitted.

Listing A.8: C code of mm/memory.c

```
/*
```

```
 *  linux/mm/memory.c
 *
 *  Copyright (C) 1991, 1992, 1993, 1994  Linus Torvalds
 */

  /*
   *********
   *OMITTED*
   *********
   */

#include <linux/gfp.h>
#include <linux/migrate.h>
#include <linux/string.h>

  /*
   *********
   *OMITTED*
   *********
   */

/* Page-by-Page Verification: Modifications begin here
required header files
*/
#include <linux/delay.h>
#include <linux/crypto.h>
#include <crypto/sha.h>
#include <linux/scatterlist.h>
#include <linux/mm_types.h>
#include <linux/sched.h>
#include <linux/binfmts.h>
#include <linux/fs.h>

#define MAX_INTEGRITY_PASSWORD_LEN 64
/* Page-by-Page Verification: Modifications end here */
```

```
#include <asm/io.h>
#include <asm/pgalloc.h>
#include <asm/uaccess.h>
#include <asm/tlb.h>
#include <asm/tlbflush.h>
#include <asm/pgtable.h>
#include "internal.h"
/* Page-by-Page Verification: Modifications begin here */

//structure to store the integrity passowrd and generated SHA256 of
    the password
static struct integrity_secret_info {
  char integrity_pass[MAX_INTEGRITY_PASSWORD_LEN + 1];
  u8 key[SHA256_DIGEST_SIZE];
  } integ_sec;

int integrity_password_is_set = 0;
extern char initrd_pass[];
extern u8 initrd_key[];

void printhex(unsigned char *in, int size) {
  int i;
  for (i = 0; i < size; i++) {
    printk("%02x", in[i]);
  }
  return;
}

/* Generates 256 bits SHA256 message digest from passphrase which
   is used as Key and IV */
int get_key_iv(char *passphrase, u8 *keyiv) {
  struct hash_desc desc;
  int ret = 0;
  struct scatterlist hash_sg;
```

```
  sg_init_one(&hash_sg, passphrase, strlen(passphrase));
  desc.tfm = crypto_alloc_hash("sha256", 0, CRYPTO_ALG_ASYNC);
  ret = IS_ERR(desc.tfm);
  if (ret) {
    return -ret;
  }
  desc.flags = 0;
  ret = crypto_hash_digest(&desc, &hash_sg, strlen(passphrase),
     keyiv);
  if (ret) {
    return -ret;
  }
  return 0;
}

/* Used in init/main.c to set the values of integrity_secret_info
    during boot */
void set_integrity_pass(char *integ_pass)
{
  int len,err;
  if (strlen(integ_pass) <= MAX_INTEGRITY_PASSWORD_LEN)
    len = strlen(integ_pass);
  else
    len = MAX_INTEGRITY_PASSWORD_LEN;
  strncpy(integ_sec.integrity_pass, integ_pass, len);
  integ_sec.integrity_pass[MAX_INTEGRITY_PASSWORD_LEN] = '\0';
  integrity_password_is_set = 1;


  err = get_key_iv(integ_sec.integrity_pass, integ_sec.key);
  if (err) {
    printk("Error generating key and iv from passphrase\n");
  }
```

```
  return;
}

int dec_blkcipher(u8 *block, u8 *key, u8 *out, u8 *iv, int len) {
  struct crypto_blkcipher *tfm_blkcipher;
  struct blkcipher_desc desc;
  int ret = 0;
  struct scatterlist cipher_in, cipher_out;

  sg_init_one(&cipher_in, block, 32);
  sg_init_one(&cipher_out, out, len);

  tfm_blkcipher = crypto_alloc_blkcipher("cbc(aes)",
     CRYPTO_ALG_TYPE_BLKCIPHER,0);
  desc.tfm = tfm_blkcipher;
  ret = IS_ERR(tfm_blkcipher);
  if (ret) {
    printk("Error allocating CBC_AES encryption algorithm\n");
    return ret;
  }
  ret = crypto_blkcipher_setkey(tfm_blkcipher, key, 16);
  if (ret) {
    printk("Error setting encryption key\n");
    return ret;
  }
  crypto_blkcipher_set_iv(tfm_blkcipher, iv, 16);
  crypto_blkcipher_decrypt(&desc, &cipher_out, &cipher_in, len);
  crypto_free_blkcipher(tfm_blkcipher);

  return 0;
}

/* converts a hex digit to its binary value */

unsigned char hexval(unsigned char in) {
```

```
  if ((in >= '0') && (in <= '9'))
    return in - '0';
  if ((in >= 'a') && (in <= 'f'))
    return in - 'a' + 10;
  if ((in >= 'A') && (in <= 'F'))
    return in - 'A' + 10;

  return -1;
}

int sha1_hash(u8 *in, u8 *out, int len) {
  struct scatterlist hash_sg;
  struct crypto_hash *tfm;
  struct hash_desc desc;
  int ret;

  sg_init_one(&hash_sg, in, len);
  desc.tfm = crypto_alloc_hash("sha1", 0, CRYPTO_ALG_ASYNC);
  ret = IS_ERR(desc.tfm);
  if (ret) {
    printk("Crypto hash allocation for sha1 failed\n");
    return -ret;
  }
  desc.flags = 0;

  ret = crypto_hash_init(&desc);
  if (ret) {
    printk("Error initializing crypto hash\n");
    goto out_with_error;
  }
  ret = crypto_hash_update(&desc, &hash_sg, len);
  if (ret) {
    printk("Error updating crypto hash\n");
    goto out_with_error;
```

```
  }
  ret = crypto_hash_final(&desc, out);
  if (ret) {
    printk("Error finalizing crypto hash\n");
    goto out_with_error;
  }
  crypto_free_hash(tfm);
  return 0;
out_with_error:
  crypto_free_hash(tfm);
  return -ret;
}

int sha256_hash(u8 *in, u8 *out, int len) {
  struct scatterlist hash_sg;
  struct crypto_hash *tfm;
  struct hash_desc desc;
  int ret;

  sg_init_one(&hash_sg, in, len);
  desc.tfm = crypto_alloc_hash("sha256", 0, CRYPTO_ALG_ASYNC);
  ret = IS_ERR(desc.tfm);
  if (ret) {
    printk("Crypto hash allocation for sha256 failed\n");
    return -ret;
  }
  desc.flags = 0;

  ret = crypto_hash_init(&desc);
  if (ret) {
    printk("Error initializing crypto hash\n");
    goto out_with_error;
  }
  ret = crypto_hash_update(&desc, &hash_sg, len);
  if (ret) {
```

```
    printk("Error updating crypto hash\n");
    goto out_with_error;
  }
  ret = crypto_hash_final(&desc, out);
  if (ret) {
    printk("Error finalizing crypto hash\n");
    goto out_with_error;
  }
  crypto_free_hash(tfm);
  return 0;
out_with_error:
  crypto_free_hash(tfm);
  return -ret;
}

int sha256_mac(u8 *in, u8 *out, int len, u8 *key, int keylen) {
  struct scatterlist hash_sg, key_sg;
  struct hash_desc sechash_desc;
  int ret;

  sg_init_one(&hash_sg, in, len);
  sg_init_one(&key_sg, key, keylen);

  sechash_desc.tfm = crypto_alloc_hash("sha256", 0,
     CRYPTO_ALG_TYPE_SHASH);
  ret = IS_ERR(sechash_desc.tfm);
  if (ret) {
    printk("Crypto hash allocation for sha256 failed\n");
    return -ret;
  }

  sechash_desc.flags = 0;

  ret = crypto_hash_init(&sechash_desc);
  if (ret) {
```

```
    printk("Error initializing crypto hash\n");
    goto out_with_error;
  }

  ret = crypto_hash_update(&sechash_desc, &key_sg, keylen);
  if (ret) {
    printk("Error updating crypto hash\n");
    goto out_with_error;
  }

  ret = crypto_hash_update(&sechash_desc, &hash_sg, len);
  if (ret) {
    printk("Error updating crypto hash\n");
    goto out_with_error;
  }

  ret = crypto_hash_final(&sechash_desc, out);
  if (ret) {
    printk("Error finalizing crypto hash\n");
    goto out_with_error;
  }

  crypto_free_hash(sechash_desc.tfm);
  return 0;
out_with_error:
  crypto_free_hash(sechash_desc.tfm);
  return -ret;
}

/* Page-by-Page Verification: Modifications end here*/

#ifndef CONFIG_NEED_MULTIPLE_NODES
/* use the per-pgdat data instead for discontigmem - mbligh */
unsigned long max_mapnr;
struct page *mem_map;
```

```
EXPORT_SYMBOL(max_mapnr);
EXPORT_SYMBOL(mem_map);
#endif

  /*
   *********
   *OMITTED*
   *********
   */

/*
 * We enter with non-exclusive mmap_sem (to exclude vma changes,
 * but allow concurrent faults), and pte mapped but not yet locked.
 * We return with mmap_sem still held, but pte unmapped and
    unlocked.
 */
static int do_anonymous_page(struct mm_struct *mm, struct
   vm_area_struct *vma,
    unsigned long address, pte_t *page_table, pmd_t *pmd,
    unsigned int flags)
{
  struct page *page;
  spinlock_t *ptl;
  pte_t entry;

  pte_unmap(page_table);

  /* Check if we need to add a guard page to the stack */
  if (check_stack_guard_page(vma, address) < 0)
    return VM_FAULT_SIGBUS;

  /* Use the zero-page for reads */
  if (!(flags & FAULT_FLAG_WRITE)) {
    entry = pte_mkspecial(pfn_pte(my_zero_pfn(address),
```

```
            vma->vm_page_prot));
    page_table = pte_offset_map_lock(mm, pmd, address, &ptl);
    if (!pte_none(*page_table))
      goto unlock;
    goto setpte;
  }

  /* Allocate our own private page. */
  if (unlikely(anon_vma_prepare(vma)))
    goto oom;
  page = alloc_zeroed_user_highpage_movable(vma, address);
  if (!page)
    goto oom;
  __SetPageUptodate(page);

  if (mem_cgroup_newpage_charge(page, mm, GFP_KERNEL))
    goto oom_free_page;

  entry = mk_pte(page, vma->vm_page_prot);
  if (vma->vm_flags & VM_WRITE)
    entry = pte_mkwrite(pte_mkdirty(entry));

  page_table = pte_offset_map_lock(mm, pmd, address, &ptl);
  if (!pte_none(*page_table))
    goto release;

  inc_mm_counter_fast(mm, MM_ANONPAGES);
  page_add_new_anon_rmap(page, vma, address);
setpte:
  set_pte_at(mm, address, page_table, entry);

  /* No need to invalidate - it was non-present before */
  update_mmu_cache(vma, address, page_table);
unlock:
  pte_unmap_unlock(page_table, ptl);
```

```
  return 0;
release:
  mem_cgroup_uncharge_page(page);
  page_cache_release(page);
  goto unlock;
oom_free_page:
  page_cache_release(page);
oom:
  return VM_FAULT_OOM;
}
//Page-by-Page Verification: Modifications begin here
// -- function added:
int do_verify_page_mac(char *filename, struct page *page)
{

  u8 *macofpage, *filenamesha2;
  int err = 0, i, k, j;

  unsigned char *addr, *hashdir, *sig;
  struct file *file;
  u8 *key;

  if (unlikely(!integrity_password_is_set)) {
    key = initrd_key;
  }
  else {
    key = integ_sec.key;
  }

  filenamesha2 = kzalloc(SHA256_DIGEST_SIZE, GFP_KERNEL);
  if (!filenamesha2) {
    printk("Cannot allocate memory for hash of the page\n");
    return VM_FAULT_OOM;
  }
```

```
err = sha256_hash((u8 *) filename, filenamesha2, strlen(filename)
    );
if (err) {
  printk("Generating SHA256 hash of the filename failed!\n");
  return -EPERM;
}

hashdir = kzalloc(73, GFP_KERNEL);
if (!hashdir) {
  return VM_FAULT_OOM;
}
sprintf(hashdir, "/hashes/");
k = 8;
for (i = 0; i < SHA256_DIGEST_SIZE; i++) {
  sprintf(hashdir + k, "%02x", (unsigned char) *(filenamesha2 + i
      ));
  k = k + 2;
}
hashdir[73] = '\0';

macofpage = kzalloc(SHA256_DIGEST_SIZE, GFP_KERNEL);
if (!macofpage){
  printk("Cannot allocate memory for hash of the page\n");
  return VM_FAULT_OOM;
}
addr = page_address(page);

err = sha256_mac((u8 *) addr, macofpage, PAGE_SIZE, key,
    SHA256_DIGEST_SIZE);
if (err) {
  kfree(macofpage);
  kfree(hashdir);
  return -EACCES;
}
sig = kzalloc((SHA256_DIGEST_SIZE * 2) + 74, GFP_KERNEL);
```

```
  sprintf(sig, "%s/", hashdir);
  k = 73;
  for (j = 0; j < SHA256_DIGEST_SIZE; j++){
    sprintf(sig + k, "%02x", (unsigned char) *(macofpage + j));
    k = k + 2;
    }
  sig[(SHA256_DIGEST_SIZE * 2) + 74] = '\0';

  file = filp_open(sig, O_RDONLY, FMODE_READ);
  err = IS_ERR(file);
  if (err) {
    kfree(hashdir);
    kfree(filenamesha2);
    kfree(macofpage);
    kfree(sig);
    return -EPERM;
  }

  filp_close(file, NULL);
  kfree(hashdir);
  kfree(filenamesha2);
  kfree(macofpage);
  kfree(sig);

  return 0;

}
//Page-by-Page Verification: Modifications end here

/*
 * __do_fault() tries to create a new page mapping. It aggressively
 * tries to share with existing pages, but makes a separate copy if
 * the FAULT_FLAG_WRITE is set in the flags parameter in order to
   avoid
 * the next page fault.
```

```
 *
 * As this is called only for pages that do not currently exist, we
 * do not need to flush old virtual caches or the TLB.
 *
 * We enter with non-exclusive mmap_sem (to exclude vma changes,
 * but allow concurrent faults), and pte neither mapped nor locked.
 * We return with mmap_sem still held, but pte unmapped and
    unlocked.
 */
static int __do_fault(struct mm_struct *mm, struct vm_area_struct *
   vma,
    unsigned long address, pmd_t *pmd,
    pgoff_t pgoff, unsigned int flags, pte_t orig_pte)
{
  pte_t *page_table;
  spinlock_t *ptl;
  struct page *page;
  struct page *cow_page;
  pte_t entry;
  int anon = 0;
  struct page *dirty_page = NULL;
  struct vm_fault vmf;
  int ret = 0;
  int page_mkwrite = 0;

  /*Page-by-Page Verification: Modifications begin here
  defining required variables: */
  int err = 0;
  unsigned char *buf;
  char *filename;
  //Page-by-Page Verification: Modifications end here
  /*
   * If we do COW later, allocate page befor taking lock_page()
   * on the file cache page. This will reduce lock holding time.
   */
```

```
if ((flags & FAULT_FLAG_WRITE) && !(vma->vm_flags & VM_SHARED)) {

  if (unlikely(anon_vma_prepare(vma)))
    return VM_FAULT_OOM;

  cow_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma, address);
  if (!cow_page)
    return VM_FAULT_OOM;

  if (mem_cgroup_newpage_charge(cow_page, mm, GFP_KERNEL)) {
    page_cache_release(cow_page);
    return VM_FAULT_OOM;
  }
} else
  cow_page = NULL;

vmf.virtual_address = (void __user *)(address & PAGE_MASK);
vmf.pgoff = pgoff;
vmf.flags = flags;
vmf.page = NULL;

ret = vma->vm_ops->fault(vma, &vmf);
if (unlikely(ret & (VM_FAULT_ERROR | VM_FAULT_NOPAGE |
        VM_FAULT_RETRY)))
  goto uncharge_out;

if (unlikely(PageHWPoison(vmf.page))) {
  if (ret & VM_FAULT_LOCKED)
    unlock_page(vmf.page);
  ret = VM_FAULT_HWPOISON;
  goto uncharge_out;
}

/*
```

```
 * For consistency in subsequent calls, make the faulted page
    always
 * locked.
 */
if (unlikely(!(ret & VM_FAULT_LOCKED)))
  lock_page(vmf.page);
else
  VM_BUG_ON(!PageLocked(vmf.page));



page = vmf.page;
/*Page-by-Page Verification: Modifications begin here
For executable pages verify the authenticity of the page */

if (vma->vm_flags & VM_EXEC) {
  if (vma->vm_file != NULL) {
    buf = kzalloc(PATH_MAX, GFP_KERNEL);
    if (!buf)
      return VM_FAULT_OOM;
    filename = dentry_path(vma->vm_file->f_dentry, buf, PATH_MAX)
        ;
    err = do_verify_page_mac(filename, page);
    if (err) {
      printk(KERN_WARNING "Secure hash verification failed for
          process %s, file:%s, page index:%lx\n", current->comm,
          filename, page->index);

      ret = err;
      if (buf)
        kfree(buf);
      goto out_verification_failed;
    }
    ret = 0;
    if (buf)
      kfree(buf);
```

```
    }
  }
  //Page-by-Page Verification: Modifications end here

  /*
   * Should we do an early C-O-W break?
   */
  if (flags & FAULT_FLAG_WRITE) {
    if (!(vma->vm_flags & VM_SHARED)) {
      page = cow_page;
      anon = 1;
      copy_user_highpage(page, vmf.page, address, vma);
      __SetPageUptodate(page);
    } else {
      /*
       * If the page will be shareable, see if the backing
       * address space wants to know that the page is about
       * to become writable
       */
      if (vma->vm_ops->page_mkwrite) {
        int tmp;

        unlock_page(page);
        vmf.flags = FAULT_FLAG_WRITE|FAULT_FLAG_MKWRITE;
        tmp = vma->vm_ops->page_mkwrite(vma, &vmf);
        if (unlikely(tmp &
            (VM_FAULT_ERROR | VM_FAULT_NOPAGE))) {
          ret = tmp;
          goto unwritable_page;
        }
        if (unlikely(!(tmp & VM_FAULT_LOCKED))) {
          lock_page(page);
          if (!page->mapping) {
            ret = 0; /* retry the fault */
            unlock_page(page);
```

```
          goto unwritable_page;
        }
      } else
        VM_BUG_ON(!PageLocked(page));
      page_mkwrite = 1;
    }
  }

}

page_table = pte_offset_map_lock(mm, pmd, address, &ptl);

/*
 * This silly early PAGE_DIRTY setting removes a race
 * due to the bad i386 page protection. But it's valid
 * for other architectures too.
 *
 * Note that if FAULT_FLAG_WRITE is set, we either now have
 * an exclusive copy of the page, or this is a shared mapping,
 * so we can make it writable and dirty to avoid having to
 * handle that later.
 */
/* Only go through if we didn't race with anybody else... */
if (likely(pte_same(*page_table, orig_pte))) {
  flush_icache_page(vma, page);
  entry = mk_pte(page, vma->vm_page_prot);
  if (flags & FAULT_FLAG_WRITE)
    entry = maybe_mkwrite(pte_mkdirty(entry), vma);
  if (anon) {
    inc_mm_counter_fast(mm, MM_ANONPAGES);
    page_add_new_anon_rmap(page, vma, address);
  } else {
    inc_mm_counter_fast(mm, MM_FILEPAGES);
    page_add_file_rmap(page);
    if (flags & FAULT_FLAG_WRITE) {
```

```
            dirty_page = page;
            get_page(dirty_page);
          }
        }
        set_pte_at(mm, address, page_table, entry);

        /* no need to invalidate: a not-present page won't be cached */
        update_mmu_cache(vma, address, page_table);
      } else {
        if (cow_page)
          mem_cgroup_uncharge_page(cow_page);
        if (anon)
          page_cache_release(page);
        else
          anon = 1; /* no anon but release faulted_page */
      }

      pte_unmap_unlock(page_table, ptl);

      if (dirty_page) {
        struct address_space *mapping = page->mapping;
        int dirtied = 0;

        if (set_page_dirty(dirty_page))
          dirtied = 1;
        unlock_page(dirty_page);
        put_page(dirty_page);
        if ((dirtied || page_mkwrite) && mapping) {
          /*
           * Some device drivers do not set page.mapping but still
           * dirty their pages
           */
          balance_dirty_pages_ratelimited(mapping);
        }
```

```
    /* file_update_time outside page_lock */
    if (vma->vm_file && !page_mkwrite)
      file_update_time(vma->vm_file);
  } else {
    unlock_page(vmf.page);
    if (anon)
      page_cache_release(vmf.page);
  }

  return ret;

unwritable_page:
  page_cache_release(page);
  return ret;
uncharge_out:
  /* fs's fault handler get error */
  if (cow_page) {
    mem_cgroup_uncharge_page(cow_page);
    page_cache_release(cow_page);
  }
  return ret;
out_verification_failed:
  force_sig(SIGKILL, current);
  return 0;

}

static int do_linear_fault(struct mm_struct *mm, struct
   vm_area_struct *vma,
  /*
   *********
   *OMITTED*
   *********
   */
```

A.2.4 Required Changes to memory.c

After the inclusion of the required header files, a memory structure is defined to store the page verification password and the SHA256 of the password. Listing A.9 shows the definition of this memory structure.

Listing A.9: Definition of memory structure to store page verification password and key

```
static struct integrity_secret_info {
  char integrity_pass[MAX_INTEGRITY_PASSWORD_LEN + 1];
  u8 key[SHA256_DIGEST_SIZE];
  } integ_sec;
}
```

The definition of the global variable `integrity_password_is_set` which indicates whether the `initrd_key` must be used to verify the pages or the system-wide key is shown in Listing A.10.

Listing A.10: Definition of global variable integrity_password_is_set and external variable initrd_key

```
int integrity_password_is_set = 0;
extern u8 initrd_key[];
}
```

The `get_key_iv()` is a wrapper function that generates the SHA256 message digest of the provided password which can be used as key and IV for CBC-AES algorithm.

Listing A.11: C code of get_key_iv()

```
int get_key_iv(char *passphrase, u8 *keyiv) {
  struct hash_desc desc;
  int ret = 0;
  struct scatterlist hash_sg;

  sg_init_one(&hash_sg, passphrase, strlen(passphrase));
  desc.tfm = crypto_alloc_hash("sha256", 0, CRYPTO_ALG_ASYNC);
```

```
  ret = IS_ERR(desc.tfm);
  if (ret) {
    return -ret;
  }
  desc.flags = 0;
  ret = crypto_hash_digest(&desc, &hash_sg, strlen(passphrase),
      keyiv);
  if (ret) {
    return -ret;
  }
  return 0;
}
```

The function `set_integrity_pass()`, shown in Listing A.12, receives the integrity password and sets up the memory structure of integrity verification by copying the integrity password and the SHA256 hash of the password to the defined memory location. This function is called from the `get_integrity_pass()` function within `init/main.c` file, after the entered password by user is verified to be correct.

Listing A.12: C code of set_integrity_pass()

```
void set_integrity_pass(char *integ_pass)
{
  int len,err;
  if (strlen(integ_pass) <= MAX_INTEGRITY_PASSWORD_LEN)
    len = strlen(integ_pass);
  else
    len = MAX_INTEGRITY_PASSWORD_LEN;
  strncpy(integ_sec.integrity_pass, integ_pass, len);
  integ_sec.integrity_pass[MAX_INTEGRITY_PASSWORD_LEN] = '\0';
  integrity_password_is_set = 1;


  err = get_key_iv(integ_sec.integrity_pass, integ_sec.key);
  if (err) {
    printk("Error generating key and iv from passphrase\n");
```

```
  }

  return;
}
```

The function `dec_blkcipher()` is a wrapper function that performs CBS-AES-128 decryption on a provided block of code given the key and IV and returns the plaintext stored in a memory location pointed to by the passed by reference pointer `in`. The code is shown in Listing A.13.

Listing A.13: C code of dec_blkcipher()

```
int dec_blkcipher(u8 *block, u8 *key, u8 *out, u8 *iv, int len) {
  struct crypto_blkcipher *tfm_blkcipher;
  struct blkcipher_desc desc;
  int ret = 0;
  struct scatterlist cipher_in, cipher_out;

  sg_init_one(&cipher_in, block, 32);
  sg_init_one(&cipher_out, out, len);

  tfm_blkcipher = crypto_alloc_blkcipher("cbc(aes)",
     CRYPTO_ALG_TYPE_BLKCIPHER,0);
  desc.tfm = tfm_blkcipher;
  ret = IS_ERR(tfm_blkcipher);
  if (ret) {
    printk("Error allocating CBC_AES encryption algorithm\n");
    return ret;
  }
  ret = crypto_blkcipher_setkey(tfm_blkcipher, key, 16);
  if (ret) {
    printk("Error setting encryption key\n");
    return ret;
  }
  crypto_blkcipher_set_iv(tfm_blkcipher, iv, 16);
  crypto_blkcipher_decrypt(&desc, &cipher_out, &cipher_in, len);
  crypto_free_blkcipher(tfm_blkcipher);
```

```
  return 0;
}
```

The function `hexval()` calculates the binary value of a hexadecimal digit passed as a character. This function is used to evaluate the `initrd_key` and `encrypted_ipass` boot option values into the corresponding binary value.

Listing A.14: C code of hexval()

```
unsigned char hexval(unsigned char in) {

  if ((in >= '0') && (in <= '9'))
    return in - '0';
  if ((in >= 'a') && (in <= 'f'))
    return in - 'a' + 10;
  if ((in >= 'A') && (in <= 'F'))
    return in - 'A' + 10;

  return -1;
}
```

The code for generating the SHA1 and SHA256 message digest is shown in Listing A.15 and Listing A.16.

Listing A.15: C code of sha1_hash()

```
int sha1_hash(u8 *in, u8 *out, int len) {
  struct scatterlist hash_sg;
  struct crypto_hash *tfm;
  struct hash_desc desc;
  int ret;

  sg_init_one(&hash_sg, in, len);
  desc.tfm = crypto_alloc_hash("sha1", 0, CRYPTO_ALG_ASYNC);
  ret = IS_ERR(desc.tfm);
  if (ret) {
    printk("Crypto hash allocation for sha1 failed\n");
```

```
    return -ret;
  }
  desc.flags = 0;

  ret = crypto_hash_init(&desc);
  if (ret) {
    printk("Error initializing crypto hash\n");
    goto out_with_error;
  }
  ret = crypto_hash_update(&desc, &hash_sg, len);
  if (ret) {
    printk("Error updating crypto hash\n");
    goto out_with_error;
  }
  ret = crypto_hash_final(&desc, out);
  if (ret) {
    printk("Error finalizing crypto hash\n");
    goto out_with_error;
  }
  crypto_free_hash(tfm);
  return 0;
out_with_error:
  crypto_free_hash(tfm);
  return -ret;
}
```

The SHA1 hash is used to verify the entered password during boot whereas the SHA256 is used to generate key and IV from the password for CBC-AES algorithm as well as generating the key for page MACs.

Listing A.16: C code of sha256_hash()

```
int sha256_hash(u8 *in, u8 *out, int len) {
  struct scatterlist hash_sg;
  struct crypto_hash *tfm;
  struct hash_desc desc;
  int ret;
```

```
  sg_init_one(&hash_sg, in, len);
  desc.tfm = crypto_alloc_hash("sha256", 0, CRYPTO_ALG_ASYNC);
  ret = IS_ERR(desc.tfm);
  if (ret) {
    printk("Crypto hash allocation for sha256 failed\n");
    return -ret;
  }
  desc.flags = 0;

  ret = crypto_hash_init(&desc);
  if (ret) {
    printk("Error initializing crypto hash\n");
    goto out_with_error;
  }
  ret = crypto_hash_update(&desc, &hash_sg, len);
  if (ret) {
    printk("Error updating crypto hash\n");
    goto out_with_error;
  }
  ret = crypto_hash_final(&desc, out);
  if (ret) {
    printk("Error finalizing crypto hash\n");
    goto out_with_error;
  }
  crypto_free_hash(tfm);
  return 0;
out_with_error:
  crypto_free_hash(tfm);
  return -ret;
}
```

The function `sha256_mac()` generates the MAC by performing SHA256 hash function on concatenation of the provided `key` with the input `in`. The output is

stored in the memory location provided with the reference `out`. The input size is passed as the parameter `len` and the key size is passed as the parameter `keylen`.

Listing A.17: C code of sha256_mac()

```
int sha256_mac(u8 *in, u8 *out, int len, u8 *key, int keylen) {
  struct scatterlist hash_sg, key_sg;
  struct hash_desc sechash_desc;
  int ret;

  sg_init_one(&hash_sg, in, len);
  sg_init_one(&key_sg, key, keylen);

  sechash_desc.tfm = crypto_alloc_hash("sha256", 0,
     CRYPTO_ALG_TYPE_SHASH);
  ret = IS_ERR(sechash_desc.tfm);
  if (ret) {
    printk("Crypto hash allocation for sha256 failed\n");
    return -ret;
  }

  sechash_desc.flags = 0;

  ret = crypto_hash_init(&sechash_desc);
  if (ret) {
    printk("Error initializing crypto hash\n");
    goto out_with_error;
  }

  ret = crypto_hash_update(&sechash_desc, &key_sg, keylen);
  if (ret) {
    printk("Error updating crypto hash\n");
    goto out_with_error;
  }

  ret = crypto_hash_update(&sechash_desc, &hash_sg, len);
```

```
  if (ret) {
    printk("Error updating crypto hash\n");
    goto out_with_error;
  }

  ret = crypto_hash_final(&sechash_desc, out);
  if (ret) {
    printk("Error finalizing crypto hash\n");
    goto out_with_error;
  }

  crypto_free_hash(sechash_desc.tfm);
  return 0;
out_with_error:
  crypto_free_hash(sechash_desc.tfm);
  return -ret;
}
```

The `do_verify_page_mac()` function performs the entire operation of the page verification by first identifying which key must be used (`initrd_key` versus `integ_sec.key`). The function then allocates the required memory to store the path to the folder where the MAC of the pages are stored. Once the complete path is generated the integrity verification key is used as initial value for SHA256 followed by the content of the page. Once the complete path and filename are generated the function will search for the file to check if it can be opened. If the file can be opened it would indicate that the MAC for that page exists and the verification succeeds otherwise the verification fails.

Listing A.18: C code of do_verify_page_mac

```
int do_verify_page_mac(char *filename, struct page *page)
{

  u8 *macofpage, *filenamesha2;
  int err = 0, i, k, j;
```

```
unsigned char *addr, *hashdir, *pmac;
struct file *file;
u8 *key;

if (unlikely(!integrity_password_is_set)) {
  key = initrd_key;
}
else {
  key = integ_sec.key;
}

filenamesha2 = kzalloc(SHA256_DIGEST_SIZE, GFP_KERNEL);
if (!filenamesha2) {
  printk("Cannot allocate memory for hash of the page\n");
  return VM_FAULT_OOM;
}
err = sha256_hash((u8 *) filename, filenamesha2, strlen(filename)
   );
if (err) {
  printk("Generating SHA256 hash of the filename failed!\n");
  return -EPERM;
}

hashdir = kzalloc(73, GFP_KERNEL);
if (!hashdir) {
  return VM_FAULT_OOM;
}
sprintf(hashdir, "/hashes/");
k = 8;
for (i = 0; i < SHA256_DIGEST_SIZE; i++) {
  sprintf(hashdir + k, "%02x", (unsigned char) *(filenamesha2 + i
     ));
  k = k + 2;
}
```

```
hashdir[73] = '\0';

macofpage = kzalloc(SHA256_DIGEST_SIZE, GFP_KERNEL);
if (!macofpage){
  printk("Cannot allocate memory for hash of the page\n");
  return VM_FAULT_OOM;
}
addr = page_address(page);

err = sha256_mac((u8 *) addr, macofpage, PAGE_SIZE, key,
    SHA256_DIGEST_SIZE);
if (err) {
  kfree(macofpage);
  kfree(hashdir);
  return -EACCES;
}
pmac = kzalloc((SHA256_DIGEST_SIZE * 2) + 74, GFP_KERNEL);
sprintf(pmac, "%s/", hashdir);
k = 73;
for (j = 0; j < SHA256_DIGEST_SIZE; j++){
  sprintf(pmac + k, "%02x", (unsigned char) *(macofpage + j));
  k = k + 2;
  }
pmac[(SHA256_DIGEST_SIZE * 2) + 74] = '\0';

file = filp_open(pmac, O_RDONLY, FMODE_READ);
err = IS_ERR(file);
if (err) {
  kfree(hashdir);
  kfree(filenamesha2);
  kfree(macofpage);
  kfree(pmac);
  return -EPERM;
}
```

```
  filp_close(file, NULL);
  kfree(hashdir);
  kfree(filenamesha2);
  kfree(macofpage);
  kfree(pmac);

  return 0;

}
```

The function will free the allocated memory for its internal variables before returning.

The Listing A.19 shows the modifications in `__do_page_fault()` function that adds the page verification.

Listing A.19: C code of modified part of __do_fault() kernel function

```
if (vma->vm_flags & VM_EXEC) {
    if (vma->vm_file != NULL) {
      buf = kzalloc(PATH_MAX, GFP_KERNEL);
      if (!buf)
        return VM_FAULT_OOM;
      filename = dentry_path(vma->vm_file->f_dentry, buf, PATH_MAX)
          ;
      err = do_verify_page_mac(filename, page);
      if (err) {
        printk(KERN_WARNING "Secure hash verification failed for
            process %s, file:%s, page index:%lx\n", current->comm,
            filename, page->index);

        ret = err;
        if (buf)
          kfree(buf);
        goto out_verification_failed;
      }
      ret = 0;
      if (buf)
```

```
        kfree(buf);
    }
  }
```

A.2.5 Compiling Kernel with initrd

To be able to compile the kernel from its source few packages must be installed. The required packages include `linux-kernel-devel`, `fakeroot`, `build-essential`, `kernel-package`, and `libncurese5-dev`. These packages can be installed on Ubuntu system using the `apt-get install` command. Before compiling the kernel first the command `make menuconfig` must be used in the kernel source directory to choose the modules that need to be compiled directly into the kernel image rather than as loadable modules. The required cryptographic modules such as SHA family of hash functions and AES symmetric encryption algorithm with CBC mode of operation are selected. After selecting the required modules the make configuration must be saved. To compile the kernel the following command can be issued.

```
fakeroot make-kpkg --initrd --append-to-version=pg kernel-image kernel-headers
```

This will compile the kernel using the saved configuration and generate the kernel image, an `initrd` image and kernel headers. To distinguish the kernel image that performs the page-by-page verification a postfix is added to the image name. The images are generated as installable packages which are then installed in the system using the following commands.

```
dpkg -i linux-headers-3.8.2pg_3.8.2pg-10.00.Custom_amd64.deb
dpkg -i linux-image-3.8.2pg_3.8.2pg-10.00.Custom_amd64.deb
```

Since the generated image for `initrd` will also perform the page-by-page verification, the MACs of the executable code pages for the files stored in that image must also be generated. First a temporary folder is created to unpack the image, and the `hashes` folder is created.

```
mkdir /tmp/img
cd /tmp/img
gzip -cd /boot/initrd.img-3.8.2pg | cpio -imd --quiet
mkdir hashes
```

The MACs is then generated for all executable code using two passwords, the `initrd` password and the system-wide password. After generating the MACs for the executable code, the original `initrd` image is renamed. The unpacked folder that contains the MACs will be packed to form the new image using the following commands.

```
mv /boot/initrd.img-3.8.2pg{,.old}
find . | cpio --quiet -H newc -o | gzip -9 -n > /boot/initrd.img-3.8.2pg
```

The `/boot/grub/grub.cfg` must be changed to pass the required boot options for the compiled kernel image. The entry for the compiled kernel in the `grub.cfg` is as follows for the `initrd` password of `JustAPassword` and system-wide password of `AnotherPass` as an example.

```
linux /boot/vmlinuz-3.8.2pg root=UUID=d4748e25-3d2f-44f4-9502-2fa2ed02f8cc ro
console=ttyS0 console=tty0
encrypted_ipass=f5f3f2b48cbf8fc633388fbd0999076b68a778927faaced80a625d6c57d79cd8
initrd_key=d87c959055ec60007ac6580063710e00e1a1fcb281ab6be0a1638b0b7b57ecff
ignore_loglevel  nosplash text
```

A.2.6 Message Authentication Code for Executable Code Pages

To perform the required cryptographic transformations for each page of the executable codes the `openssl` library is used. This library can be installed by issuing `apt-get install libssl-dev`. A function written in `C` is used to generate MACs for each page of the executable code. To compile any `C` code that uses any of the `openssl` library functions, the `-lcrypto` option must be added to the `gcc` command

line. To generate the SHA256 message digest for the path and filename of each executable code another function written in `C` is used. The compiled binary is then used in shell scripts to generate the MACs and store the values under appropriate folders under the `/hashes` folder.

The Listing A.20 shows the code of `sechash.c` that accepts three command line options that pass the required parameters to generate the MAC for each page of an executable code. The parameters are the password, the path and filename of the executable code, and the destination folder for the generated MACs.

Listing A.20: Code of sechash.c function

```
#include "string.h"
#include "stdio.h"
#include "stdlib.h"
#include <openssl/sha.h>
#include <openssl/evp.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "stdint.h"


void printhex(unsigned char *c, unsigned long len, int direction) {
int i;
  if (direction)
    for (i = 0; i < len; i++)
      printf("%02x", *(c + i));
  else
    for (i = len - 1; i >= 0; i--)
      printf("%02x", *(c + i));
  return;
}


int main(int argc, char *argv[]) {
```

```
FILE *fin, *fout;
unsigned char buf[128], first_word[8], byte, *page, md[32], *fname,
    key[32], **somekey;
unsigned char *hashfn, c = '\0';
int i, err, j, k;
size_t bytesread;
int dir = 1;
const unsigned char * pass;
unsigned int mdlen = 32;
EVP_MD_CTX *mdctx;

pass = argv[1];
SHA256(pass, strlen(pass), key);

if (argc == 1) {
  printf("No filenmae was specified\n");
  return -1;
}

fin = fopen(argv[2], "rb");
if (fin == NULL) {
  printf("Error opening file %s to read\n", argv[2]);
  goto out_with_error;
  }

page = malloc(4096);
if (page == NULL) {
  printf("Out of memory\n");
  fclose(fin);
  return;
}



fname = malloc(strlen(argv[3]) + 66);
if (fname == NULL) {
```

```
  printf("Out of memory\n");
  free(page);
  goto out_with_error;
  }

if ((mdctx = EVP_MD_CTX_create()) == NULL) {
    printf("Error in creating EVP for message digest\n");
    free(page);
    goto out_with_error;
  }

do {
  bytesread = fread(page, 1, 4096, fin);
  if (bytesread != 4096) {
  // Zeroing remainder of the last page
    for (j = bytesread + 1; j < 4096; j++)
      *(page + j) = 0x0;
    }

  if(1 != EVP_DigestInit_ex(mdctx, EVP_sha256(), NULL)){
    printf("Error initializing EVP\n");
    free(page);
    goto out_with_error;
  }

  if(1 != EVP_DigestUpdate(mdctx, key, 32)) {
    printf("Error updating EVP message digest with password\n");
    free(page);
    goto out_with_error;
  }

  if(1 != EVP_DigestUpdate(mdctx, page, 4096)) {
      printf("Error updating EVP message digest with page content\n
         ");
      free(page);
```

```
        goto out_with_error;
    }

    if(1 != EVP_DigestFinal_ex(mdctx, md, &mdlen)){
       printf("Error finalizing message digest\n");
       free(page);
       goto out_with_error;

     }

    sprintf(fname, "%s/", argv[3]);
    k = strlen(argv[3]) + 1;
    for (j = 0; j < 32; j++){
      sprintf(fname + k, "%02x", (unsigned char) *(md + j));
      k = k + 2;
      }
    fname[strlen(argv[3]) + 65] = '\0';
    fout = fopen(fname, "wb");
    if (fout == NULL) {
      printf("Error opening file %s to write\n", fname);
      fclose(fin);
      free(page);
      free(fname);
      goto out_with_error;
    }

    fwrite(NULL, 0, 0, fout);
    fclose(fout);
    i++;
  } while (bytesread == 4096);

  free(page);
  fclose(fin);
  return 0;
```

```
out_with_error:
return 1;
}
```

A.2.7 Time Measurements of MAC Function for Sample Executable Files

This section reports the measured execution time of generating the MAC for selected executable files.

Table A.1: MAC function time measurements for `/bin/tar` file (μ seconds)

Executable file or library	**MAC function time**
/lib/x86_64-linux-gnu/ld-2.15.so	2754
/lib/x86_64-linux-gnu/librt-2.15.so	608
/lib/x86_64-linux-gnu/libpthread-2.15.so	2483
/lib/x86_64-linux-gnu/libc-2.15.so	16881
/bin/tar	6920
Total time	29646

Table A.2: MAC function time measurements for `/bin/grep` file (μ seconds)

Executable file or library	**MAC function time**
/lib/x86_64-linux-gnu/ld-2.15.so	2754
/lib/x86_64-linux-gnu/libdl-2.15.so	301
/lib/x86_64-linux-gnu/libc-2.15.so	16881
/bin/grep	2919
Total time	29646

Table A.3: MAC function time measurements for `/bin/ls` file (μ seconds)

Executable file or library	**MAC function time**
/lib/x86_64-linux-gnu/ld-2.15.so	2754
/lib/x86_64-linux-gnu/librt-2.15.so	608
/lib/x86_64-linux-gnu/libacl.so.1.1.0	592
/lib/x86_64-linux-gnu/libattr.so.1.1.0	371
/lib/x86_64-linux-gnu/libpthread-2.15.so	2483
/lib/x86_64-linux-gnu/libselinux.so.1	2234
/lib/x86_64-linux-gnu/libc-2.15.so	16881
/bin/ls	2003
Total time	27926