PROGRAM SECURITY EVALUATION USING DYNAMIC DISASSEMBLY OF MACHINE INSTRUCTIONS IN VIRTUALIZED ENVIRONMENTS

E.A. Wanniarachchi

148242T

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

April 2016

PROGRAM SECURITY EVALUATION USING DYNAMIC DISASSEMBLY OF MACHINE INSTRUCTIONS IN VIRTUALIZED ENVIRONMENTS

E.A. Wanniarachchi

148242T

Thesis submitted in partial fulfillment of the requirements for the degree
Master of Science Specialized in Security Engineering

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

April 2016

Declaration

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis/dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works.

....

E.A Wanniarachchi Date

The above candidate has carried out research for the Masters thesis under my supervision.

....

Dr. Chandana Gamage Date

Abstract

Having strong built-in security features has become a paramount requirement in any system. There is a clear difference between bolted vs. built-in security, where in bolted security, the security of the system will depend on the security strength of its bolted parts, where as in built-in security, it is embedded to the system by design. Therefore in order to ensure security, it is required to build security features in to the system by design so that the ultimate security of the system will be ensured by default; ensuring security by design and by default.

The execution of a computer program is not stand alone, but instead is a collaborative execution of several programs. Generally at run time, a given program will call functions from other programs and also transfer its control to other program segments, introducing a change to its control flow. In most cases caller (the main program) is not fully aware about its callee (the called program), in the context of its vulnerabilities and security risks. In addition to that, this control transfer will potentially change the trust boundary of the system, while increasing the attack surface of the program in terms of Control Flow Integrity (CFI). On the contrary, completely eliminating this execution behavior is impractical since it is required to build applications having such a modular design due to various reasons, such as performance. Complexity is treated as the enemy of computer security. The more complex a system gets, harder to make it secure. This principle has been studied in detail in the context of program complexity and its relation with security. This research explicitly addresses the question “what is the risk that a microprocessor undergoes due to the execution of user programs?” This opens up a new dimension in security by imposing the importance of runtime program analysis.

The research introduces RECSRF; a novel framework to quantitatively evaluate the security of an execution in line with the impact it makes over the microprocessor. RECSRF consists of two components; a novel concept called The Runtime Execution Complexity (REC) of a program execution, which evaluates the tradeoff between performance vs. security, while adhering the Control Flow Integrity (CFI) of programs, and an information theoretic technique to approximate the Security Risk Factor (SRF), which approximates the risk of a particular execution by analyzing dynamically disassembled machine instructions. The RECSRF value allows software designers to select the most secure resource combination among given set of resources, and software implementers to decide whether to proceed or not with a software change. The method can also be used to detect control flow hijacks at runtime by using it as an intrusion detection mechanism which allows transforming the same to an intrusion preventer upon successful implementation. The most notable feature of RECSRF is that it can be applied on highly volatile microprocessors such as on microprocessors hosting virtualized environments.

Acknowledgement

I would like to take this opportunity to express my special appreciation and thanks to my supervisor Dr. Chandana Gamage, who has been a tremendous mentor for me throughout this research. Your advice on both research as well as on my academic career have been priceless. I would like to thank you for encouraging my research and for allowing me to grow as a research scientist. I also want to thank you for letting my defense be an enjoyable moment, and for your brilliant comments and suggestions. Thank to you Sir.

I would also like to express my sincere gratitude to all the academic and non-academic staff members at the Department of Computer Science and Engineering, University of Moratuwa for their contribution in making this research a success. My sincere gratitude also goes to Dr. Dasarath Weeratunge from Intel Corporation for his time, patience, motivation, and immense knowledge. Your thoughts have immensely helped in this research.

A special thanks to my family. Words cannot express how grateful I am for all of the sacrifices that you have made on my behalf. At the end I would like express appreciation to my loving wife who spent sleepless nights with and was always my support in the moments when there was no one to answer my queries. Thank you all.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

CC - Cyclomatic Complexity

CFI – Control Flow Integrity

CIP **-** Confidentiality and Integrity Protection

CISC - Complex Instruction Set Computing

CPU – Central Processing Unit

DoS - Denial-of-Service

DMA - Direct Memory Access

HLL - Higher Level Languages

IT – Information Technology

LLVA - Low Level Virtual Architecture

MIPS - Microprocessor without Interlocked Pipeline Stages

RISC – Reduced Instruction Set Computing

ROP – Return Oriented Programming

SDLC – Software Development Life Cycle

SMM –System Management Mode

TCB – Trusted Computing Base

VM – Virtual Machine

VMM – Virtual Machine Monitor

VT – Virtualization Technologies

CHAPTER 1

1. The Analysis of Software Systems and their Indirect Security Impacts

1.1. Introduction

Modern software systems have a nearly an uncontrolled growth in complex requirements that is fueled by the need to interact with multiple other complex systems. While some of the complexity increases due to activities such as bug fixes which are unavoidable, other activities such as expanding and enhancing functionality of a software system becomes a risky endeavor to undertake as responding to the complexity increases requires scarce resources including technical expertise and time.

This chapter provides an introductory overview about the background, the problem and the motivational factors that have paved the way for this research. Starting with an analysis on software complexity and its relation with security, section one shows how adverse impacts are being made on software system as a result of complexity. Interestingly, it has been identified that some well-established software design principles are also contributing indirectly to make systems complex, making things even more complex. The next section of the chapter has detailed the analysis of these problems along with its resulting motivational factors that have driven this research.

The objectives were identified as critical requirements to be addressed due to its highly practical nature, and these have been detailed under research objectives section. At next, research questions were detailed to check whether the research problem and the objectives are in line. The chapter concludes with an analysis of the research outcomes where the research questions were taken in to account, and finally with the chapter conclusion.

1.2. The Art of Complexity and Software Security

The escalation of software complexity can be treated as a natural byproduct of the functional complexity increases that in turn impact the program code base of the software. There are two types of complexities; Essential Complexity and Accidental Complexity [4]. Essential complexity is unavoidable and is required to fulfill the functional requirements, whereas, the accidental complexity on the other hand is the additional complexity introduced due to design issues and lack of complexity management. However, all these will be reflected in terms of code complexity, results in numerous problems.

In software engineering, there are number of methods to evaluate complexity, such as seize measures, control flow based method, information flow based and software science based matrices [5]. Out of different software complexity measures, McCabe's complexity evaluation [2] that comes under control flow based technique is the widely accepted one due to its applicability to the practical nature. The Cyclomatic Complexity [5] is a code complexity measure that is being correlated to the number of linearly-independent paths through a program module. It is calculated by developing a Control Flow Graph [7] of that particular code segment. High complexity results in leaving untested program segments in a given code base, and that will introduce a security risk.

Software complexity is a major concern among organizations, which used to manage numerous technologies and applications within a multi-tier infrastructure. Therefore it is extremely important to manage the interactions between different layers and components in an application. To make this interaction efficient and manageable, software engineering principles states that object orientation and modular design [10] concepts as best practices that can be used to avoid complexity. Object orientation encourages writing programs in a modular way, i.e. by sub dividing the program into smaller functionally grouped sub parts (called modules). A modular system can be treated as a system having a functional partitioning into discrete and scalable modules where the reusability, performance, maintainability and efficiency are ensured.

In contrary, it can be argued that this design principle will negatively be an impact to the overall security of the system because due to this modular execution behavior, a given main program has to jump back and forth between different program segments to invoke different functionalities. With this execution behavior, the program control will be transferred to different segments, where the control is not with the original author of the main program. An application accessing the database using an API provided by the database vendor, where the source code is not available to the application developer is a classic example for this. A vulnerable return from the database API program will potentially compromise the entire application, regardless of the security strength of the application, i.e. the victim. Conventional security evaluation methods such as static evaluation [8], dynamic testing, fuzz testing [9] or that matter conventional complexity matrices cannot be used to capture these risks, i.e. the risk that a procedural program or a function call introduces to the main program. This is because those measures do not take the impact of its linked programs in to account.

Although cyclomatic complexity is useful, relying only with that will probably produce false positives. A module can be complex, but it might have few interactions with outside modules. That will produce a bad number in terms of complexity. Similarly a module can be relatively simple, but highly coupled to many other modules raising its overall complexity, producing a good complexity number. However, the results can be deceptive. Therefore it is utmost important to take the coupling and cohesion [10] nature of the modules also in to account in order to get a true system wide complexity measure. Deriving these characteristics is not straight forward since the author of the main program st will not always be fully aware about all its modular programs, hence there is no way to get an understanding about the overall complexity the program produces. As a conclusion, it can be stated that deriving a complexity measure considering all these practical constrains is truly challenging.

By excavating these practical aspects, it has been observed that it is vital to have a method that is capable in deriving the security of the system considering the overall impact of all the modular programs involved during the execution. To address that need, this research introduces a novel technique to evaluate the security strength of an execution, followed by few security concepts. Disassembled machine instructions generated by a program during its runtime have become the main data stream of this evaluation technique. The method is effective since it encounters the impact of modular programs as well.

1.3. Research Inspiration

Based on the introduction above, it can be stated that the observations mentioned in this section were inspired to carry out this research.

- It has been observed that there is a rapid growth in software industry and application development is playing a major role in it. Due to ever growing client needs, bug fixes and with continuous development process, software programs are getting complex and sometimes grows beyond the expected boundaries.

- Consequently, it has been observed that complex software results in leaving untested parts in a program, introducing a risk to the entire application. In addition to that, immensely complex software will introduce a whole lot of new problems including difficulties in testing, maintaining and increased costing. Most of all complex software will decrease the security of the system due to its complex nature. Additionally the probability in leaving backdoors behind the system will also considerably increase with increased complexity.

- The analysis on complexity measures reviled that there are number of software complexity measures to evaluate complexity on high level code segments. However, it is questionable whether such methods can be fully adopted when practically analyzing code segments. It has also being observed that, even the widely used code complexity matrices will provide false positives due to having inadequate interactions with other relevant program segments. This has become the motivational factor which allowed being in deriving a new security measure considering the practical nature of program execution. This has enlightened to adopt machine instructions as the key source of information in this research.

- Additionally, it has been observed that different operating systems will introduce different risk levels to their systems because; the underlying data structures, algorithms and code bases on different operating systems are different. In addition to that, the architectural designs on different OSs are also different; for an example a UNIX like operating systems has a monolithic kernel whereas Windows has a micro kernel. On top of that if the system is virtualized, then hypervisor characteristics will also come in to play, making the overall complexity of the program massively complex. Unfortunately no proper research was carried out in this area to detect architecture specific security threats that software programs are imposing.

- Interestingly it was observed that some well-established software design principles are also contributing to increase the overall complexity of programs. This is because, due to modular software design characteristics, program control will be transferred frequently to different program segments, where the invoked program is not fully under the control of the original program author. This introduces a risk to the overall system, due to its interaction with unknown program segments. Unfortunately, currently available software complexity matrices are not capable in evaluating this risk. This has become a motivational fact to research ways in evaluating the overall complexity that programs will introduce over the system.

- The analysis on quantitative methods about security strength of programs, have shown very poor results and it has been observed that there are no effective means in quantifying security. This has been observed as a major drawback in security, compared to other disciplines such as performance. This has motivated to research ways in deriving methods to quantitatively measure security of a program.

- Upon analyzing different attacks that software complexity introduces, some common characteristics have been observed, such as control hijacking prior to privilege escalations. These will be in the form of machine instructions, and the state of the microprocessor will also get changed due to these instructions. This state change of the microprocessor increases the attack surface, opening the doors for processor level attacks on the system, which are extremely powerful and vigorous. However it was observed that a very little attention was paid on these attack vectors, which are triggered simply due to the instructions generated by high level programs. This factor motivated to research on the security impacts that user program instructions are imposing on the underlying microprocessor; a challenging but an interesting topic.

- The analysis on future computing platforms where software systems will be hosted, it has been observed that virtualization will be the key selection due to its scalability and cost effectiveness. With rapid growth in this technology, microprocessor vendors have introduced changes to their existing processor architectures and also have introduced changes to their Instruction Set Architecture (ISA) to better support virtualization. With newly introduced processor modes on Intel-VT microprocessor, a rapid state transition is possible in the system, due to guest system instructions. It has been identified the importance in analyzing the potential security risk such a virtualized system undergoes due to the microprocessor state changes resulted in machine instructions. This has motivated to carry out research on security impacts that Intel-VT microprocessors introduces over the system as a result of the complex nature of software programs.

1.4. The Research Problem

Inspired by the factors mentioned above, this research has successfully addressed the problems mentioned in this section. At high level, the problems were listed below,

1. *Lack of risk quantification mechanisms for security:*

   Unlike in other disciplines, there are no effective methods to quantify security. The importance in quantifying security as a risk has been emphasized in this research.

2. *Software complexity and the security impact due to the modular nature of programs:*

   There are no proper means in analyzing or measuring software complexity that will be presented as a result of the collaborative nature of program execution. The importance of this has been emphasized in this research, i.e. the impact of complexity on software security, and how certain software engineering best practices increases the overall program complexity.

3. *The need to extend and redefine some existing security concept to consider dynamic execution nature of programs:*

   It has been observed that some security theories and principles that are there, are not correctly addressing some practical aspects of program executions. It has been emphasized the importance in extending some of these concepts to better support the practical aspects of program execution.

4. *The microprocessor state change due to dynamic execution complexity and threats it introduces to the system:*

   The attacks on microprocessors are extremely vigorous, but currently there are no methods to evaluate this risk. It has been noted that due to the complex nature of programs, the microprocessor will undergo on different risks

5. *The risk analysis on Intel based processors specifically designed to support virtualization:*

   With ever growing popularity in virtualization technologies, microprocessor vendors have introduced modifications to their processor architectures including some special instructions to better support virtualization. It is still unanswered the level of security risk that these newly designed special purpose microprocessors will produce due to the complex nature of its program executions.

1.4.1. Software based Analysis

By following the above mentioned research inspirations and problems, a comprehensive analysis has been performed about research problem. This section has detailed the major areas of interest, along with its applicability.

1.4.1.1. Issues of Software Complexity

The main problem addressing in this research is the impact that code complexity introduces to the overall security of the system. With ongoing development and bug fixes, the complexity of programs will keep on be increasing and this is something that cannot be completely avoided. On top of that, it has been observed that with object oriented and modular design principles, program complexity will be cyclomatically increased, causing an additional threat to the system. This is a practical concern in software development and there should be a mechanism to evaluate programs against a given complexity measure. This is something lagging in security.

1.4.1.2. Quantification of Security Risks

Currently there are no effective measures of quantifying security and this research enforce the importance in this quantification. This is a major drawback in this area. Comparing this with other disciplines in technology such as performance, which has a lot of quantifying methods, security can be treated as if it is in a primitive stage. A measurement is important, because it provides ways of continuously evaluating the state of a program. The ability to quantify something is vital in many ways such as in maintenance, planning and costing.

1.4.1.3. Complexity vs. Modular Program Design

Even though cyclomatic complexity is useful, it has been observed that relying only on that is problematic and it will produce false positives. This is because a given program author will not always be fully aware about its invoked programs, in terms of vulnerabilities and threats it produce. This is risky, i.e. the risk a given program imposes over system due to its integrated nature of execution. This is practical, yet an unaddressed problem.

1.4.2. Hardware based Analysis

1.4.2.1. Complexity Impact on the Microprocessor

The other notable problem is the security impacts that microprocessor state changes imposes over the system. Depending on the machine instructions it receives, the microprocessor will oscillate among different states. These state changes will introduce a risk to the system in terms of control hijacking [14], i.e. by fooling the processor to execute malicious instructions. Due to its low level nature, these attacks are extremely powerful, but will not be detectable by conventional security evaluation techniques. The research has proposed a novel technique to identify this risk at the microprocessor, and an approximation to quantify this security risk.

1.4.2.2. Complexity Introduced by Machine Instructions

Consequently, considering the processor level program execution nature at runtime all these will be about machine instructions. The CPU will execute these instructions sequentially, in spite of checking the authenticity of instructions, i.e. whether a given instruction has come from a legitimate program or from a malicious program segment. The microprocessor doses not perform any checks on the authenticity of instructions that are required to be executed. This loophole at the processor introduces a greater risk to the execution, including malicious modifications and microprocessor level privilege escalations [24]. These are powerful, yet extremely vigorous security concerns at low level machine boundaries.

1.4.2.3. Demand in Virtualization

The other problem that have taken in to account is the security impact that virtualization has introduced. The main reason for the popularity the virtualization has gained over the years is its cost. Apart from the other benefits such as feasibility and extendibility, the cost reduction has captured the attention of many organizations. Unfortunately a very little attention was paid on its security aspects in this whole process. In a virtualized environment, guest systems are running on hardware where the underlying hardware ownership is not under the control of the guest system. Traditional Virtual Machine Monitors [15] (VMMs) were relying on “ring compression [section 3.3.1]” or “de-privileging”. Hardware-assisted virtualization [section 6.9.3] has become the preferred choice because it has revolutionized this technology by allowing the guest operating system to run its own native ring 0. This has vastly improved performance barriers that were there with previous techniques, but we have observed that it has opened up a whole lot of new threats to the system.

These critical concerns and their practical constrains have paved the way to think out of the box and led to come up with the novel technique RECSRF [chapter 7], followed by introducing The Runtime Execution Complexity (REC) [section 7.4.1], The Security Risk Factor (SRF) [section 7.5] and the concept of a Threat Block (TB), to quantitatively approximate the security strength of an execution.

1.5. The Research Design

This section contains the details about the construction of the research.

1.5.1. Research Objectives

Having observed some natural byproducts that software complexity produces, it has been motived to follow a research to detect existing gaps between complexity and security. Additionally, the researches on quantitative measures about security have shown that there are no formal means to quantitatively measure security. As a result, deriving a technique to quantitatively measure security has become one of the major objectives in this research. Apart from that, analyzing machine instructions to detect complex nature of programs has become another objective. The evaluation of security risk on virtualization enabled system and quantitative evaluation of its security risks has become our final objective.

1.5.2. Research Questions

In order to achieve the research objectives, it is required to have a complete, clear and a well-defined set of questions. In parallel to this, below research questions were formulated.

> *What approach should be followed to analyze software complexity of programs?*

The focus of this question is the research approach. An approach has to be supported by a sound theoretical foundation. Therefore, it is vital to design a number of appropriate research sub-questions to guide the research to find a solution to the main research question. Such questions are supplementary to the main research question and helped in arriving at a feasible solution. Few sub-questions were raised they are listed below.

> *How to address tradeoffs between modular program design principles and its adverse impact in line with security?*

In software design, object orientation and modular design concepts are treated as best practices to adhere with, when writing programs. However, observing this from disassembled instruction point of view, this has reflected in terms of program control transfers, which introduces a risk to the program.

> *How to extend the currently available principles in software complexity to take it to a new dimension that incorporates the practical nature of execution?*

Currently available software complexity principles cannot be used effectively in evaluating security risks that programs are introducing. The main reason is that those measures does not take the risk that modular programs are introduces to the system. Due to this reason, it has been identified the requirement in deriving a new method to analyze this risk factor.

> *What is the risk that the microprocessor undergoes due to execution of machine instructions?*

Microprocessor has to change its state depending on the machine instructions it is executing. This state change will introduces a risk to the system. However it is vital to have a method to quantify this risk.

> *What is the risk that a virtualized system undergoes as a result of newly introduced virtualization extensions?*

In order to support virtualization, processors designers have introduced instruction set extensions. In that context, special instructions were introduced to better deal with virtualization needs. The in-built processor level instruction sets are heavily used by the critical applications running on top of such virtualized systems, hence, any security loophole in those instructions will put the entire system in danger.

1.5.3. Research Outcomes

The overall outcome of the research is shaped by research objectives and with related main research questions. According to the research objective and the main research question, the essence of this research will be an approach that can be used by Software Developers, Architects, and Security Professionals or even by Chief Information Officers. The approach present in this research can effectively be adopted in secure program development process.

The first research sub question is concerned about software complexity and approaches that can be used to implement it. This imposes the importance in analyzing different approaches to measure software complexity. This requires adopting an information theoretic method to analyze software complexity, and therefore the approach consists of a sound theoretical base. The next sub question is concerned about the limitations that currently available software complexity matrices are having. This requires performing a comprehensive gap analysis about their capabilities. Apart from that, in order to address certain practical aspects, it is required to extend some existing principles to better address those needs.

The next sub question draws the attention on the potential threats that the underlying microprocessor undergoes as a result of executing instructions. This is an area where not much attention was paid, but the importance behind it in the context of microprocessor based attacks has been emphasized in this research. Finally with the forth sub question, the risk that virtualized systems undergoes as a result of these microprocessor state changes were taken in to account. Due to the changes introduced on microprocessors, it can introduce a risk to the execution environment, which the research is arguing as something that should be measured quantitatively.

1.6. Summary

This chapter has provided a broad introduction to the research and its background. A comprehensive analysis has been performed around the research problem and its practical nature. The relationship between software complexity and security has been identified as an art due to the tightly coupled nature of those two disciplines. The chapter has shown how complex software decreases security, and how that complex nature contributes in increasing the attack surface of the underlying microprocessor.

The analysis of the research problem has shown that it is not limited to a particular area only, but can be applied to several other areas. It has reviled that it is difficult to completely isolate a particular problem and come up with a specific solution; hence what could be done is to go in search of an optimum solution that will successfully address several problems.

The two main problems identified under research problem are lack quantification methods and software complexity impact on systems especially on microprocessors. Research objectives and questions were the driving forces of this research and those have been detailed having research problems in background. Finally a comprehensive outcome analysis was done considering the research objectives and questions.

CHAPTER 2

2. Computer Architecture and Security Mechanisms

2.1. Introduction

This chapter contains the details about computers and their organization. The chapter starts with an analysis on the Turing machine and the Von Newman architecture followed by an analysis on different layers of a computer system. An evaluation about the instruction set architectures was also performed in this chapter along with an introduction to different types of ISAs. The chapter then details the information about the Central Processing Unit and its components related to this research. Finally an analysis was done on X86 architecture and assembly language.

The Part 2 of the chapter focuses on Security Kernels. The security kernel is a critical part of a computer system focuses on the design aspects of hardware, firmware, the kernel, and trusted services that could be verified to implement a specific security policy. Low level protection mechanisms are the mechanisms inside a computer system that by design ensures security of the system. This chapter discusses about security kernels and its underlying low level protection mechanisms focusing the x86-architecture. During the research, the details mentioned under this chapter were studied in detail, in order to get an understanding about the current situation of security.

Part 1: Computer Organization

2.2. Computer Organization and Architecture

Computer organization refers to the operational units and their interconnections that realize the architectural specifications. This at high level includes hardware details that are transparent to the programmer, such as control signals, interfaces between the computer and peripherals; and related memory technologies. Architecture on the other hand refers to the attributes of system visible to programmer or those attributes that have a direct impact on the logical execution of the program, such as architectural attributes include instruction set, the number of bits used to represent the data types, I/O mechanism and techniques of addressing memory.

A computer system consists of a number of layers of components, where each layer is tightly coupled with its both lower and upper layers. These layers were shown in Figure 2.1 given below.

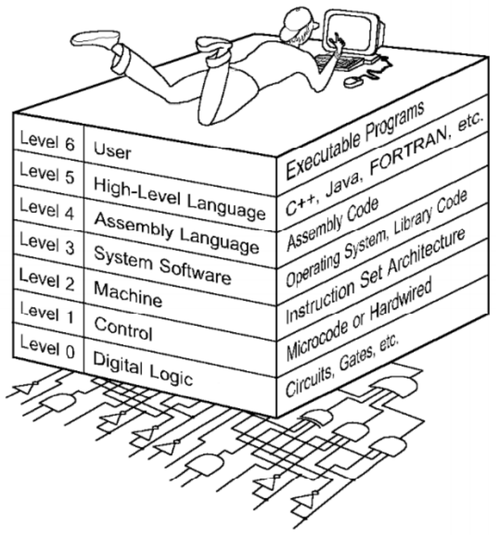


Figure 2.1: Different layers of a computer system

2.2.1. The Turing Machine

A Turing machine [18] invented by Alan Turing in 1937 is a theoretical computing machine, served as an idealized model for mathematical calculation. It can be treated as a state machine, where at any time the machine is in any one of a finite number of states. A Turing machine consists of a line of cells known as "tapes" that can be moved back and forth, an active element known as the "head" that possesses a property known as "state" and that can change the property known as "color" of the active cell underneath it. Instructions for a Turing machine consist in specified conditions under in which the machine will transition between one state and another. For this grate innovation that revolutionized the field of computer science, Turing is treated as the father of modern computer science.

2.2.2. The Von Neumann Architecture

The Von Neumann Architecture [19] was named after the mathematician John von Neumann, forms the core of nearly every computer system in use today. This architecture in contrast to a Turing machine has a random-access memory (RAM), where each successive operation and read or write any memory location, independent of the location accessed by the previous operation. The architecture also has a central processing unit (CPU) with one or more registers that hold data that are being operated on.

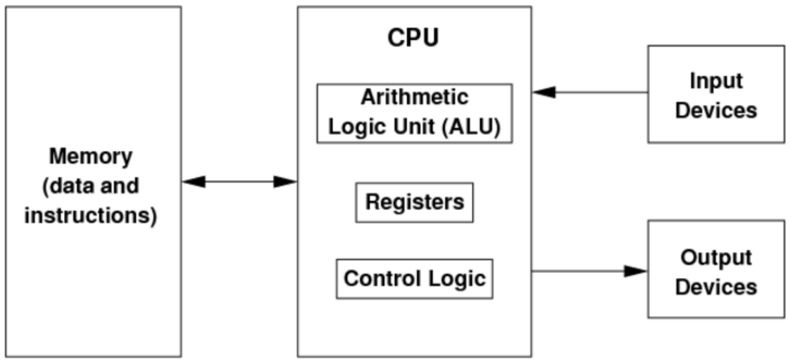


Figure 2.2: The Von Neumann Architecture

The CPU obtains its data from an external memory unit, and writes back the results to the memory unit. The memory unit is also used to hold the program instructions, which control the processing unit and instructs on how to manipulate data. The idea of keeping both data and the instructions in the memory unit is the essence of the stored-program architecture. The CPU has a set of built-in operations (its instruction set) that is far richer than with the Turing machine, e.g. adding two binary integers, or branching to another part of a program if the binary integer in some register is equal to zero (conditional branch). It can also interpret the contents of memory either as instructions or as data according to the fetch-execute cycle. Von Neumann considered parallel computers but recognized the problems of construction and hence settled for a sequential system. For this reason, parallel computers are sometimes referred to as non-von Neumann architecture.

2.2.3. Early Inventions in Microprocessors

A microprocessor is one of the most central parts in a modern personal computer or in any advanced computer device [21]. It integrates the functions of a central processing unit, the portion of a computer responsible for carrying out programmed instructions, onto a single integrated circuit that couples the important thinking devices of the machine with the electrical infrastructure needed to support them. Microprocessor design is able to incorporate a tremendous amount of processing power in a very small space rather than any other component of the modern computer.

Before the development of the microprocessor, there were a variety of early technologies for simulating logic functions in computing devices. Many of these early inventions were spurred by wartime necessity during World War II. These early technologies were extremely expensive, slow, and prone to failure. Computing technologies based on vacuum tubes and transistors helped make IBM a giant in the large-scale computing industry, but were not realistic for business or home use due to their prohibitive costs and intensive maintenance schedules. Early integrated circuits appeared in calculators, of all things, before Intel began work on the first recognizable microprocessor. Another view of evolution of the microprocessor from a more technical perspective, focusing on the different companies and competition involved at various stages of microprocessor design.

2.2.4. Microprocessors Today

Today's microprocessors are immensely powerful, capable of executing complex instructions at a faster rate than ever before. The continued forward march of microprocessor technology depends as much on pure computing research as it does on cutting edge developments in other fields of science. It is indisputable that today's microprocessors are more powerful than anything that could have been imagined at the dawn of the computing age over half a century ago. As increasingly globalized societies demand better computing technology, more great advances are sure to be made.

2.3. An Introduction to Instruction Set Architecture (ISA)

The Instruction Set, also called Instruction Set Architecture (ISA), [21] is a part of the computer that refers to programming, which is basically the machine language. The Instruction Set provides commands to the processor, to tell it what it needs to do. Instructions direct the computer in terms of data manipulation. A typical instruction consists of two parts; Opcode and Operand. Opcode or operational code is the instruction applied. It can be loading data, storing data etc. Oprand is the memory register or data upon which instruction is applied. The Instruction Set consists of multiple pieces including addressing modes, instructions, native data types, registers, memory architecture, interrupt, and exception handling, and external input output.

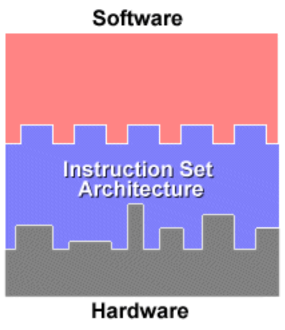


Figure 2.3: Instruction Set Architecture

An example of an instruction set is the x86 instruction set, which is common to find on computers today. Different computer processors can use almost the same instruction set, while still having very different internal design. Both the Intel Pentium [22] and AMD Athlon processors [23] use nearly the same x86 instruction set. An instruction set can be built into the hardware of the processor, or it can be emulated in software, using an interpreter. The hardware design is more efficient and faster for running programs than the emulated software version.

2.3.1. CISC Architecture

Complex Instruction Set Computing (CISC) refers to computers designed with a full set of computer instructions that were intended to provide needed capabilities in the most efficient way [25] The obvious reason for this classification is the “complex” nature of its Instruction Set Architecture (ISA). The motivation for designing such complex instruction sets is to provide an instruction set that closely supports the operations and data structures used by Higher-Level Languages (HLLs).

The decision of CISC processor designers to provide a variety of addressing modes leads to variable-length instructions. For an example, instruction length increases if an operand is in memory as opposed to in a register, because it has to specify the memory address as part of instruction encoding. This complicates instruction decoding and scheduling. The side effect of providing a wide range of instruction types is that the number of clocks required to execute instructions varies widely. This again leads to problems in instruction scheduling and pipelining. Later, it was discovered that, by reducing the full set to only the most frequently used instructions; the computer would get more work done in a shorter amount of time for most applications.

2.3.2. RISC Architecture

Reduced Instruction Set Computer (RISC) is a microprocessor that is designed to perform a smaller number of types of computer instructions so that it can operate at a higher speed [25]. Since each instruction type that a computer must perform requires additional transistors and circuitry, a larger list or set of computer instructions tends to make the microprocessor more complicated and slower in operation. There is no precise definition of what constitutes a RISC design. Since both CISC and RISC have their advantages and disadvantages, modern processors take features from both classes.

2.3.3. MIPS Architecture

Microprocessor without Interlocked Pipeline Stages (MIPS) [26] is a reduced instruction set computer (RISC) instruction set (ISA) developed by MIPS Technologies. There are different extensions to the original MIPS architecture as listed below.

- MIPS I, implemented in the R2000 and R3000
- MIPS II, implemented in the R6000
- MIPS III, implemented in the R4400
- MIPS IV, implemented in the R8000 and R10000

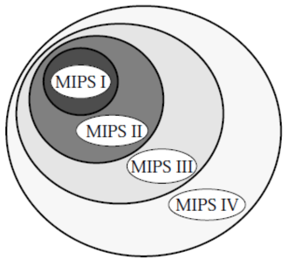


Figure 2.4: MIPS Extensions

2.4. The Central Processing Unit

The Central Processing Unit [27], canonically called The CPU, treated as the central brain of the computer, is the main component which carries out all the logical and arithmetic operations. It is the main component which carries out all the logical and arithmetic operations inside the computer. A CPU contains three main sections; the register section, the arithmetic-logic unit (ALU), and the control unit. These sections work together to perform the sequences of micro-operations needed to perform the fetch, decode, and execute cycles of every instruction in the CPU's instruction set.

2.4.1. CPU Instruction Cycle

Instructions are processed under direction of the control unit step-by-step, called the Fetch-Decode-Execute cycle, also called the Fetch-Execute Cycle. This execution as a function of time is shown below.

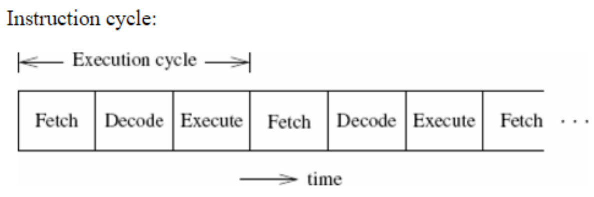


Figure 2.5: The Fetch-Execute Cycle

The Fetch-Decode-Execute cycle is the process by which the CPU;

- Fetches a program instruction from its memory,
- Determines what the instruction wants to do, and
- Carries out those actions.

This cycle is continuously repeated by the CPU, from boot-up till the computer is shut down. In modern computers this means completing the cycle billions of times a second.

2.4.1.1. Fetch

The first step the CPU carries out is to fetch some data and instructions (program) from main memory then store them in its own internal temporary memory areas. These memory areas are called “registers”. This is called the 'fetch' part of the cycle. For this to happen, the CPU makes use of a vital hardware path called the “address bus”. The CPU places the address of the next item to be fetched on to the address bus. Data from these address then moves from main memory into the CPU by travelling along another hardware path called the “data bus”.

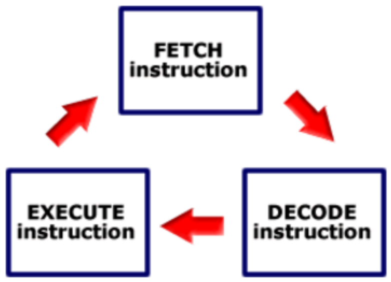


Figure 2.6: Fetch-execute cycle

2.4.1.2. Decode

The next step is for the CPU to make sense of the instruction it has just fetched. This process is called “decode”. The CPU is designed to understand a specific set of commands. These are called the “instruction set” of the CPU. Each make of CPU has a different instruction set. The CPU decodes the instruction and prepares various areas within the chip in readiness of the next step

2.4.1.3. Execute

This is the part of the cycle when data processing actually takes place. The instruction is carried out upon the data (executed). The result of this processing is stored in yet another register. Once the execute stage is complete, the CPU sets itself up to begin another cycle once more

2.4.2. CPU Registers

A register (also called a circuit) is a special, high-speed storage area within the CPU. It is a discrete memory location within the CPU designed to hold temporary data and instructions. All data must be represented in a register before it can be processed. When a register is being used to move data or instructions from one part of the system to another, this is called a buffer. If it is being fetched from the main memory (The RAM), then that would pass through a buffer before being loaded into another internal register.

In the Fetch-Execute cycle, registers are used extensively to move data and instructions around the CPU. Registers are used during the Fetch-Execute cycle are;

- **Program Counter (PC):**

  A program counter contains the address of the instruction being executed at the current time, upon which for each fetched instruction the program counter increases its stored value by 1. After fetching an instruction, the program counter points to the next instruction in the sequence where the program counter normally reverts to 0.

- **Memory Buffer Register (MBR):**

  Memory Buffer Register, also called the Memory Data Register (MDR) temporarily holds data or program instructions once fetched from memory. It contains the data value being fetched or stored. MBR has two inputs and two outputs, in which the data is loaded into MBR either from the memory bus or from the internal processor bus.

- **Memory Address Register (MAR):**

  The memory address register has its output hooked up to the address bus, allowing the communication between the CPU and the address bus.

- **Current Instruction register (CIR):**

  Having been fetched from memory, CIR holds the current instruction to be executed.

- **Control Unit (CU):**

  Control Unit decodes the program instruction in the CIR, selecting machine resources such as a data source register and a particular arithmetic operation, and coordinates activation of those resources

- **Arithmetic logic unit (ALU):**

  ALU performs mathematical and logical operations inside the CPU.

2.5. An Evaluation of Intel x86 Machine Instructions

X86 is the world's predominant personal computer CPU platform, used in Windows and Linux PCs, Macs and Chrome-books. x86 is a generic name for the series of Intel microprocessor families that is based on the Intel 8086 CPU [27]. The 8086 was launched as a fully 16-bit extension of Intel's early 8-bit based microprocessors and also introduced segmentation to overcome the 16-bit addressing barrier of earlier chips. The term x86 derived from the fact that early successors to the 8086. This series has been the provider of computing for personal computers since the 80286 was introduced in 1982. The x86 line was developed by Intel and includes the Core, Xeon, Pentium.

The electronic operation that a processor core can perform is called a machine operation. A processor performs these one at a time, but billions of them in a second. A machine instruction consists of several bytes in memory that tells the processor to perform one machine operation. The processor looks at machine instructions in main memory one after another, and performs one machine operation for each machine instruction. The collection of machine instructions in main memory is called a machine language program or (more commonly) an executable program.

2.5.1. Instruction Types

There are different instruction types and some of them listed below. These instruction types will be used with in the processor upon executing instructions. In general these can be stated as data manipulation mechanisms with in the processor.

- Data movement instructions.
- Dyadic operations.

- Monadic operations.
- Comparisons and conditional branches.
- Procedure-call instructions.
- Loop control.
- Input/output.

Details about different instructions according to their categorization are available in Appendix section.

2.5.2. X86 Instruction Classification and Architecture Analysis

Instruction classification can be listed as below.

- A Complex Instruction Set Computer (CISC) has many specialized instructions, some of which may only be rarely used in practical programs.
- Reduced Instruction Set Computer (RISC) simplifies the processor by only implementing instructions that are frequently used in programs. Unusual operations are implemented as subroutines, where the extra processor execution time is offset by their rare use.
- Minimal Instruction Set Computer and the One Instruction Set Computer are theoretically important types but these are not implemented in commercial processors.
- Very Long Instruction Word (VLIW) is another variation where the processor receives many instructions encoded and retrieved in one instruction word.

2.5.3. The Importance of Control Transfer Instructions

The x86-architecture provides both conditional and unconditional control transfer instructions to direct the flow of execution. Conditional control transfers depend on the results of operations that affect the flag register. Unconditional control transfers are always executed.

2.5.3.1. Conditional Transfer Instructions

The conditional transfer instructions are jumps that may or may not transfer control, depending on the state of the CPU flags when the instruction executes.

2.5.3.2. Unconditional Transfer Instructions

JMP, CALL, RET, INT and IRET instructions transfer control from one code segment location to another [28]. These locations can be within the same code segment (near control transfers) or in different code segments (far control transfers). The variants of these instructions that transfer control to other segments are discussed in a later section of this chapter. If the model of memory organization used in a particular 80386 application does not make segments visible to applications programmers, intersegment control transfers will not be used.

- **Jump Instruction**

JMP (Jump) unconditionally transfers control to the target location. JMP is a one-way transfer of execution; it does not save a return address on the stack. The JMP instruction always performs the same basic function of transferring control from the current location to a new location. Its implementation varies depending on whether the address is specified directly within the instruction or indirectly through a register or memory.

- **Call Instruction**

CALL (Call Procedure) activates an out-of-line procedure, saving on the stack the address of the instruction following the CALL for later use by a RET (Return) instruction. CALL places the current value of EIP on the stack. The RET instruction in the called procedure uses this address to transfer control back to the calling program.

- **Return From Procedure Instruction**

RET (Return from Procedure) terminates the execution of a procedure and transfers control through a back-link on the stack to the program that originally invoked the procedure. RET restores the value of EIP that was saved on the stack by the previous CALL instruction

- **Return from Interrupt Instruction**

IRET (Return from Interrupt) returns control to an interrupted procedure. IRET differs from RET in that it also pops the flags from the stack into the flags register. The flags are stored on the stack by the interrupt mechanism.

2.5.4. Disassembled Code

In programming terminology, to disassemble is to convert a program in its executable (ready-to-run) form into a representation in some form of assembler language so that it is readable by a human. In essence, a disassembler is the exact opposite of an assembler. Where an assembler converts code written in an assembly language into binary machine code, a disassembler reverses the process and attempts to recreate the assembly code from the binary machine code.

2.5.5. Conditional Execution in Assembly Language

Assembly language is a low-level programming language used to interface with computer hardware. X86 machine instruction can be classified in to different groups based on their function such as floating point, arithmetic, control transfer instructions etc.

X86 assembly provides both conditional and unconditional control transfer instructions to direct the flow of execution. Conditional control transfers depend on the results of operations that affect the flag register. Unconditional control transfers are always executed. JMP, CALL, RET, INT and IRET instructions transfer control from one code segment location to another. The JMP instruction always performs the same basic function of transferring control from the current location to a new location. CALL (Call Procedure) activates an out-of-line procedure, saving on the stack the address of the instruction following the CALL for later use by a RET (Return) instruction. CALL places the current value of EIP on the stack. The RET instruction in the called procedure uses this address to transfer control back to the calling program. T (Return from Procedure) terminates the execution of a procedure and transfers control through a back-link on the stack to the program that originally invoked the procedure. RET restores the value of EIP that was saved on the stack by the previous CALL instruction.

Almost all programming languages have the ability to change the order in which statements are evaluated, and assembly is no exception. The instruction pointer

(EIP/RIP) register contains the address of the next instruction to be executed. To change the flow of control, the programmer must be able to modify the value of EIP. This is where control flow functions come in. We cannot directly access or change the instruction pointer because this register will be changed only by programs. However, instructions that control program flow, such as calls, jumps, loops, and interrupts, automatically change the instruction pointer. Programs change Instruction Pointer (or the Program Counter) by:

1) Unconditional Jumps
2) Conditional Jumps
3) Procedure calls and
4) Return instructions.

2.5.6. The Importance of Security Analysis in Assembly

Information theory is concerned with the amount of information contained in a message, or a string of symbols. Its basic premise is that the occurrence of a symbol with low probability provides more information (or uncertainty or surprise) than the occurrence of a symbol with high probability. There are several information theory software complexity metrics based on the frequency of program elements (Eg Harrison and Leighton) [5]. The probability of an operator is the number of the times the particular operator occurs in relation to the total occurrences of operators. The information theoretic complexity measure that Berlinger has developed [5] is based on the frequency of program tokens in the program. His complexity measure M is defined by

$$\mathbf{M = - \Sigma\, pi*log2\, pi}$$

Where fi is the frequency of the ith token and pi is the probability of the ith token. M is sensitive to the frequency and probability of tokens. M will have a low value if many high probability (e.g. familiar) tokens are used. Cook came up with an approach to mitigate practical limitations of Berlinger’s method. He came up with an approach to group instructions into classes and to consider the classes as tokens, and finally defined below information based complexity measure (M’).

$$\mathbf{M' = M/(Number\ of\ instructions)}$$

2.5.7. Privilege levels in Assembly

Operating systems have protection mechanism called privileged rings. A "Ring", in Assembly Language, represents the level of protection and control the program has over the system. Ring 0 programs have absolute control over everything within a system, while ring 3 has less control. The smaller the ring number, the more control (and less level of protection), the software has. A Ring is more than a concept; it is built into the processor architecture.

Part 2: Security Kernels

2.6. The Operating System

An Operating System is low-level software that enables a user and application software to interact with computer hardware, data and other programs stored on the computer. It manages the hardware and software resources of the system; something utmost important, as various programs and input methods compete for the attention of the central processing unit (CPU) and demand memory, storage and input/output (I/O) bandwidth for their own purposes. It also performs activities, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices. In this context, the operating system can be treated as if it is playing the role of a good parent, making sure that each application gets the necessary resources while playing nicely with all the other applications, while catering the limited capacity of the system to the greatest good of all the users and applications.

Another notable feature of an operating system is that, it provides a stable, consistent way for applications to deal without creating a need for it to interact with the underlying hardware system. This is especially important if there is to be more than one of a particular type of computer using the operating system, such as virtual machines or if the hardware making up the computer is ever open to change.

2.6.1. The Kernel

The kernel is a piece of software that constitutes the central core of a computer operating system, which has complete control over everything that occurs in the system [29]. Direct hardware access is cumbersome task due to its complex nature and therefore, kernels are in use to provide a set of hardware abstractions. These abstractions hide the above mentioned complexity and provide a uniform interface to the underlying hardware, making application development easy.

It is interesting to note that though kernels are useful, they are not mandatory. We can easily load and execute programs at specific addresses without any kernel involvement. In fact, this is how the early computer systems worked. However, it was eventually realized the drawbacks of this approach, and the importance of having a convenient and an efficient way to deal with hardware accessing methods. These modular programs were gradually evolved to be operating system kernels. Security Kernel is essentially the hardware firmware and software elements of a TCB [section 3.4.2] that implement the reference monitor concept [section 3.4.1]. It must mediate all accesses, be protected from modification, and be verifiable as correct.

2.6.2. Kernel Space vs. User Space

The kernel vs. user space concept comes in line with computer memory. The memory of a computer system can be divided in to two distinct areas, called the user space and the kernel space, in which the user space consists of a set of locations where normal user processes are running, whereas that of the kernel space is in which the code of the kernel is stored, and executes under.

2.6.2.1.The Kernel Mode

As mentioned above, in Kernel mode [30], the code under execution has complete and unrestricted access to the underlying hardware. It can execute any CPU instruction and reference any memory address. Kernel mode is generally reserved for most trusted functions of the operating system and crashes in kernel mode are catastrophic

2.6.2.2. The User Mode

In User mode [30], the code under execution has no direct access to hardware or reference memory. Code running in user mode must delegate to system APIs to access hardware or memory. Due to the protection afforded by this isolated nature, in most situations crashes in user mode are recoverable.

2.6.3. Different Operating system Architectures

2.6.3.1.Monolithic kernel

A monolithic kernel [31] is an operating system software framework that holds all privileges to access input-output devices, memory, hardware interrupts and the CPU stack. All kernel services exist and execute in the kernel address space and it can invoke functions directly. Examples of monolithic kernel based OSs: UNIX, Linux.

Monolithic kernels are relatively larger than other kernels because they incorporate with so many aspects of processing at the lowest level, and as a result monolithic kernels have to incorporate code that interfaces with many devices, I/O and interrupt channels, and other hardware operators. A monolithic kernel is a kernel where services like file systems, VFS, device drivers as well as core functionalities including scheduling and memory allocation.

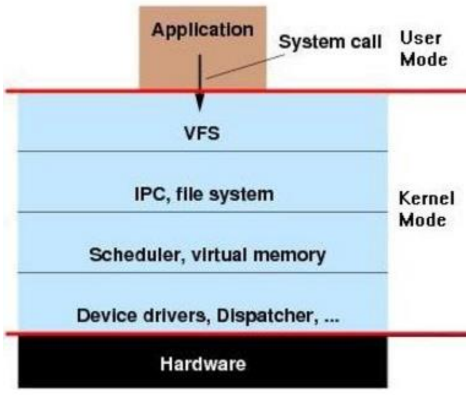


Figure 2.7: Monolithic kernel

One consequence of all parts of the kernel running in the same address space is that if there is an error somewhere in the kernel it will be an impact to the entire address space, for an example, a bug in the subsystem that takes care of networking might crash the kernel as a whole, resulting a reboot in the system.

2.6.3.2. Microkernels

In microkernels [31], the kernel is broken down into separate processes, known as servers. Some of the servers are running in kernel space and some are running in user-space. All servers are kept separately running in different address spaces which invoke "services" from each other by sending messages via IPC (Inter-process Communication). This separation has the advantage that if one server fails, all the other servers still can work efficiently. Examples of microkernel based OSs: Mac OS X and Windows NT. Essentially micro kernels are designed to fix the problem of limiting the damage a bug can cause. This has been achieved by taking the parts of the kernel away from the dangerous kernel space into user space.

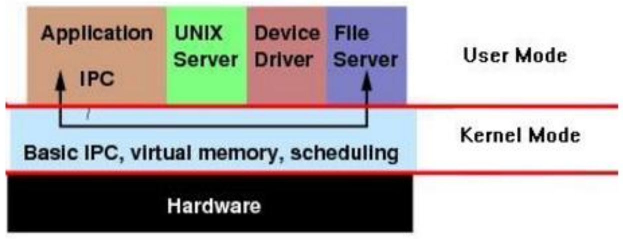


Figure 2.8: Microkernel

The bug in the networking subsystem which crashed monolithic kernel as mentioned in the above example will have far less severe results in a microkernel design. In that context, the networking sub system will crash, leaving the other sub systems to function properly. Many microkernel operating systems have an in-built sub system, whose functionality is to reload crashed servers. Though this design seems to be elegant, it has few downsides compared to monolithic kernels, i.e. added complexity and performance penalties.

In a microkernel design, the kernel space consists of a small subset of the tasks compared to that of a monolithic kernel. The part residing in kernel space i.e. the actual microkernel takes care of the communication between the servers running in user space; called inter-process communication (IPC). These servers provide functionality such as sound, display, disk access, networking, and so on. In a monolithic design, this IPC communication is not needed as all the servers are tied into one big piece of computer code, instead of several different pieces. The result is that a monolithic kernel will generally out perform a micro kernel, given that they are similar feature-wise.

2.7. Processor level Hardware Security Features in x86 Architecture

Operating system designers and hardware designers tend to put a lot of thought into how the kernel can be protected from user-space processes. The security of the system as a whole depends on that protection. But there can also be value in protecting user space from the kernel.

Among several different low-level protection mechanisms inside a computer system, **CPU ring** concept can be treated as the most sort-after feature. It controls which CPU instructions are allowed to be executed. The second and third protection mechanisms are related to memory access, called "Paging" and "Segmentation" respectively. They control which areas of memory are allowed to be accessed and how those areas of memory are allowed to be accessed.

2.7.1. CPU Ring Concept

Privilege levels, is the mechanism whereby the OS and CPU conspire to restrict what different programs can do in a computer system [27]. There are four different Ring Levels, with the most outer ring being the least privileged and the inner most ring being the most privileged. Under this context, three main resources being protected: memory, I/O ports, and the ability to execute certain machine instructions. At any given time an x86 CPU is running in a specific privilege level, which determines what the programs can and cannot do. These privilege levels are described as protection rings, where the innermost ring is corresponding to the highest privilege.

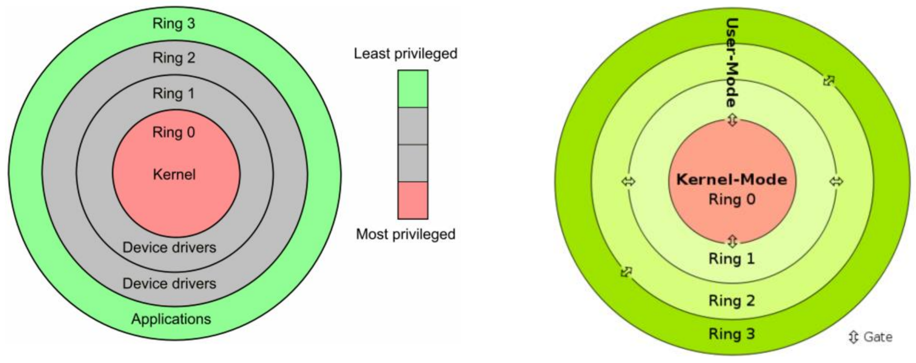


Figure 2.9: Privilege Rings

Details about different CPU rings and their details are given below.

2.7.1.1. Ring 0

Ring 0 is the kernel mode. This is also known as the supervisor mode, which has the least protection and the most access to resources. The operating system upon booting up, and interrupt handlers runs in this privilege level.

2.7.1.2. Rings 1 and 2

Ring 1 and 2 are most of the times used for device drivers. These rings offer more protection, but not as much as the protection provides at ring 3.

2.7.1.3. Ring 3

This is the ring that most operating systems are using for its applications. Ring 3 offers the most protection, but least resource access.

In most cases operating systems are using ring 0 and 3 only, due to the fact that device drivers can run in either ring. In some situations, application requires to access system resources, which cannot be performed from the ring level it is running. A General Protection Fault interrupt will occur in such a situation upon trying to execute unauthorized instructions. At these situations, the application must somehow interact with the kernel; the component providing the hardware abstraction. This is facilitated by System Calls.

2.7.1.4. Current Privilege Level (CPL)

CPL [1] indicates the current mode of the processor. In general CPL=0 is considered to be privileged mode while CPL=Non Zero is considered to be unprivileged mode. Certain instructions are privileged and they can be executed only when CPL=0. These instructions are said to be CPL sensitive. Set of x86 Privileged instructions are listed below. These instruction can be executed if and only if the current privilege level of the processor is equals to zero, i.e. when CPL=0.

- LGDT - Load Global Descriptor Table
- LLDT - Load Local Descriptor Table
- LTR - Load Task Register
- LIDT - Load Interrupt Descriptor Table Register

- MOV (to and from control registers only) - Control Register is used to change the CPU behavior, related to interrupt control, addressing mode, paging and coprocessor control.
- MOV (to and from debug registers only) - Debugging programs, and use to set debugging controls.
- LMSW - Load Machine Status Word
- CLTS - Clear Task Switched
- INVD - Invalidate Cache
- WBINVD - Write Back and Invalidate Cache
- INVLPG - Invalidate TLB Entry
- HLT - Halt; halts the CPU and causing it to wait for the next interrupt.
- RDMSR - Read From Model Specific Register
- WRMSR - Write to Model Specific Register
- RDPMC - Read Performance Monitoring Counters
- RDTSC - Read Time Stamp Counter

2.7.1.5. I/O Privilege Level (IOPL)

IOPL [32] is an import feature in x86-architecture that determines which rings have unrestricted access to I/O ports. It is a two bit number set in the EFLAGS register. Ring 0 has full I/O permissions while those greater than it have none. FLAGS register contains the current state of the processor. It is a 16 bit register. Bits 12 and 13 of FLAGS register contain the value of IOPL (I/O Privilege Level). IOPL value indicates the minimum CPL value required to execute certain privileged I/O instructions. These instructions are said to be IOPL sensitive and mainly include: IN, INS, OUT, OUTS, CLI and STI instructions.

2.7.2. Page Level Protection

2.7.2.1. Paging

Paging [33] is playing the role of a memory optimizer, where it optimizes the use of RAM (The Random Access Memory) by dumping unused memory pages in to the virtual memory of the hard disk. Paging essentially plays a role in memory management for computer’s operating system.

2.7.2.2. Segmentation

Segmentation [33] is involved with loading programs into memory. Being a low-level protection mechanism inside the operating system segmentation is used as memory protection mechanism. A segment has a set of permissions, where the currently running process is allowed by the permissions to make the type of reference to memory that it is attempting to make, and the offset within the segment is within the range specified by the length of the segment, the reference is permitted; otherwise, a hardware exception is raised.

2.7.3. Supervisory Protection Mechanisms

2.7.3.1. Supervisor Mode Execution Protection (SMEP)

With a new generation of Intel processors based on the Ivy Bridge architecture [34] a new security feature has been introduced called Supervisor Mode Execution Protection (SMPE). SMEP is a powerful security feature, which can combat against a considerable percentage of exploits in terms of kernel level privilege escalations. It prevents execution of a code located on a user-mode page at a CPL = 0. This feature significantly complicates an exploitation of kernel-mode vulnerabilities because there's just no place for a shellcode [35] to be stored, hence will make things complicate for an attacker. Usually while exploiting some kernel-mode vulnerability an attacker would allocate a special user-mode buffer with a shellcode and then trigger vulnerability gaining control of the execution flow and overriding it to execute prepared buffer contents. With this innovation, attackers find it difficult to execute the malicious shellcode.

2.7.3.2. Supervisor Mode Access Protection (SMAP)

Supervisor Mode Access Prevention (SMAP) is a new security feature disclosed by Intel in revision 014 of the Intel Architecture Instruction Set Extensions [38]. This extension defines a new SMAP bit in the CR4 control register; when that bit is set, any attempt to access user-space memory while running in a privileged mode will lead to a page fault. Essentially, when SMAP is active, the kernel cannot normally access pages that are in user space. Since the kernel does have the need to access user space pages under specific circumstances, an override is provided, where the kernel can access user space pages if EFLAGS.AC=1.

2.7.3.3. Write Protection (WP)

Write protection (WP) is a very old feature [38], controlled by CR0.WP bit. When this bit is set, it inhibits supervisor mode code from writing into read-only pages, and when it is cleared, it allows supervisor mode code write into read-only pages, regardless of the U/S bit setting. These flag are often used to protect the kernel mode code sections, since those code sections will be configured as read-only pages. A hardware CPU exception will be triggered whenever it detects a malicious modification on kernel code pages, which are mostly triggered by kernel rootkits. Write protection feature is also used to protect kernel static data sections, which are not being modified at runtime.

2.8. Layered Design Principle in x86 Architecture

The x86-architecture is based on a layered design, according to which the execution space delivered by processor is divided into four protected security domains, each of which have its own level of privileges assigned. The goal of this design principle is to ensure that the code under execution will be executed with least possible privilege; hence it ensures the principle of least privilege. Apart from that, the layered protection design states that the control cannot be passed arbitrary among different security domains. To facilitate the control transfer between less privilege code segments to high privilege code segments, a feature called “Gate” has been introduced by the x86-architecute.

The Gate Concept

A gate is a special memory address that facilitates the connection between low-privilege segments to high-privilege ones. When a low-privilege program calls a gate, it automatically raises its CPL to the higher level, whereas upon returning from a gate subroutine, CPL automatically dropped to original level. A gate essentially is used to transfer control of execution across different segments. Privilege level checking is done differently depending on the type of destination and instruction used. There are three types of gates in the x86-architecture.

- Call gate

The purpose of a call gate is to allow less privileged code to call code with a higher privilege level. It transfer control from lower privilege code to higher privilege code, and it uses the CALL and JMP instructions. A call gate plays an important role in memory protection by allowing the user applications to use system calls as well as kernel functions in a way that can be controlled by the operating system.

- Trap gate

The trap gate is called by the INT instruction, which can be stored only in the interrupt descriptor table (IDT). This gate just passes the control to the particular address specified in the trap gate descriptor in the more privileged segment.

- Interrupt gate

Similar to the trap gate, the interrupt gate is called by the INT instruction, but additionally it prohibits future interrupt acceptance by automatically clearing of the IF flag in the EFLAGS register.

2.9. Summary

The Part 1 of this chapter provides details about the importance of security kernels and processor level security mechanisms in place by design. It was identified that even though there exists a number of processor level security mechanisms, those mechanisms were implemented on different microprocessors. This is a drawback due the non-existence of a central microprocessor with all the important security features implemented. The section essentially provides an introduction to the hardware level security mechanisms that were embedded in microprocessors. However the chapter has shown that the mentioned security mechanisms are not highly available features in general.

The Part 2 of the chapter has three major areas; the computer organization and its evolution, the central processing unit and its execution cycle and the importance of machine instructions. The chapter provides theoretical facts behind these sections that are used in this research. The instruction cycle of the CPU has been detailed in this research since it plays as a major contributor in the in deriving the novel framework presented in this research. Additionally an analysis was performed about x86 architecture and its components related to the research including a study on control transfer instructions, and the characteristics about unconditional control transfer instructions were greatly helped in deriving the concept of a threat block (section 5.4).

CHAPTER 3

3. Information Theoretic Concepts in Security

3.1. Introduction

This chapter details the theoretical background behind software complexity and its relation with information theory. Information theory is the main building block in complexity and entropy. Starting with an overview about information theory, the chapter provides details about information theoretical quantification methods and its relation with security. At next the Shannon entropy has been discussed in detail along with its applications. This is a very important section in this research since information quantification was done using the method invented by Shannon. In addition to that different software complexity measures were also discussed in the chapter, highlighting the McCabe's cyclomatic complexity method.

The part 2 of the chapter provides an overview about secure system design concepts, security models, and the importance in maintaining control flow integrity in programs in the context of security attacks. It can be stated that the concepts detailed under Part 2 were greatly helped in deriving the novel concept proposed in this research; RECSRF. It starts with core security concepts that any system should adhere at their design stage, which will ultimately be the security backbone of any system. This enforces the importance in building security in to the system, rather than bolting it in. The next section starts with an overview of different security models that can be applied depending on the application in use. Next, it focuses on control flow graphs and control flow integrity along with its role with respect to modern attack vectors. The chapter concludes with an overview on software attacks and the importance in maintaining control flow integrity of programs.

Part 1: Information Theory

3.2. Information Theory Concepts

Information theory [3] is concerned with the amount of information contained in a message, or a string of symbols. Its basic premise is that the occurrence of a symbol with low probability provides more information than the occurrence of a symbol with high probability. Information theory is a concept that has revolutionized the field of computer science, by its innovative method that is capable in quantifying information. Since its inception the revolutionized paper by Shannon entitled "A Mathematical Theory of Communication". The concept has been broadly applied in many areas such as in cryptography, natural language processing, neurobiology and statistics etc.

Information theory is based on probability theory and statistics, which is often, concerns itself with measures of information of the distributions associated with random variables and studies the transmission, processing, utilization, and extraction of information. The most notable quantities of information are entropy, which measures the information in a single random variable, and mutual information, a measure of information in common between two random variables.

3.2.1. Information sources

An information source is a mathematical model for a physical entity that produces a succession of symbols in a random manner. The symbols produced may be real numbers such as binary numbers as in computer data, voltage measurements from a transducer, continuous or discontinuous waveforms, and so on. The space containing all of the possible output symbols is called the alphabet of the source and a source is essentially an assignment of a probability measure to events consisting of sets of sequences of symbols from the alphabet.

3.2.2. Quantification of Information

There are few information quantification methods [5] and those will measure information with several quantities of information, in which the selection of the logarithmic base will determine the unit of information.

3.2.2.1. Self-Information

Self-information [39] is a measure of the information content associated with an event in a probability space or with the value of a discrete random variable. It attempts to describe the amount of information gained from a specific outcome of an experiment. Intuitively, if an experimental outcome gives a surprising result, then the information gain is said to be high and vice versa. The idea behind this concept is that, a surprising result changes understanding of the world whereas an expected result supports the current understanding. The amount of self-information contained in a probabilistic event depends only on the probability of that event, i.e. smaller the probability, larger the self-information associated with receiving the information that the event indeed occurred.

Self-information is a function of I, where;

$$I : \mathbb{R} \to [0, \infty)$$

defined as,

$$I\,[P(X = x)] = \log \frac{1}{P(X = x)}$$

$$= -\log P(X = x) \tag{3.1}$$

Considering these properties, the self-information I(X) associated with outcome X with probability P(X) is;

$$I(X) = \log (1/ P(X)) = -\log (P(X)) \tag{3.2}$$

3.2.2.2. Entropy

Entropy (or uncertainty) and its complement, information, are perhaps the most fundamental quantitative measures in cybernetics, extending the more qualitative concepts of variety and constraint to the probabilistic domain.

3.2.2.3. Joint Entropy

The joint entropy H(X, Y) of a pair of discrete random variables with a joint distribution p(x, y) is defined as

$$H(X,Y) = -\sum_{x\in\mathscr{X}}\sum_{y\in\mathscr{Y}} p(x,y)\log p(x,y). \tag{3.3}$$

3.2.2.4. Conditional Entropy

The conditional entropy H(Y |X) is defined as,

$$H(Y|X) = \sum_{x\in\mathscr{X}} p(x)H(Y|X=x). \tag{3.4}$$

3.2.2.5. Shannon Entropy

A key measure in information theory is entropy which quantifies the amount of uncertainty involved in the value of a random variable. The idea of entropy by Claude Shannon provided the beginnings of information theory and related measures. The Shannon entropy equation provides a way to estimate the average minimum number of bits needed to encode a string of symbols, based on the frequency of the symbols.

The entropy of a discrete random variable X is defined by,

$$H(X) = -\sum_{x\in\mathscr{X}} p(x)\log p(x). \tag{3.4}$$

Entropy is always positive, i.e. $H(X) \geq 0$ since $0 \leq p(x) \leq 1$ for all p(x).

3.2.3. Entropy vs. Security

Entropy was introduced by Shannon [5] as a quantitative measurement of the uncertainty associated with random phenomena. It is said that one phenomenon represents less uncertainty than a second one if we are more confident about the result of experimentation associated with the first phenomenon than we are about the result of experimentation associated with the second one.

3.3. Software Complexity

3.3.1. An Overview of Software Complexity

Cyberspace is becoming less secure even as security technologies improve. There are many reasons for this seemingly paradoxical phenomenon, but they can all be traced back to the problem of complexity. It has been found [2] that the complexity of a program is inversely proportional to the security of that program. With too many

"moving parts" or interfaces between programs and other systems, the system or interfaces become difficult to secure while still permitting them to operate as intended.
The reasons are complex and can get very technical. A flavor of the rationale can be listed as below. Complex systems,

- Have more lines of code and therefore more security bugs.
- Have more interactions and therefore more security bugs.
- Harder to test and therefore are more likely to have untested portions.
- Harder to design securely, implement securely, configure securely and use securely.
- Harder to understand.

There are different types of software complexity metrics [4] such as

- Size measures (Eg: Lines of code (LOC), Function counts)
- Data structure matrices
- Control flow metrics (Eg: McCabe cyclomatic complexity)
- Knot count
- Information flow metrics
- Software science metrics

There are different methods to evaluate software complexity of programs.

3.3.2. Cyclomatic Complexity

Cyclomatic complexity [5] is defined as measuring the amount of decision logic in a source code function. It is a source code complexity measurement that is being correlated to a number of coding errors. There are graph theoretic complexity measures which illustrate how those can be used to manage and control program complexity. It explains how graph-theory concepts apply and gives an intuitive explanation of the graph concepts in programming terms.
Cyclomatic complexity is calculated by developing a Control Flow Graph [25] of the code that measures the number of linearly-independent paths through a program module. Lower the Cyclomatic Complexity in a program, lower the risk to modify and easier to understand. The Cyclomatic Complexity of a structured program can be defined with reference to the control flow graph of the program, which is a directed graph containing

the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second. Mathematically the Cyclomatic complexity can be expressed in below formulas.

$$\mathbf{M = E - N + 2P,}$$

Where,

E = the number of edges of the graph.

N = the number of nodes of the graph.

P = the number of connected components.

This may be seen as calculating the number of linearly independent cycles that exist in the graph, i.e. those cycles that do not contain other cycles within themselves, because each exit point loops back to the entry point, there is at least one such cycle for each exit point. For a single program (or subroutine or method), P is always equal to 1. So a simpler formula for a single subroutine is,

$$\mathrm{M} = \begin{cases} \mathrm{E} - \mathrm{N} + \mathrm{P}\,; & \text{for a subroutine} \\ \mathrm{E} - \mathrm{N} + 2\mathrm{P}\,; & \text{otherwise} \end{cases} \qquad (3.5)$$

3.3.3. McCabe Cyclomatic Complexity Number

McCabe [2] showed that the Cyclomatic Complexity (CC) of any structured program with only one entrance point and one exit point is equal to the number of decision points. McCabe's Cyclomatic Complexity is a software quality metric that quantifies the complexity of a software program. Complexity is inferred by measuring the number of linearly independent paths through the program. The higher the CC number the more complex the code. Studies show a correlation between a program's Cyclomatic Complexity and its maintainability and testability, implying that with files of higher complexity there is a higher probability of errors when fixing, enhancing, or refactoring. It is accepted that the programs with high McCabe numbers (e.g. > 10) are likely to be difficult to understand and therefore have a higher probability of containing defects.

The selected threshold is based on categories established by the Software Engineering Institute (SEI). This is shown in Table 3.1 below.

Table 3.1 SEI recommendation for cyclomatic complexity

Cyclomatic Complexity	Risk Evaluation
1-10	A simple module without much risk
11-20	A more complex module with moderate risk
21-50	A complex module of high risk
51 and greater	An untestable program of very high risk

Part 2: The Importance of Secure System Design and Control Flow Integrity

3.4. Secure System Design Principles

Security Concepts

In most situations, computer security principles are designed around three well known objectives;

- Confidentiality; which ensures that the system resources on a system can be accessed by authorized parties.
- Integrity; which ensures that the assets can only be modified or deleted only by authorized parties in authorized ways.
- Availability; which ensures that the assets are accessible to the authorized parties in a timely manner.

3.4.1. The Reference Monitor

A reference monitor [40] is a separable module that enforces access control decisions, where all sensitive operations are routed through the reference monitor. The concept is essentially an access control concept that refers to an abstract machine that mediates all accesses to objects by subjects. As an abstraction, the reference monitor concept does not refer to any particular policy to be enforced by a system, nor does it address any particular implementation. However the concept does not judge whether a policy is appropriate.

The concept states that a computer system can be depicted in terms of subjects, objects, an authorization database, an audit trail, and a reference monitor. The reference monitor is the main control center whose function is to authenticate subjects and to implement security policies for each and every access that a subject makes over an object. The reference monitor enforces the security policy by authorizing the creation of subjects, by granting subjects access to objects based on the information in the authorization database, and by recording events, as necessary in the audit trail. The concept can be shown in Figure 5.1 below.

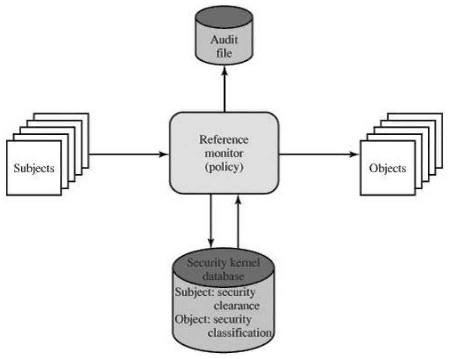

Audit file
Subjects
Reference monitor (policy)
Objects
Security kernel database
Subject: security clearance
Object: security classification

Figure 5.1: The Reference Monitor Concept

Subjects are the active entities that gain access to information such are user processes or services, whereas objects are the passive repositories of information to be protected, such as files. The security kernel database (the authorization database) is the repository for the security attributes of subjects and objects. This is from where the reference monitor determines the kind of access to be authorized.   Audit trial keeps a record of all security relevant events and access attempts, regardless the attempts were successful or not.

3.4.2. Trusted Computing Base (TCB)

A trusted computing base (TCB) [41] refers to hardware, software and firmware components in a system that work collaboratively to provide a system wide secure environment. It enforces security policies to ensure security of the system and its information, whereby system safety is achieved by provisioning methods, like controlling access, requiring authorization to access specific resources, enforcing user authentication, safeguarding anti-malware and backing up data. Thus, if any component in the TCB is compromised, then so is the system's security. By having the TCB be small, it is less likely to contain vulnerabilities, because it will be easier to understand, test, and analyze. Size and complexity of a reference monitor implementation is therefore a reasonable metric of quality.

A TCB can be defined according to the requirement, but commonly it contains;

- The kernel
- The configuration files that control system operation
- Any program that runs with the privilege or access rights to alter the kernel or the configuration files

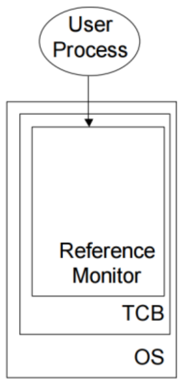


Figure 5.2: Reference Monitor and TCB

Over time, the concepts of trusted systems continued evolve due to the confidence it gained with trusted technology and new applications. This, in turn, creates demands for new capabilities. This has introduced new alternative approaches to access control including non-access control policies, the role of separation kernels to deal with complexity, and the presence of multiple access control policies.

3.4.3. Secure Design Guidelines

3.4.3.1. Separation of Duties

Separation of duties (SoD) is a notable security feature to manage conflict of interest, its appearance, and fraud. It restricts the amount of power held by any one individual by enforcing barriers in place to prevent fraud that may be perpetrated by one individual. It is a key concept of internal controls and is the most difficult and sometimes the most costly one to achieve. This objective is achieved by disseminating the tasks and associated privileges for a specific security process.

3.4.3.2. Principle of Least Privilege

The least privilege principle states that every program and every user of the system should operate using the least set of privileges necessary to complete their tasks. The principle essentially reduces the number of potential interactions among privileged programs to the minimum for correct operation, and ensures that unintentional, unwanted, or improper uses of privilege are less likely to occur. Primarily, this principle limits the damage that can result from an accident or error. With this design strategy, it is required to audit a least amount of programs upon detecting misuse of privileges.

3.4.3.3. Least Common Mechanism

Least common mechanism [42] minimizes the amount of mechanisms common to more than one user and depended on by all users. With that, every shared mechanism represents a potential information path between users and must be designed with great care to make sure it does not unintentionally compromise security.

3.4.3.4. Economy of Mechanism

Economy of mechanism is a well-known principle applies to any aspect of a system, but it deserves emphasis for protection mechanisms. This ensures that design and implementation errors results in unwanted access paths will not be noticed during the normal use of operation. Due to this reason different protection mechanisms such as line-by-line inspection of software and physical examination of hardware are required to be in place.

3.4.3.5. Complete Mediation

This concept states that every access to every object must be checked for authority [42]. It forces a system-wide view of access control, which in addition to normal operation including initialization, recovery, shutdown, and maintenance etc. systematically applied complete mediation will be the primary underpinning of the protection system in any system. The principle also restricts the caching of information, which often leads to simpler implementations of mechanisms.

Whenever a subject attempts to read an object, according to this principle the operating system should mediate the action. First, it determines if the subject is allowed to read the object. If allowed, it provides the resources for the read to occur. The system should check that the subject is still allowed to read the object upon user retrying to read the object again. Most systems would not make the second check. They would cache the results of the first check and base the second access on the cached results, which introduces a potential security risk to the system.

3.4.3.6. Open Design

This principle suggests that complexity does not add security, which is another supportive factor in analyzing the relationship between complexity and security. The principle states that the design should not be a secret and the designers and implementers of a program must not depend on secrecy of the details of their design and implementation to ensure security. If the security strength of a program depends on the ignorance of the user, then is it highly likely that a user with adequate knowledge will defeat that security mechanism. This behavior is described by the concept call the "security through obscurity".

The principle strongly holds for cryptographic software and systems; since cryptography is a highly mathematical subject, vendors in cryptographic business or the ones are in use cryptography to protect user data are frequently trying to keep their algorithms secret. However in this context, keeping cryptographic keys and passwords secret does not violate this principle, because a key is not an algorithm, whereas keeping the enciphering and deciphering algorithms secret would violate it.

3.4.4. Security models

3.4.4.1. The Chinese-wall Security Policy

The Chinese-wall Security Policy [43] is a principle that derives from the ability to read or write information. The principle states that information with mutual interest on an environment where there is a conflict of interest should be prohibited. There are three different levels in Chinese Wall Model:

- Objects: is the lowest level of the chart. For an example an object can be any information about a company.
- Company: Group of businesses in the same class.
- Conflict of Interest Class: Type of business. (Eg: business in two different domains, i.e. a bank and a oil company)

3.4.4.2. Simple Security

The basis of the Chinese-wall policy is that objects are only allowed access to information which is not held to conflict with any other objects that they already possess.

3.4.4.3. Bell-La Padula Model (BLP)

The Bell-La Padula Model of protection systems deals with the control of information flow [44]. It is a linear non-discretionary mode which consists of the following components:

- A set of subjects, a set of objects, and an access control matrix.
- Several ordered security levels. Each subject has a clearance and each object has a classification which attaches it to a security level. Each subject also has a current clearance level which does not exceed its clearance level.

3.4.4.4. Biba Model

The Biba model has a similar structure to the BLP model, but it addresses integrity rather than confidentiality. Under this model objects and users are assigned integrity levels that form a partial order, similar to the BLP model. The Biba integrity model is similar to the BLP model for confidentiality uses subjects and objects, in which, it controls object modification in the same way that BLP controls disclosure. Integrity levels in the Biba model indicate degrees of trustworthiness, or accuracy, for objects and users, rather than levels for determining confidentiality.

3.4.4.5. Clark-Wilson Model

The Clark-Wilson model defines a set of rules based on commercial data processing practices [45]. It concerned with information integrity using an integrity policy that defines enforcement rules (E) and certification rules (C). The model built upon principles of change control rather than an integrity level, which establishes a system of subject-programs -object relationship.

3.5. Control flow Graphs and Integrity

3.5.1. Control Flow Graphs

A control flow graph is a directed graph in which each node represents a basic block and each edge represents the flow of control between basic blocks [19]. A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The approaches differ with respect to the handling of branching and the merging of branches, and the representation of segments of statements that are always executed together.

3.5.2. Secure Control Flow

Most of the sophisticated attacks against computers take advantage of software flaws, such as buffer-overflow or integer-overflow vulnerabilities.

```
Program Sums
1.      read(n);
2.      i = 1;
3.      sum = 0;
4.      while (i<= n) do
5.          sum = 0;
6.          j = 1;
7.          while (j <= i) do
8.              sum = sum + j;
9.              j = j + 1;
           endwhile;
10.         write(sum, i);
11.         i  = i + 1;
        endwhile;
12.     write(sum, i);
end Sums
```

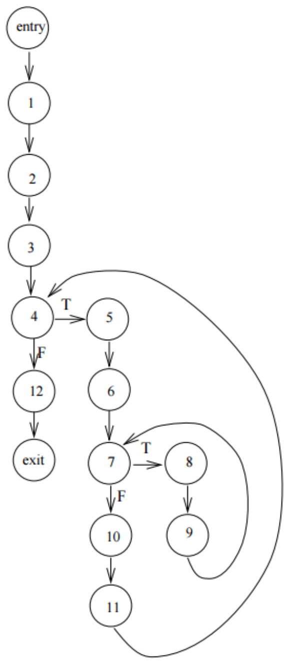


Figure 5.3: A sample code segment and its control flow graph

3.5.3. Control Flow Integrity

Control-Flow Integrity (CFI) means that the execution of a program dynamically follows only certain paths, along with a static policy. CFI can prevent attacks that, by exploiting buffer over- flows and other vulnerabilities, such as attempts to control program behavior. Control-flow integrity has historically been considered a strong defense against control-flow hijacking attacks and Return Oriented Programming (ROP) attacks, if implemented to its fullest extent.

Arbitrary malicious code execution, caused by buffer overflow and stack or heap manipulations are one of major threats in computer security. Due to the fact that many hardware-enforced security features are introduced in recent processors, the attackers are starting to explore the other advanced techniques. As stated in chapter 3, SMEP, SMAP and NX enforcements are powerful security features that can significantly prevent malicious code modification and execution. They make traditional attacking methods, like code injection or user arbitrary code execution, extremely harder than ever. As a result of that, attackers have begun excavating other opportunities like control flow hijacking by misusing the existing machine code execution without code injection to achieve the same or similar malicious behaviors.

Control flow integrity effectively provides a way that can prevent control flow hijacking attacks from arbitrarily controlling program behaviors. Unlike a legitimate control flow execution in an application, the hijacked control flow generally has many significant differences, like too many indirect JMP, CALL, and RET instructions, calling a procedure without corresponding RET (or vice versa) etc.

3.6. Related Software Attacks

Memory related attacks have become the most popular, yet powerful type of attack vectors in the reason past, due to the use of memory-unsafe programming languages in software development. Operating systems and compiler vendors have introduced various protection mechanisms to combat these potential system level exploitations. Unfortunately those hardening techniques are still inadequate to completely mitigate these attacks. This section highlights few notable control flow integrity based software attacks.

3.6.1. Control-Flow Hijacking

Hijacking the control flow of the program in order to execute the malicious code or already-existing code inside the address space of the program is a common way of exploiting a memory corruption [46]. These techniques led to memory corruption bugs to change the target of indirect branch instructions such as CALL*, JMP* or RET, which allows an attacker to gain the complete control of the next instruction to be executed. Control flow based hijacking is possible when an attacker gains control of the instruction pointer register.

3.6.2. Code-Reuse Attack

Code-reuse attacks are software exploits in which an attacker directs control flow through existing code with a malicious result. It allows attackers to execute arbitrary code on a compromised machine. As an example, ROP is an effective code-reuse attack in which short code sequences ending in a ret instruction are found within existing binaries and executed in arbitrary order by taking control of the stack. This is an attack exemplified by return-oriented and jump-oriented programming approaches, thus avoiding the need for explicit injection of attack code on the stack. Since the executed code is reused existing code, the attack bypass the current hardware and software security measures that prevent execution from data or stack regions of memory.

3.6.3. Non-Control-Data Attacks

A non-control-data attack is an attack where memory corruption vulnerability is used to corrupt only data, but not any code pointer. Depending on the circumstances, these attacks can be as effective as arbitrary code-execution attacks; for an example by corrupting the parameter to a sensitive function may allow an attacker to execute arbitrary programs. An attacker may also be able to overwrite security configuration values and disable security checks. Due to the fact that most defense mechanisms focus on the protection of code pointers, these attacks are realistic threats and hard to defend against.

3.6.4. The Stack

A stack is contiguous block of memory which is used by functions, two instructions are used to put or remove data from stack, PUSH puts data on stack, & POP removes data from stack. The stack works on Last in First out LIFO basis and grows downwards towards lower memory addresses on Intel based systems.

3.6.5. Buffer Overflow Attacks

A buffer overflow, probably the most common security vulnerability occurs when a program attempts to put more data in a buffer than it can hold or else when a program attempts to put data in a memory area past a buffer. A buffer is a section in memory in which is allocated to contain anything from a character string to an array of integers.

Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause the execution of malicious code. Attackers use buffer overflows to corrupt the execution stack of programs, by sending carefully crafted input to the application, in a way such that the attacker can cause the application to execute malicious code of his interest, and thereby effectively taking over the machine.

Essentially the buffer overflow exploit takes the advantage of programs that requires user inputs. The attack can exploit in two forms; stack based and heap based. In a stack-based buffer overrun, programs are being exploited by the stack. In a normal situation the stack will be empty until and unless a program requires a user input. At that point, the program writes a return memory address to the stack and then the user's input is placed on top of it. On the other had heap-based attacks flood the memory space reserved for a program, but the difficulty involved with performing such an attack makes them rare.

3.7. Summary

This chapter provides details about information theory entropy and its significance with security. The details mentioned in this chapter were directly adopted in the research at the information quantification phase. The chapter provides a significant amount of information to the research in the context security principles and potential attacks. It has been identified that traditional software attacks can be mitigated to a great extent by adopting different software and hardware based mechanisms that are available, such as microprocessor level enhancements. However, the attack surface has been significantly changed over time, and at present different set of attack vectors exists that can be hardly addressed by existing security mechanisms. This chapter has shown the importance in deriving techniques to analyze the security of systems with ever changing security world.

CHAPTER 4

4. Security Design Concepts for Virtualized Environments

4.1. Introduction

This chapter provides details about virtualization, their challenges and solutions on x86-architecture. Starting with an introduction to virtualization, the chapter then moves towards the virtualization requirements and traditional challenges on x86 that hindered introducing virtualization. It also discuss about hypervisors and their contribution towards virtualization followed by different virtualization techniques.

Apart from virtualization methods, the chapter has paid its close attention on hardware assisted virtualization and the enhancements introduced by Intel-VT microprocessors. The last few sections of the chapter have evaluated the microprocessor level extensions that Intel-VT has introduced and their applications.

4.2. Background of Virtualization Technologies

The concept of Virtualization is originated in the mainframe days in the late 1960s and early 1970s, when IBM invested a substantial time and effort in developing robust time-sharing solutions. Time-sharing refers to the shared usage of computer resources among a large group of users, targeting to increase the efficiency of both the users and the expensive computer resources they share [2]. This model represented a major innovation in computer technology: the cost of providing computing capability dropped considerably and it became possible for organizations, and even individuals, to use a computer without actually owning one. Similar reasons are driving Virtualization for industry standard computing today: the capacity in a single server is so large that it is almost impossible for most workloads to effectively use it. The best way to improve resource utilization, and at the same time simplify data center management, is through Virtualization.

Data centers today use Virtualization techniques to make the ideas of physical hardware, create large aggregated pools of logical resources consisting of CPUs, memory, disks, file storage, applications, networking, and offer those resources to users or customers in the form of agile, scalable, consolidated virtual machines. Even though the technology and use cases have evolved, the core meaning of virtualization remains the same enabling a computing environment to run multiple independent systems.

4.3. Why there is a Big Demand for Virtualization?

Among the leading business challenges confronting CIOs and IT managers today are: cost-effective utilization of IT infrastructure; responsiveness in supporting new business initiatives; and flexibility in adapting to organizational changes. Driving an additional sense of urgency is the continued climate of IT budget constraints and more stringent regulatory requirements. Virtualization [15] is a fundamental technological innovation that allows skilled IT managers to deploy creative solutions to such business challenges Virtualization technology is possibly the single most important issue in IT and has started a top to bottom overhaul of the computing industry. The increasing awareness of the returns provided by virtualization technology is brought about by economic factors of scarce resources, government regulation, and more competition.

Virtualization is being used by a growing number of organizations to reduce power consumption and air conditioning needs and trim the building space and land requirements. Virtualization also provides high availability for critical applications, and streamlines application deployment and migrations. Virtualization can simplify IT operations and allow IT organizations to respond faster to changing business demands.

4.4. Security Issues Associated

Virtualization systems are complex systems and they are susceptible to vulnerabilities. Vulnerability in an operating system or an application may lead to the compromise of a single server within an infrastructure. When that vulnerable operating system or application is of a single compromise can increase across all other virtual machines within the same physical machine.

Virtualized infrastructure associates risks with virtualization. When the boundary between a virtual machine and a host machine becomes transparent (through vulnerabilities), the risk of significant data exposure and system compromise increases dramatically. Classifying the data and types of virtual machines that run on the same physical machine can reduce this exposure.

Major security issues associated with virtualization have been listed below.

- I/O service based DoS attacks.
- Unauthorized memory dumps
- Memory reuse without scrubbing
- Direct Memory Access (DMA) attacks

The recent increase in the use of Virtualization products and services has been driven by many benefits. One of the most common reasons for adopting virtualization is operational efficiency, and as a result of that the organizations got the ability to use their existing hardware (and new hardware purchases) more efficiently by putting more loads on each computer. In general, servers using virtualization technologies can use more of the computer's processing and memory resources than servers running a single OS instance and a single set of services. Recent advances in CPU architectures have made virtualization capabilities faster than it was just a few years ago, and similar advances are expected to continue to be made both by CPU vendors and Virtualization software vendors. Also, CPU architecture changes have made virtualization more secure by strengthening hypervisor restrictions on resources.

Different approaches were taken to address security issues in virtual systems, but most of them are software based approaches. Most of the research approaches taken with respect to secure virtualization are hypervisor based and are vendor specific. The issue with these software based approaches is that they are vulnerable to any hardware based attack. There is need in having security assurance in hardware level and this research addresses that requirement.

4.5. Need for Secure Virtualization

Virtualization technologies do not just enable software abstraction but can also give stricter control over the computing platform resources, which in turn, allows creation of secure execution environments on server and application platforms [47]. For single CPU systems the performance and security tradeoffs provided through different virtualization technologies are rather well understood. However, this is not true for multi-core systems.

Virtualization adds layers of technology, which can increase the security management burden by requiring additional security controls. Combining many systems onto a single physical computer can cause a larger impact if a security compromise occurs. Further, some Virtualization systems make it easy to share information between the systems; this convenience can turn out to be an attack route if it is not carefully controlled. In some cases, virtualized environments are quite dynamic, which makes creating and maintaining the necessary security boundaries more complex.

Security in virtual environments will be the key feature in near future. There are, of course, many ways to implement virtualization in an organization. Some of those ways include server virtualization, network virtualization, storage virtualization, and desktop virtualization. Many companies choose to use one of multiple methods to bring their businesses up to date with all the latest technology, but each type does present challenges when confronting security risks. That's why there are security solutions for each virtualization strategy. It's important to note that while virtualization can improve security.

4.6. Challenges in virtualizing x86 Architecture

To address the different virtualization challenges, hypervisor designers have developed novel solutions by modifying guest software. Paravirtualization offers high performance and does not require making changes to guest applications. A disadvantage of paravirtualization is that it limits the range of supported operating systems. For example, Xen cannot currently support an operating system that its developers have not modified. There are number of challenges in introducing virtualization to x86 platform, and some notable factors are described below.

4.6.1. Ring Aliasing

Ring aliasing [27] where the exact privilege level of a guest OS is exposed, contrary to the belief of the guest that it is running in ring 0. It refers to problems that arise when software is run at a privilege level other than the level for which it was written, for example, executing a PUSH instruction on the CS register, which includes the current privilege level, and then subsequently examining the results would reveal the privilege discrepancy

4.6.2. Address-space Compression

Address space compression [27] is another hurdle for virtualizing the x86 architecture. The operating systems are expecting to have access to the full virtual address space of the processor. Protecting these memory areas is truly challenging, hence this problem address space compression.

4.6.3. Non-faulting Access to Privileged State

An attempt to access protected portions of the CPU is a security concern, prevented by privilege based protection mechanisms. It ensures that unprivileged software cannot access certain components of CPU state. However such access attempts result in faults, allowing a hypervisor to emulate the desired guest instruction, but the IA- 32 architecture includes instructions that access privileged state and do not fault when executed with insufficient privileges. This is a big security concern.

4.6.4. Interrupt Virtualization

Managing external interrupts is challenging on IA-32 architecture due to the fact that the mechanisms it provides for masking external interrupts, preventing their delivery when the OS is not ready for them. Even if it were possible to prevent guest modifications of interrupt masking without intercepting each attempt, challenges would remain when a VMM has a “virtual interrupt" to deliver to a guest. A virtual interrupt should be delivered only when the guest has unmasked interrupts. To deliver virtual interrupts in a timely way, a VMM should intercept some, but not all, attempts by a guest to modify interrupt masking. Doing so could significantly complicate the design of a VMM.

4.6.5. Ring Compression

IA-32 architecture has two mechanisms to protect the hypervisor from guest software at ring de-privileging situations; called segment limits and paging. However segment limits do not apply in 64-bit mode, therefore paging must be used in this mode. With this behavior, the guest OS will run at the same privilege level as guest applications and will introduces a security risk, due to the fact that the IA-32 paging does not distinguish privilege levels 0-2, the guest OS must run at privilege level 3.

4.7. Popek and Goldberg Virtualization Requirements

The Popek and Goldberg virtualization requirements [48] are a set of conditions sufficient for computer architecture to support system virtualization efficiently. Even though the requirements are derived under simplifying assumptions, they still represent a convenient way of determining whether computer architecture supports efficient virtualization and provide guidelines for the design of virtualized computer architectures. Popek and Goldberg provide a set of sufficient (but not necessary) conditions for virtualization. The authors have classified the Instruction Set Architecture in to three different groups.

- Privileged instructions
  Instructions that are providing traps if the processor is in user mode and do not provide traps if it is in system mode (supervisor mode).

- Control sensitive instructions
  Those that attempt to change the configuration of resources in the system.

- Behavior sensitive instructions
  Instructions, whose behaviors or result depend on the configuration of resources.

4.8. Hypervisors

4.8.1. Different types of VMMs

The evolution of virtualization greatly revolves around one piece of very important software, called the hypervisor. A hypervisor is a program that allows multiple operating systems to share a set of common system resources. Under a hypervisor each operating system appears to have a processor, memory and other resources dedicated to it. However, the hypervisor is actually controlling their underlying hardware resources of the host system while making sure the guest operating systems, i.e. the guests cannot disrupt each other. To further clarify the technology, it is important to analyze a few key definitions:

- Guest Machine

A guest machine the VM, is the workload installed on top of the hypervisor. It has the same functionalities as a physical or hosted virtual machine, having its own operating system, installed applications, processes, I/O requests and other related services, which are provided by the machine on which the guest is hosted. The guest can be a virtual appliance, operating system or any other type of virtualization-ready workload.

- Host Machine

The host machine is known as the physical host. It is the host and the components that make up a virtual machine. The guest machine is an independent instance of an operating system and associated software, whereas that of the host machine is the hardware component that provide the guest with computing resources such as processing power, memory, disk and network I/O. etc.

There are two different types of hypervisors namely type 1 and type 2 hypervisor

4.8.1.1. Type I Hypervisor

Type 1 hypervisor is deployed as a bare-metal installation. This means the hypervisor is installed directly on hardware, instead of the operating system. The benefit of this method is that the hypervisor will communicate directly with the underlying physical server hardware, making the resources Para-virtualized and delivered to the running virtual machines.

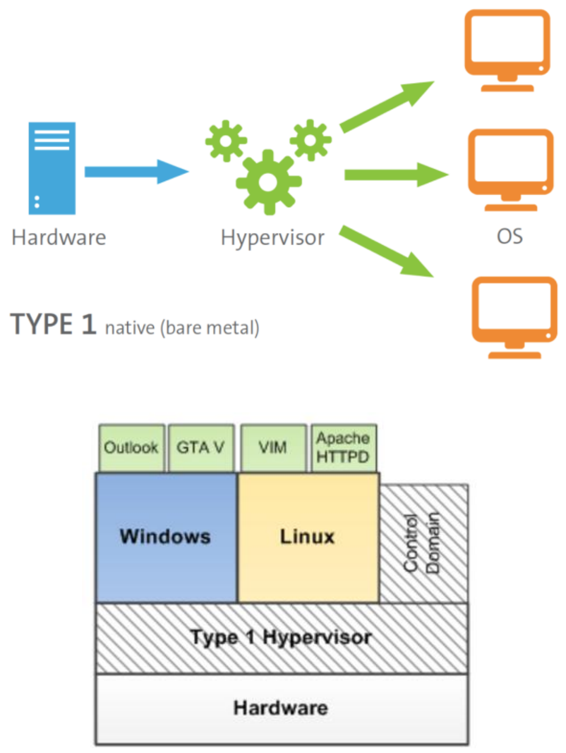


Figure 4.1: Type I hypervisor

4.8.1.2. Type II Hypervisor

Type 2 hypervisor is also known as a hosted hypervisor, where the software is not installed directly onto the bare-metal, but instead is loaded on top of an already live operating system. Bare metal hypervisors (type 1) are faster and more efficient as they do not need to go through the operating system and other layers that usually make hosted hypervisors slower. Although there is an extra hop for the resources to take when they pass through to the virtual machine, the latency is minimal and with modern software enhancements, the hypervisor can still perform optimally. In this context, it can be stated that native hypervisors run directly on the hardware while a hosted hypervisor needs an operating system to do its work.

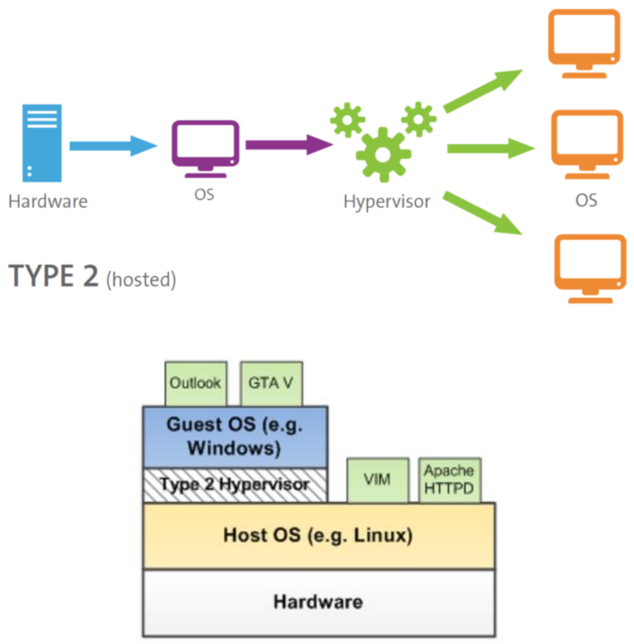


Figure 4.2: Type I hypervisor

4.9. CPU Virtualization

X86 operating systems are designed to run directly on the bare-metal hardware, where they naturally act as if the system is having the full ownership of the hardware. The CPU ring concept [section 3.3.1]) introduced by the X86 architecture allows to maintain a logical separation in terms of execution inside the CPU. In that context, the applications that are running on the user space runs on ring 3, and the operating system and the other programs required direct access towards the memory and hardware runs on ring 0.

Virtualizing the X86 architecture requires placing a virtualization layer under the operating system (which expects to be in the most privileged Ring 0) to create and manage the virtual machines that deliver shared resources. Further complicating the situation, some sensitive instructions cannot effectively be virtualized as they have different semantics when they are not executed in Ring 0 [27]. The difficulty in trapping and translating these sensitive and privileged instruction requests at runtime was the challenge that originally made x86 architecture virtualization look impossible.

There were number of solutions proposed for this problem. One notable technique was binary translation, which addresses the problem by allowing the VMMs to run in Ring 0, while moving the operating system to a user level ring with greater privilege than applications in Ring 3 but less privilege than the virtual machine monitor in Ring 0. There are three alternative techniques for handling sensitive and privileged instructions to virtualize the CPU on the x86-architecture, and they are listed below:

- Full virtualization using binary translation
- OS assisted virtualization or paravirtualization
- Hardware assisted virtualization (first generation)

4.9.1. Binary translation

Binary translation translates kernel code to replace non-virtualizable instructions with new sequences of instructions that have the intended effect on the virtual hardware, where, the user level code is directly executed on the processor, making the execution more efficient. This combination of binary translation and direct execution provides Full Virtualization since the guest OS is completely decoupled from the underlying hardware by the virtualization layer.

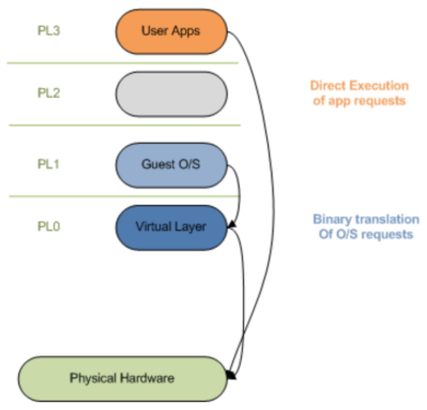

PL3
User Apps
PL2
PL1
Guest O/S
PL0
Virtual Layer
Physical Hardware
Direct Execution of app requests
Binary translation Of O/S requests

Figure 4.3: Binary translation

With binary translation, the guest OS is not aware that it is being virtualized and requires no modification. However, certain sensitive and privileged instructions were identified as the biggest challenge in introducing virtualization to x86-platform. Full virtualization successfully addressed this by eliminating the requirement of having hardware assistance or the operating system assist to virtualize sensitive and privileged instructions. It offers the best isolation and security for virtual machines, and simplifies migration and portability as the same guest OS instance can run virtualized or on native hardware.

4.9.2. Paravirtualization

In paravirtualization, the guest operating system is modified, in order to provide a special interface that can be used by the virtual layer to translate non-virtualizable instructions with hypercalls. This method is different from binary translation, where the unmodified OS does not know it is virtualized and sensitive system calls are trapped using binary translation. However paravirtualization introduces a significant number of support and maintainability issues, as it required operating system kernel level modifications.

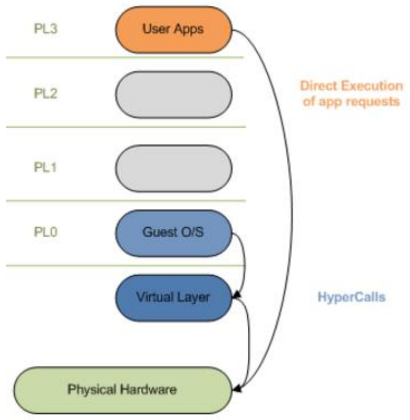


Figure 4.4: Paravirtualization

4.9.3. Hardware Assisted Virtualization

In hardware-assisted virtualization [12], the hardware provides architectural support that facilitates building a virtual machine monitor and allows guest operating systems to be run natively on host hardware resources. With this method the operating system kernels expect direct CPU access running in Ring 0, which is the most privileged level. Before divining in to the details of hardware-assisted virtualization, it is required to have a clear understanding on what made hardware-assisted techniques different from software based technique.

With software virtualization, the guest cannot run in Ring 0, due to the fact that the virtual machine monitor runs there. Therefore the guest operating systems must run in

Ring 1, except some exceptional scenarios. This is because certain x86 instructions run only in Ring 0, introducing an overhead to the operating system by introducing a need to recompile them. This method is called paravirtualization; an infeasible approach due to several drawbacks of it, especially when the source code of the guest operating system is unavailable. As a solution, the hypervisor traps these instructions and emulate them on behalf of the guest, making an enormous performance hit, and at most making the virtualized system significantly slower than a real machine.

To address the above mentioned drawbacks, microprocessor vendors have introduced modifications to their existing processors along with some hardware level capabilities to support virtualization. In line with that, Intel [22] and AMD [23] have reviled their new virtualization technologies, a handful of new instructions and crucially a novel definition to privilege levels, allowing the hypervisor to run at "Ring -1"; which enables the guest operating systems can run in Ring 0.

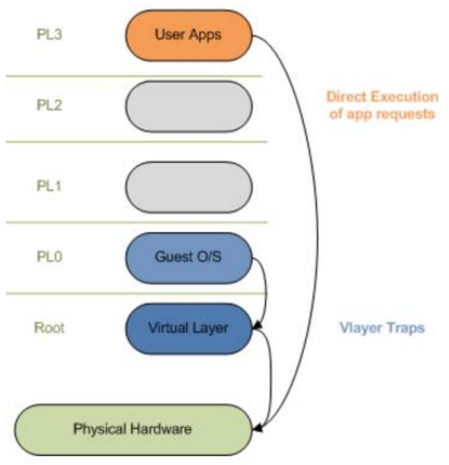


Figure 4.5: Hardware-assisted virtualization

Hardware assisted virtualization however requires the CPU that support (Eg: Intel-VT and AMD-V). With this method, the virtual layer resides in a new root mode privilege in level 0. Privileged and sensitive calls from the guest are set to auto trap to the hypervisor while user request are executed directly to the CPU.

4.10. Intel Virtualization Technology

Intel VT is a function to support virtualization within the processor [32]. It works similarly to exception handlers in software. The purpose of this section is to describe the virtual machine architecture and its newly introduced instructions along with Intel-VT implementation. The newly introduces instructions are called Virtual Machine Extinctions (VMX) which supports virtualization of processor hardware for multiple software environments.

4.10.1. VMX Instructions

The Intel VT extensions provide 10 or 12 new instructions, depending on if the CPU supports EPT (Extended Page Tables) to the Instruction Set Architecture (ISA) in order to support virtualization. The list of VMX instructions are listed below.

- **VMPTRLD:** Loads a physical address from memory, which points to a VMCS and marks it *active*.
- **VMPTRST:** Stores a physical address to memory, which points to a VMCS and marks it *clear*.
- **VMCLEAR:** Sets the launch state of the VMCS to *clear*.
- **VMREAD:** Reads a field in the VMCS.
- **VMWRITE:** Writes a field in the VMCS.
- **VMCALL:** Allows the guest OS to explicitly pass execution to the VMM (Virtual Machine Manager) or hypervisor.
- **VMLAUNCH:** Starts the VM pointed to by the loaded VMCS.
- **VMRESUME:** Resumes a launched VM pointed to by the loaded VMCS.
- **VMXOFF**: Leave VMX operation.
- **VMXON:** Loads a physical address from memory, which points to a VMXON-region and enters VMX operation.
- **INVEPT:** Invalidates entries in the TLB (Translation Look-aside Buffer) associated with EPT.
- **INVVPID:** Invalidates entries in the TLB associated with EPT and a particular VPID (Virtual Process ID).

Depending on the status of the execution, i.e. whether the execution succeeds or fails, VMX instructions will set particular bits in the flags register. VMCS level manipulations are critical and therefore Intel only supports using the VMREAD and VMWRITE instructions for access and manipulation of the VMCS. This instruction works similar to RDMSR (Read MSR) and WRMSR (Write MSR) that were there on previous Intel microprocessors.

4.10.2. VMX Operation

Intel-VT has introduced two new processor modes, called "*mode of operation*". There are two kinds of VMX operations:

- VMX root operation and
- VMX non-root operation

The hypervisor (the Virtual Machine Monitor) runs on in VMX root operation and guest software will run in VMX non-root operation. The principle difference in VMX root mode and non-root mode is that, all the newly introduced instructions under Intel-VT (VMX instructions) can only be executed only when the processor is in root mode. Additionally, the values that can be loaded into certain control registers are also limited in this mode.

The processor behavior in VMX non-root mode is restricted and modified to facilitate virtualization, while facilitating certain instructions to cause VMExits to the VMM. Unlike in VMX-root mode, the functionality in VMX-non root operation is limited, giving VMM a greater controllability. This limitation allows the VMM to retain control of processor resources. Interestingly in this architecture, there is no software visible bit whose setting indicates whether a logical processor is in VMX non-root operation, allowing the VMM to prevent its guest software from determining that it is running in a virtual machine. Due to this reason, VMX operation places restrictions even on software running with current privilege level (CPL) 0, guest software can run at the privilege level for which it was originally designed. This capability may simplify the development of a VMM.

4.10.3. VMX Transitions

Transitions between VMX root operation and VMX non-root operation are called VMX transitions. There are two kinds of VMX transitions. Transitions into VMX non-root operation are called *VM entries*. Transitions from VMX non-root operation to VMX root operation are called *VM exits*. The figure given below illustrates the life cycle of a VMM and its guest software as well as the interactions between them.

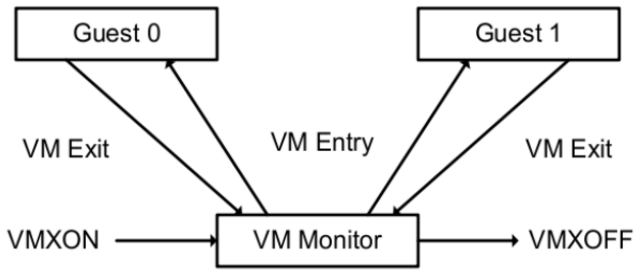


Figure 4.6: VMM and the guest interaction

First, the VMXON instruction allows the VMM to enter VMX operation. Using VM entries, a VMM can then enter guests into virtual machines, whereas VM exits transfer control to an entry point specified by the VMM. In this case the VMM affects a VM entry using instructions VMLAUNCH and VMRESUME; it regains control using VM exits.

4.10.4. Virtual Machine Control Structure (VMCS)

VMX non-root operation and VMX transitions are controlled by a data structure called a virtual-machine control structure (VMCS), which is defined for VMX operation. A VMCS manages transitions in and out of VMX non-root operation (VM entries and VM exits) as well as processor behavior in VMX non-root operation. The access to the VMCS is managed through a component of processor state called the VMCS pointer, whose values is the 64-bit address of the VMCS, in which manipulated by VMPTRST and VMPTRLD instructions. The VMCS is defined for VMX operation, which manages transitions in and out of VMX non-root operation as well as processor behavior in VMX non-root operation.

4.10.5. VMX-non-root Operation

Under VMX operation, the guest software stack typically runs on a logical processor in VMX non-root operation, which is similar to that of ordinary processor operation outside of the virtualized environment. VMX non-root mode describes the operation of VM entries which allow the processor to transition from VMX root operation to non-root operation.

4.10.6. Instructions that cause VMExits

Certain instructions may cause VM exits if executed in VMX non-root operation. These exits can be of below categories;

- Conditional VMExits
- Unconditional VMExits
- APIC Access VMExits
- Other causes of VMExits

Details about these scenarios are listed below.

Certain instructions cause VM exits in VMX non-root operation depending on the setting of the VM-execution controls. The following instructions can cause "fault-like" VM exits based on the conditions described.

4.10.6.1. Instructions causing VMExits conditionally

- **CLTS:** The CLTS instruction causes a VM exit if the bits in position 3 (corresponding to CR0.TS) are set in both the CR0 guest/host mask and the CR0 read shadow.
- **HLT:** The HLT instruction causes a VM exit if the "HLT exiting" VM-execution control is 1.
- **IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD**: The behavior of each of these instructions is determined by the settings of the "unconditional I/O exiting" and "use I/O bitmaps" VM-execution controls.
- **INVLPG:** The INVLPG instruction causes a VM exit if the "INVLPG exiting" VM-execution control is 1

- **LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR:** These instructions cause VM exits if the “descriptor-table exiting” VM-execution control is 1
- **LMSW:** In general, the LMSW instruction causes a VM exit if it would write, for any bit set in the low 4 bits of the CR0 guest/host mask, a value different than the corresponding bit in the CR0 read shadow.
- **MONITOR:** The MONITOR instruction causes a VM exit if the “MONITOR exiting” VM-execution control is 1.
- **MOV from CR3:** The MOV from CR3 instruction causes a VM exit if the “CR3-store exiting” VM-execution control is 1. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
- **MOV from CR8:** The MOV from CR8 instruction (which can be executed only in 64-bit mode) causes a VM exit if the “CR8-store exiting” VM-execution control is 1.
- **MOV to CR0:** The MOV to CR0 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR0 guest/host mask, the corresponding bit in the CR0 read shadow.
- **MOV to CR3:** The MOV to CR3 instruction causes a VM exit unless the “CR3-load exiting” VM-execution control is 0 or the value of its source operand is equal to one of the CR3-target values specified in the VMCS.
- **MOV to CR4:** The MOV to CR4 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR4 guest/host mask, the corresponding bit in the CR4 read shadow.
- **MOV to CR8:** The MOV to CR8 instruction (which can be executed only in 64-bit mode) causes a VM exit if the “CR8-load exiting” VM-execution control is 1. If this control is 0, the behavior of the MOV to CR8 instruction is modified if the “use TPR shadow” VM-execution control is 1.
- **MOV DR:** The MOV DR instruction causes a VM exit if the “MOV-DR exiting” VM-execution control is 1.
- **MWAIT:** The MWAIT instruction causes a VM exit if the “MWAIT exiting” VM-execution control is 1.

- **PAUSE:** The behavior of each of this instruction depends on CPL and the settings of the “PAUSE exiting” and “PAUSE-loop exiting” VM-execution controls.
- **RDMSR:** The RDMSR instruction causes a VM exit if any of the following are true:
    - The “use MSR bitmaps” VM-execution control is 0.
    - The value of ECX is not in the range 00000000H – 00001FFFH or C0000000H – C0001FFFH.
    - The value of ECX is in the range 00000000H – 00001FFFH and bit n in read bitmap for low MSRs is 1, where n is the value of ECX.
    - The value of ECX is in the range C0000000H – C0001FFFH and bit n in read bitmap for high MSRs is 1, where n is the value of ECX & 00001FFFH.
- **RDPMC:** The RDPMC instruction causes a VM exit if the “RDPMC exiting” VM-execution control is 1.
- **RDTSC:** The RDTSC instruction causes a VM exit if the “RDTSC exiting” VM-execution control is 1.
- **RDTSCP:** The RDTSCP instruction causes a VM exit if the “RDTSC exiting” and “enable RDTSCP” VM-execution controls are both 1.
- **RSM:** The RSM instruction causes a VM exit if executed in system-management mode (SMM).
- **WBINVD.** The WBINVD instruction causes a VM exit if the “WBINVD exiting” VM-execution control is 1.
- **WRMSR:** The WRMSR instruction causes a VM exit if any of the following are true:
    - The “use MSR bitmaps” VM-execution control is 0.
    - The value of ECX is not in the range 00000000H – 00001FFFH or C0000000H – C0001FFFH.
    - The value of ECX is in the range 00000000H – 00001FFFH and bit n in write bitmap for low MSRs is 1, where n is the value of ECX.

- The value of ECX is in the range C0000000H – C0001FFFH and bit n in write bitmap for high MSRs is 1, where n is the value of ECX & 00001FFFH.

4.10.6.2. Instructions causing VMExits unconditionally

The following instructions cause VM exits when they are executed in VMX non-root operation:

CPUID, GETSEC, INVD, and XSETBV

All the instructions introduced with Intel-VT (VMX instructions): which include INVEPT, INVVPID, VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF and VMXON.

4.10.6.3. APIC-Access VMExits

An attempt to access memory using a physical address on the APIC-access page causes a VM exit if the "virtualize APIC accesses" VM-execution control is 1. These VMExits are called an APIC-access VM exit.

4.10.6.4. Other causes of VMExits

Apart from the VMExit reasons mentioned above, below conditions will also cause VMExits.

- **Exceptions:**

  Exceptions happen due to VMExits based on the exception bit map. If the bit is 1, the exception causes as a VMExit and it is 0, the exception delivered normally through the IDT (Interrupt Descriptor Table).

- **Triple Faults:**

  Under Triple fault a VMExit occurs if the logical processor encounters an exception while attempting to call the double-fault handler and that exception itself does not cause a VM exit due to the exception bitmap.

- **External Interrupts:**

  An external interrupt causes a VM exit if the “external interrupt exiting” VM-execution control is 1.

- **Non-maskable interrupts (NMIs):**

  An NMI causes a VM exit if the “NMI exiting” VM-execution control is 1. We have eliminated this due to its conditional nature.

- **INIT signals:**

  INIT signals can cause VMExits, but if the logical processor is in the wait-for-SIPI state, INIT signals are blocked. They do not cause VM exits in this case.

- **Start-up IPIs (SIPIs):**

  If a logical processor is not in the wait-for-SIPI activity state when a SIPI arrives, no VM exit occurs and the SIPI is discarded.

- **Task Switches:**

  Task switches are not allowed in VMX non-root operation. Any attempt to affect a task switch in VMX non-root operation causes a VM exit. However there are some checks performed by the processor before causing a VMExit.

- **System-management interrupts (SMIs):**

  If the logical processor is using the dual-monitor treatment of SMIs and system-management mode (SMM), SMIs cause SMM VM exits.

- **VMX-preemption timer:**

  A VM exit occurs when the timer counts down to zero. The timer does not cause VM exits if the logical processor is outside the C-states C0, C1, and C2.

4.11. Security Analysis

Intel Virtualization Technology present on Intel processors enables a new privilege space where the VMM software can operate. It reduces the size and complexity of the

VMM software improving its efficiency and enabling greater functionality. VT capabilities have introduced an extra set of instructions called Virtual Machine Extensions (VMX), to better deal with hardware-assisted virtualization. There has not been any significant research done with respect to the security strength on Intel-VT enabled processors considering the work load of the guest software running on top those VMMs.

It can be argued that the main feature the Intel has introduced with its virtualization technology is its processor modes. The VMX non-root processor mode introduced by Intel is similar to the user space in a traditional OS, whereas the root mode is similar to that of the kernel space. With VT-x the guest OS is running on VMX-non root mode with a reduced set of privileges, but in ring 0.

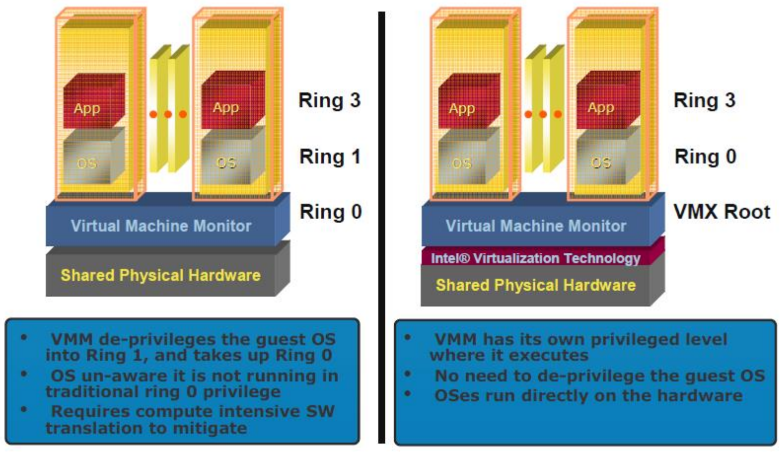

App
App
OS
OS
Ring 3
Ring 1
Ring 0
Virtual Machine Monitor
Shared Physical Hardware
App
App
OS
OS
Ring 3
Ring 0
VMX Root
Virtual Machine Monitor
Intel® Virtualization Technology
Shared Physical Hardware
• VMM de-privileges the guest OS into Ring 1, and takes up Ring 0
• OS un-aware it is not running in traditional ring 0 privilege
• Requires compute intensive SW translation to mitigate
• VMM has its own privileged level where it executes
• No need to de-privilege the guest OS
• OSes run directly on the hardware

Figure 4.7: Pre and Post Intel VT-x

With this design, the guest OS is directly running on hardware with its native ring 0 privileges, but with reduced privileges, i.e. on non-root mode. It can be stated that this feature introduces potential security risk to the system. The virtual machine monitor (the hypervisor) is running with full privileges at VMX-root mode. The guest OS will contact the VMM, whenever it is required to run privileged instructions. A VMX transition will occur at this situation. A VMX transitions is the other important feature that Intel has introduced along with VT. With VMExit instructions, the processor mode will be transferred from the VMX-non root mode to the VMX-root mode, causing a privilege elevation. With VMEnter instructions, the processor mode will be transferred from the root mode to non-root mode downgrading the privilege level.

In security’s view point, any privilege elevation is a concern, and therefore should be minimized during program execution. Going forward with this principle in line with Intel-VT design, VMExit instructions should be treated seriously, assigning a higher criticality level. it has been considered that all the VMExit instructions as the set of privilege gains during the execution. In this situation along with a VMExit, the processor mode will also get changed to a higher privilege, and therefore in between a VMExit and a VMEnter, the processor is in a privileged state. That means the processor state will oscillate in between these two privilege levels according to the behavior of its user programs. This state change of the processor will be taken place not only because the main program of our interest is having a large number of VMX transitions, but also as a result of its other called or linked programs.

4.12. Summary

This chapter contains information about the need of virtualization, virtualization challenges, and solutions provided by hardware vendors. In this chapter a comprehensive analysis on Intel-VT architecture has been performed along with a detailed study about its security aspects. The reason for this analysis is that the evaluation method of the novel framework proposed by this research has been done on a virtualized system.

CHAPTER 5

5. A Novel Risk Evaluation Technique: RECSRF

5.1. Introduction

Being probably the most important chapter of this thesis, this chapter consists of the novel evaluation mechanism proposed by this research, called the RECSRF. The chapter consists of two main sections; the novel method and its rational. Starting with a brief overview of the main problem along with a background study, the chapter then moves towards the introduction of the novel evaluation technique. The next section contains the proposed methodology, and it has been excavated in the context of its practical usage, and finally the chapter concludes with a summary.

5.2. The Impact on Privilege Elevations on Microprocessors

5.2.1. Privilege Escalations

A privilege escalation attack [49] is a type of attack where an unauthorized user gains elevated access to system or other user resources. It is a type of an intrusion that takes advantage of programming errors or design flaws to grant the attacker elevated access to the system and its associated data and applications. These attacks typically exploit software or hardware bugs as well as poor software configurations. Upon gaining elevated privileges, the attacker can access files, view private information (such as encryption keys), modify system files or install unwanted software. There are two types of privilege escalation:

- **Vertical privilege escalation**

  This attack allows the attackers to grant themselves higher privileges. This is typically achieved by performing kernel-level operations that allow the attacker to run unauthorized code. For instance, injecting and executing code at the kernel space, or performing kernel operations that allow unauthorized code execution.

- **Horizontal privilege escalation**

  Horizontal privilege escalation requires the attacker to use the same level of privileges he already has been granted, but it is possible to gain access to resources belonging to other processes or users sharing the same privilege levels.

5.2.2. Privilege Escalation attacks

Planting a privilege escalation attack requires the existence of distinct privilege levels, such as kernel, supervisor, user etc., at the microprocessor architecture specifications. Additionally, software mechanisms that are generally imposed by the operating system that escalate privileges during normal operation are also required. The degree of escalation depends on what privileges the attacker is authorized to possess, and what privileges can be obtained in a successful exploit. For an example, a programming error that allows a user to gain extra privilege after successful authentication limits the degree of escalation, because the user is already authorized to hold some privilege. Likewise, a remote attacker gaining super user privilege without any authentication presents a greater degree of escalation.

Not all instruction set architectures, however, include instructions that directly elevate privileges. Privilege escalation usually occurs in as-needed basis. Therefore, gaining escalated privileges to directly write on kernel structure is another common way that allows this attack. Another way to escalate privileges is to overwrite locations which contain critical information to the system including information about interrupt handlers, shared libraries and operating system specific code etc.

5.2.3. Context Switching

A context is the contents of a CPU's registers and program counter at any point in time. It is the procedure of storing the state of an active process for the CPU when it has to start executing a new one. A register is a small amount of very fast memory inside of a CPU (as opposed to the slower RAM main memory outside of the CPU) that is used to speed the execution of computer programs by providing quick access to commonly used values, generally those in the midst of a calculation. A program counter is a specialized register that indicates the position of the CPU in its instruction sequence and which holds either the address of the instruction being executed or the address of the next instruction to be executed, depending on the specific system. Context switches are resource intensive and most operating system designers try to reduce the need for a context switch. They can be software or hardware governed depending upon the CPU architecture.

5.3. The Importance of Microprocessor Security

It is a conspicuous fact that microprocessors today are ubiquitously deployed in a wide variety of applications, from the personal computers to space and automotive applications. The microprocessor can be treated as the nucleus of a computer system and ensuring its integrity is paramount, since the adverse impacts that it can impose on the system can range from simple information leakage to life-threatening and mission critical circumstances. Unfortunately, apart from the design level and in-built security mechanisms introduced by the underlying architecture and by the processor vendors, a diminutive amount of research is done in the past in relation with microprocessor security. In fact, the high level software programmers rely on these mechanisms and from programmers perspective, the microprocessor is inside the Trusted Computing Base (TCB).

A single CPU cycle itself is pretty complex and consists of data movements, register updates, I/O operations, memory access operations and many more. During a program execution, millions of machine instructions will be generated, and will be executed by the microprocessor. One main reason for this fact is that, a given instruction on a high level programming language will contain more than one machine instruction, i.e. unlike in assembly, high level programming languages does not have one to one mapping to machine instructions. An example of a simple C++ if statement and its corresponding assembly mapping given below shows this behavior.

```
if( op1 == op2 )
  X = 1;
else
  X = 2;
```

```
    mov ax,op1
    cmp ax,op2
    jne L1
    mov X,1
    jmp L2
L1: mov X,2
L2:
```

Figure 5.1: High level to Assembly mapping

The whole point of this argument is to emphasize the significance in closely monitoring microprocessor behavior during program executions. Depending on the instruction sequence of the program, the microprocessor will execute them while going through numerous different states including privilege escalations. The “state” here is a vital and an influential term that has been seriously looked at during this research. As mentioned above, it is the job of the microprocessor to set different CPU flags, register operations and in essence all the background needs of the operation during an instruction execution. Depending on the register values, flag status and other constrains, the microprocessor will be on different states with different capabilities. In essence, it can be stated that the security risk that a microprocessor imposes on a system by being on a high privilege state or by frequent context switching between high-low privilege levels is still a virginal area in security.

The principle of least privilege states that, at a given time the system should be on the lowest possible privilege level. However, the application of the principle is straightforward in software, but in hardware, due to the complex nature of program execution. Any malfunction, vulnerability or any potential control flow hijacking attack [49] during a program execution can cause integrity issues in a system if the system is on a higher privilege at the time of the attack. The potential impact an attacker can make in such situation is unimaginable.

With ever growing software attacks and breaches, it can be stated that there are two types of systems exists in the world; i.e. the systems that were already attacked, and the once to be attacked. This is a dominant challenge that every system or an organization is facing today, and because of this reason, security mechanisms were mandated even at the SDLC (Software Development Life Cycle) stages. This shows the importance of built in security rather than bolt in security and will make sure that security mechanisms are in place by-design and by-default. It is however extremely vital to make sure that a system is protected to some extent even after it is being attacked. The potential impact of an attack will be minimum, given the system was in the least possible privilege at the time of the attack. Therefore the control flow integrity of an execution has become a paramount design principle than ever before. This research focuses on ensuring the control flow integrity of the microprocessor.

5.3.1. The Instruction Pointer Register and its Security Impacts

The role of the instruction pointer register (EIP on 32 bit, or RIP on 64 bit) in the CPU is to provide the address of the next instruction to be executed. It is a special purpose register, whose function is to point to the next instruction to be executed. In essence, microprocessor uses the instruction pointer register to keep track of the location of the next instruction to be executed [27]. The register value increases automatically whenever an instruction is executed by the length of the encoding of that instruction. However this register cannot be directly accessed as there is no legitimate use case to do so. Having any arbitrary instruction change IP register would make branch prediction very difficult, and would probably open up whole lot of security issues.

Therefore, any program activity that results in modifying the instruction pointer register contents has the potential to create a security risk. The most visible security risk factor from a computer architecture perspective is the presence of machine instructions that cause instruction pointer register manipulation.

5.3.2. The Instruction Pointer and Control Transfer Instructions

When the CPU is in the process of instruction execution, instructions are usually fetched sequentially from memory, but control transfer instructions [21] change the sequence by placing a new value in the instruction pointer. These include branches, conditional expressions, subroutine calls, and returns. A branch ensures that the next instruction is fetched from the memory. In addition to branching, a subroutine call will save the proceeding contents of the instruction pointer. A return instruction will save contents of the instruction pointer and places it back in the instruction pointer, resuming sequential execution with the instruction following the subroutine call.

5.3.3. Software complexity impact on Microprocessor Security

Complexity is treated as the enemy of computer security [11], i.e. the more complex a system gets, harder it is to secure. With too many “moving parts” or interfaces between programs and other systems, the system or interfaces become difficult to secure while still permitting them to operate as intended. In this research, it has been observed that the machine instructions generated due to the increased cyclomatic complexity of high-level programs will increase the number of branched instructions during the execution; causing a high volatility in the instruction pointer register.

5.4. The Novel Risk Evaluation Method

Considering the practical scenarios mentioned above, it has been observed that even a normal user of a system can drive the microprocessor through risky operations. This could be either intentional or unintentional, but however the ultimate impact on the system is devastating, since it will be an attack to the nucleus of the system. It has been notices that it is also possible for a normal user to fool the microprocessor with different instructions in a way the processor will be on a higher privilege level, opening the doors for privilege escalation attacks. On the other hand, obtrusively present software complexity will magnifies this risk making attackers life easier.

The attempt of this research is to identify the instances which introduce state changes to the instruction pointer register inside the microprocessor. Accordingly, it has been considered that higher the volatility of the instruction pointer register during a program execution will significantly bring down the overall security of the system. The reason for this argument is that any change in the IP register would make branch prediction extremely difficult, and would increase the probability of system being vulnerable to control flow hijack attacks [49].

To build this argument, a program execution trace has been taken and the points which contain unconditional control transfers were taken in to account, and those instructions were marked as regions. Since the instruction pointer register manipulation itself is a risky operation, any privilege elevation after this state change will be a potential attack attempt. The method proposed in this research will identify this behavior and will quantitatively measure the risk factor involved. Quantifications were done measuring the number of risky regions created and the amount of information generated.

Unfortunately, it has been observed that existing information theoretic methods cannot be used directly, since none of them are capable in dealing with runtime disassembled machine instructions. This challenge has become the motivation behind deriving few novel concepts in this research. The traditional concept of complexity has been extended to a new dimension in order to better suit with real practical needs. The **Runtime Execution Complexity (REC)** present with this research [section 7.4.1] is a novel technique that can be adopted to evaluate the control flow integrity of an execution.

5.4.1. The Runtime Execution Complexity (REC) Concept

The execution of a computer program can be treated as a collaborative execution of several programs. Generally, at run time, a program will call functions from other programs and also transfer its control to other program segments, introducing a change to its control flow. In most cases caller (the main program) is not fully aware about its callee, in terms of the security risks it has, and vice versa. In addition to that, this control transfer will potentially change the trust boundary of the system. Practically it is difficult avoid this behavior because it is required to build applications having such a modular design due to various reasons. The Runtime Execution Complexity (**REC**) has been defined as the overall complexity a program will produce dynamically due to this collaborative execution.

The REC concept is tightly integrated with program control transfers, which is a measure of the number of decision making points in a given program. These decisions making instances will be interpreted to the processor as control transfer machine instructions. Essentially the execution flow of the program will be changed at these decision making points and it has been considered that this control flow change (IP register manipulations) will introduce a security risk to the overall execution and it will results in increasing the runtime execution complexity.

To construct this logic the control transfer instructions were analyzed in depth. As mentioned above, control transfer instructions have two forms; conditional and unconditional. Though both instruction types will change the control flow of a program, this research took only the unconditional control transfers in to account since the intention is to derive a lower bound for the risk. The unconditional control transfer instructions will always be executed, and therefore, by analyzing the disassembled instruction code, it can be exactly stated that at those points the program control has changed. The disadvantage with the conditional ones is that, they are always subjected to be executed successfully if and only if some pre conditions are met. Therefore, conditional instructions involve bit of an uncertainty by looking at it from the disassembled instructions point of view. These reasons paved the way to consider unconditional control transfer instructions in this research.

The proposed framework considers every unconditional control transfer as a potential **control flow hijack** [49, 52] attack and every return as a **malicious return**. The details of this framework and its evaluation methods have been mentioned in proceeding sections of this chapter.

A is a step by step guide to derive program risk factor has mentioned below.

Step1:

First, it is required to attach the process ID (PID) of the program under interest to the disassembled code analysis utility. During this process it is required to make sure that a minimum amount of programs are running in the background. This is to optimize the instruction collection by minimizing the possible instruction contaminations by other programs. This will also be helpful in reducing possible false positives.

Step 2:

The next step is to extract all the unconditional control transfer instructions from the initial instruction trace (the trace obtained from step 1 above). This has been shown in Figure 5.2 below.

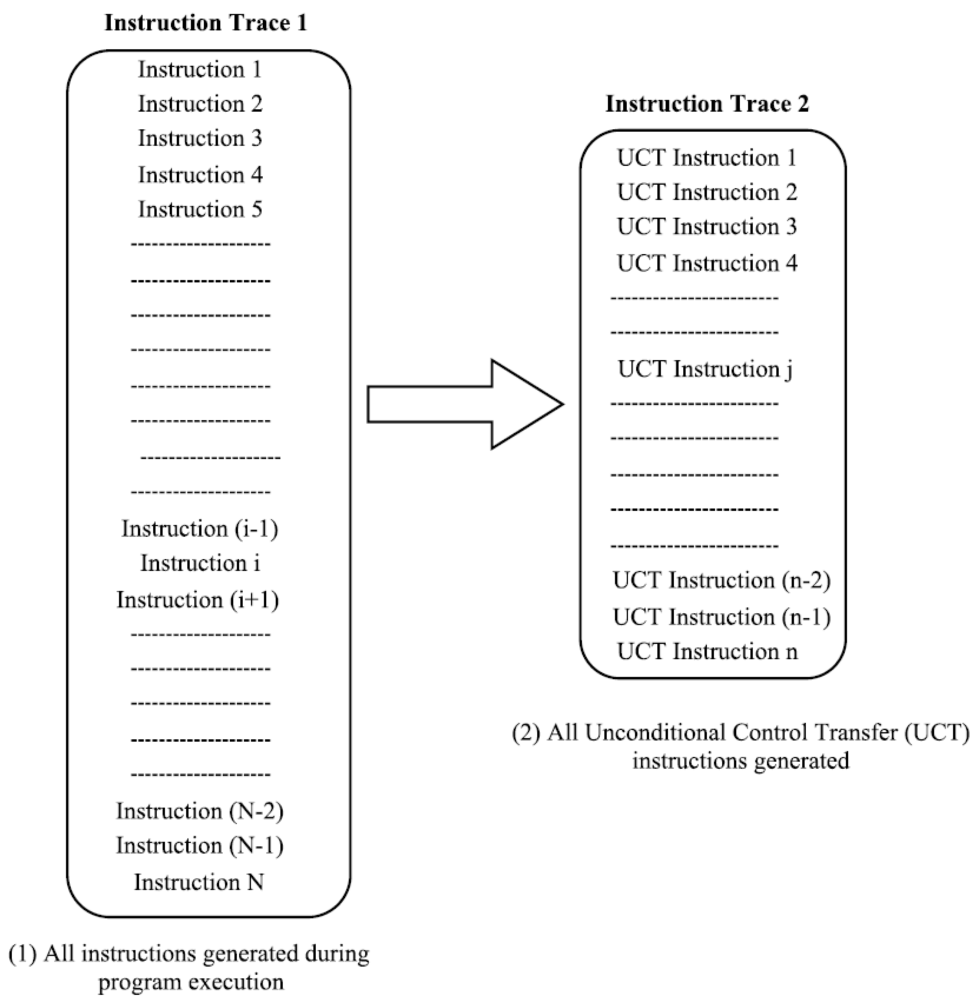


Figure 5.2: Generation of Unconditional Control Transfer Instructions

Step 3:

The next step is to calculate the number of unconditional control transfer instructions in this extracted instruction trace. Assume that there exists ***n*** number of unconditional control transfer instructions in total. That means the original instruction trace derived during the data collection (at step 1) can be divided in to ***(n+1)*** number of regions.

These regions are called "**Threat Blocks**" and it has been considered that these threat blocks will increase the attack surface of the execution. The reason for this argument is that, each threat block is a result of an unconditional control transfer. At each threat block exit the program control will be transferred unconditionally to a different region (which is again a threat block) that may or may not be under the control of the original author. Without losing generality, it has been considered that every threat block is insecure and will introduce a potential risk to the program execution in terms of control flow integrity.

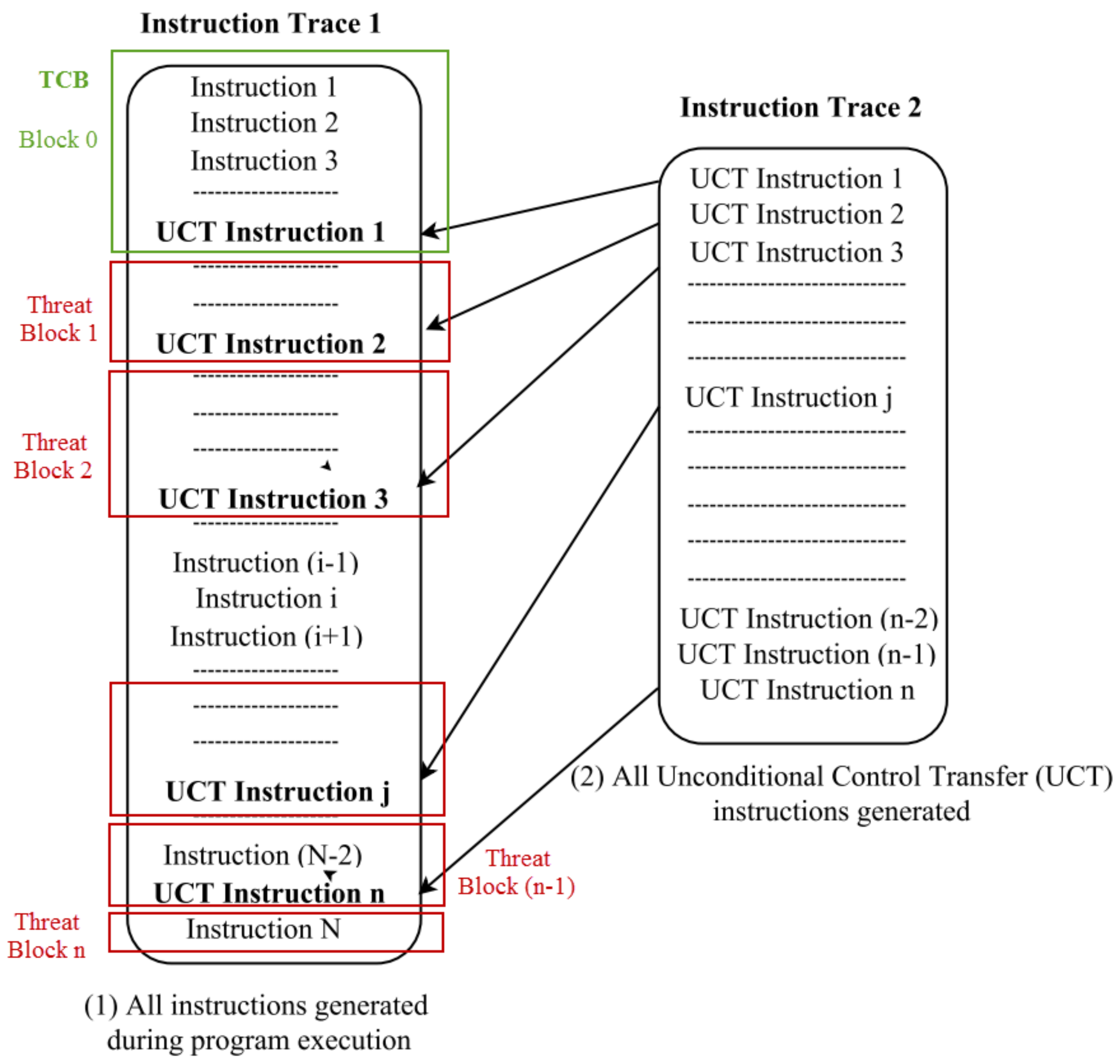


Figure 5.3: Identification of Threat Blocks

Even though there are ***(n+1)*** number of threat blocks, it is possible to eliminate the very first block (Block 0 in Figure 5.3) since it is derived from the main program which the system will trust. This is because in this model, the initial program will be trusted and considers that it is residing in the Trusted Computing Base (TCB). This argument results in having ***{(n+1) -1}***, i.e. ***n*** number of threat blocks.

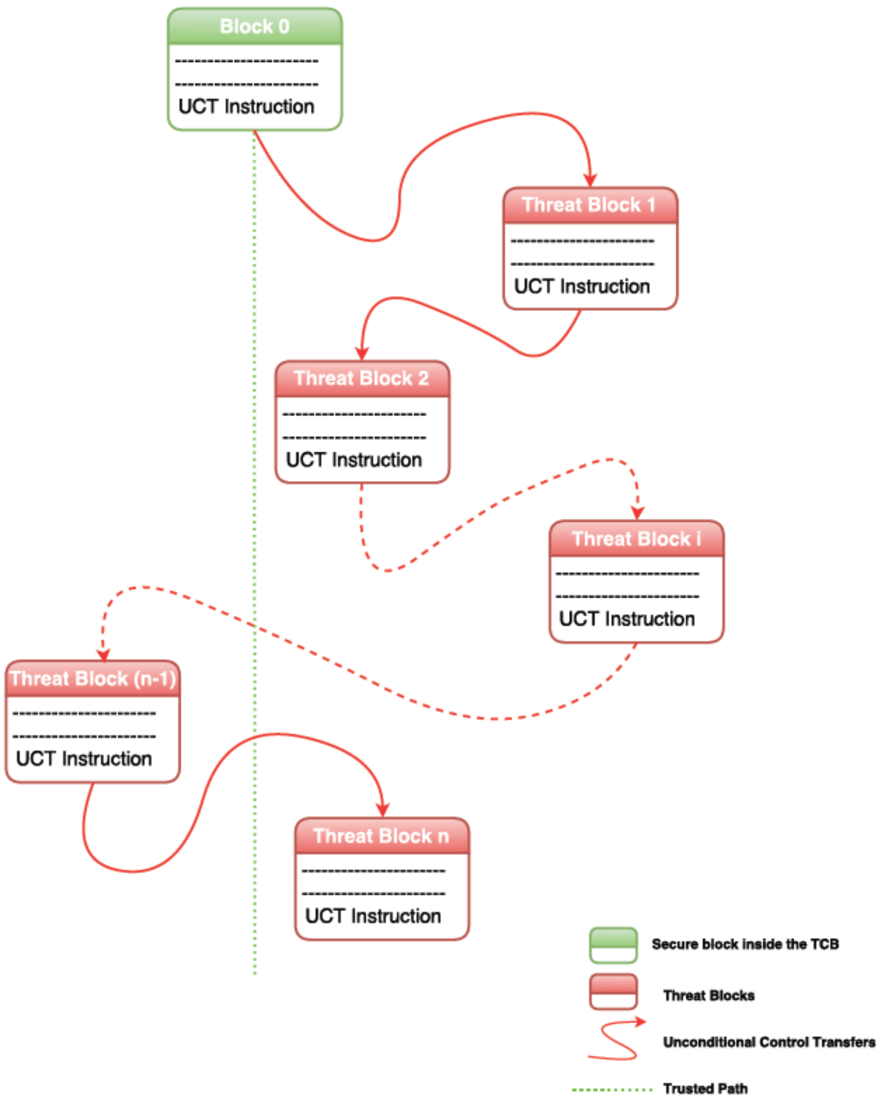


Figure 5.4: Program control transfer and threat blocks creation

Step 4:

At next, it is required to calculate the amount of privileged information generated during this activity. This can be calculated as a measure of information entropy. Assume that there exist ***n'*** number of privileged instructions in the original instruction sequence, the information entropy generated can be calculated as follows;

Total number of instructions in the trace = N

Total number of privileged instructions = n'

Entropy generated H [PRI] = - ∑ {Pi* log (Pi/P)}

$$= - \sum \{\mathbf{n'^*log\ (n'/N)}\} \qquad (5.1)$$

Step 5:

The amount of privileged instructions generated (H [PRIV]) during this activity, is distributed in the above mentioned ***n*** of threat blocks. Therefore the approximation for the amount of privileged information distributed in a given threat block is H [PRI]/ (n). The per threat block privileged information distribution is called the "**Security Risk Factor**".

Combining the Runtime Execution Complexity (REC) as mentioned in section 5.4.1 and the Security Risk Factor (SRF) mentioned in this section, the novel risk quantification method present in this research is called the **RECSRF**.

Hence the RECSRF number of the execution is

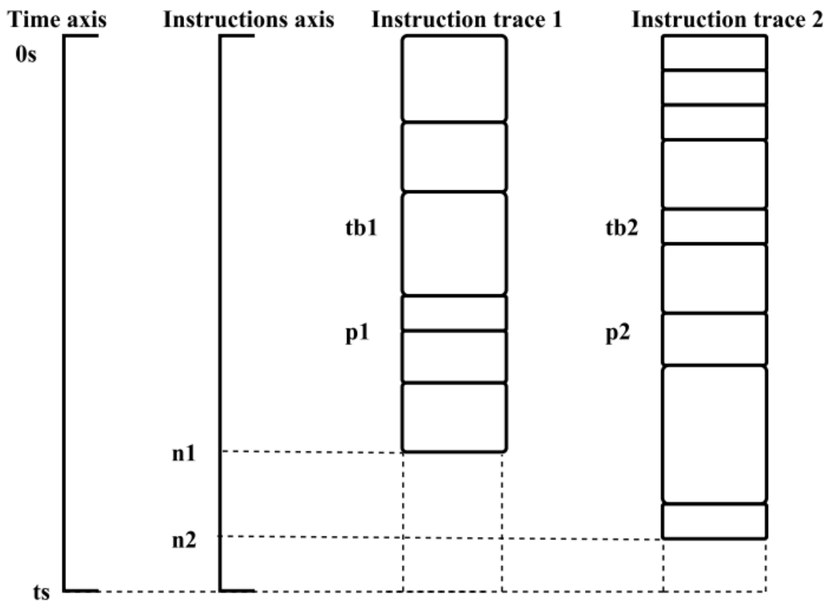
$$\mathbf{RECSRF = H\ [PRI]/\ (n)} \qquad (5.2)$$

Note: Since the frame considers only the unconditional control transfer instructions, the RECSRF value provides a lower bound for the risk factor for a given program execution.

5.4.2. Standardization of Data

Data standardization is the process of reaching agreement on common data definitions, formats, representation and structures of all data layers and elements. One challenge of this research is that the collected data will not always be on the same scale. Therefore different execution traces will contain Threat Blocks of different sizes. Additionally the privileged information distribution will also not be on a state that allows a comparison. To mitigate this challenge, it is required to standardize the data collected in a way it allows comparison.

According to Figure 5.5 shown below, assume that two instruction traces called, instruction trace 1 and instruction trace 2 were collected while executing a task. The data collection will be performed giving an adequate time to complete the execution. Time axis has shown that the data collection took place for t number of seconds (obviously, the unit of time could be selected appropriately). Take the total number of instructions generated by trace 1 to be n1, whereas those of the total generated by trace 2 to be n2.



n1	Total number of instructions on trace 1
n2	Total number of instructions on trace 2
tb1	Total number of threat blocks on trace 1
tb2	Total number of threat blocks on trace 2
p1	Amount of privileged information on trace 1
p2	Amount of privileged information on trace 2

Figure 5.5: Data Standardization

The data standardization technique has mentioned below.

- **Threat Block Size**

Size of the threat block on instruction trace 1 = tb1 x $\left\{\frac{\text{n1}}{(\text{n1}+\text{n2})}\right\}$ (5.3)

Size of the threat block on instruction trace 2 = tb2 x $\{\frac{\text{n2}}{(\text{n1}+\text{n2})}\}$ (5.4)

- **Privileged Information Distribution**

  Amount of privileged information distribution on instruction trace 1

$$= p1 \text{ x } \{\frac{n1}{(n1+n2)}\} \tag{5.5}$$

  Amount of privileged information distribution on instruction trace 2

$$= p2 \text{ x } \{\frac{n2}{(n1+n2)}\} \tag{5.6}$$

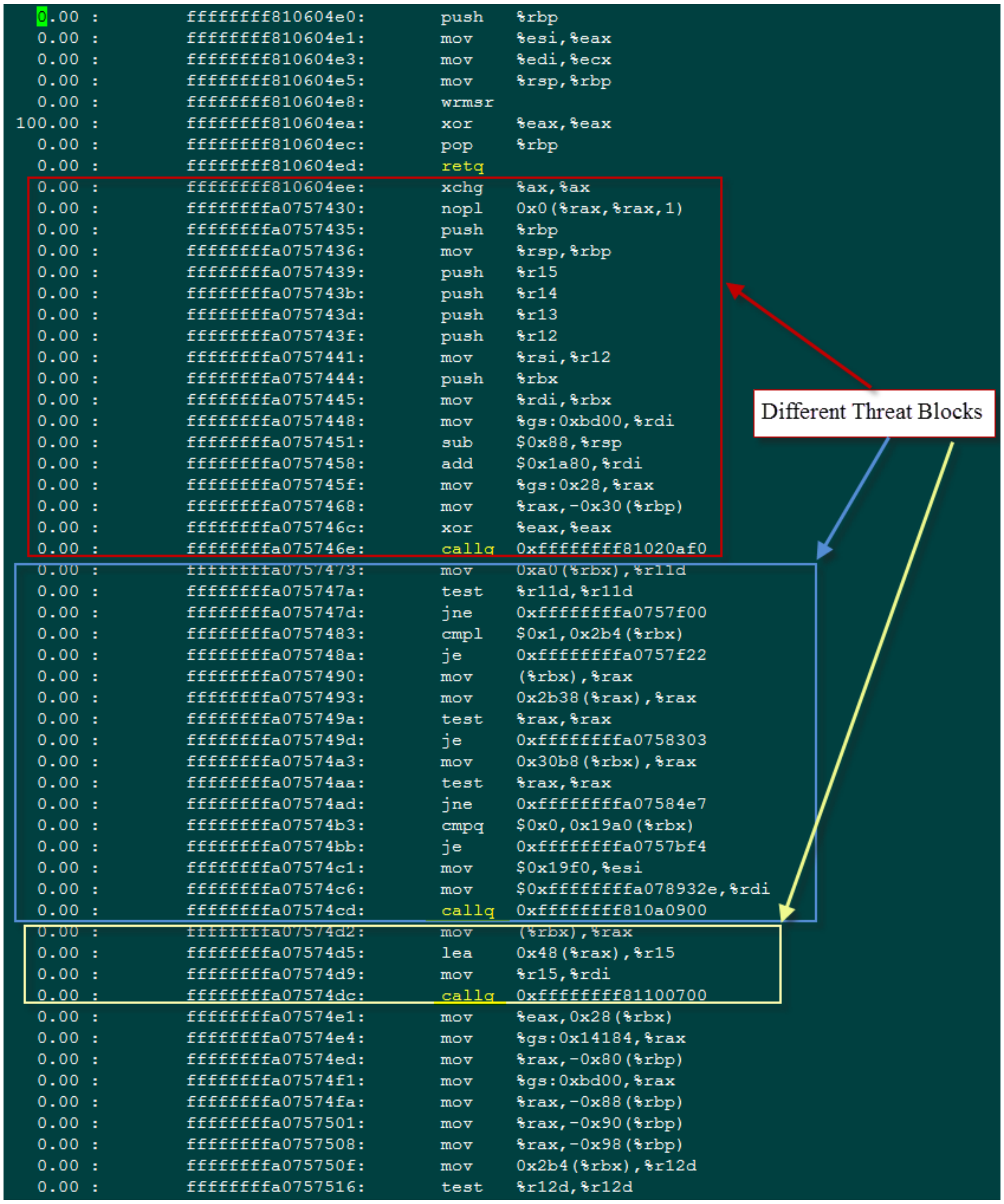
The standardization techniques mentioned in this section allows performing comparative analysis on collected data.

5.5. The Rational

The RECSRF framework can be directly analyzed in line with common software attacks. In order to gain control of the system, an attacker would intercept the program execution somehow and will try to redirect the control flow of the program to his intended zone. The technique has modeled this in terms of a threat block. To compromise the system, the next step of the attacker would be to gain control of the system, which requires the attacker to perform some privilege activities. These privilege activities will be interpreted to the processor in terms of privileged instructions. In a nutshell, the privilege elevation attempts followed by program control transfers will be evaluated by RECSRF number.

There are number of practical advantages of having this evaluation technique in place. This technique can be used in application level intrusion detection and prevention systems, as this measure evaluates a number of practical aspects in a program execution environment. In addition to that, organizations can adopt the proposed RECSRF value to detect malicious code injects and execution redirections in their programs. Additionally the number can be used in security assurance process, which will help in maintaining a consistent execution complexity.

The proposed framework can also be successfully implemented in hardened server environments as well. In most cases different services such as mail, web, and FTP are running separately on different systems under hardened environment. These are security sensitive critical servers, which are running continuously for a long time, leaving no room to take a down time to perform any periodic security related tests. Unfortunately, these are the main targets if of attackers and that is why it is required to run a minimum amount of services on those server boxes. This hardened environment is very much closer the test environment of this research, where a minimum number of background programs are running. Using the RECSRF number along with the number of threat blocks (Figure 5.6) that were there initially, the system administrators can get an idea about the run-time complexity as well as the control flow integrity of the program execution. This is vital in determining whether the system is vulnerable to different attacks.



```
  0.00 :        ffffffff810604e0:       push   %rbp
  0.00 :        ffffffff810604e1:       mov    %esi,%eax
  0.00 :        ffffffff810604e3:       mov    %edi,%ecx
  0.00 :        ffffffff810604e5:       mov    %rsp,%rbp
  0.00 :        ffffffff810604e8:       wrmsr
100.00 :        ffffffff810604ea:       xor    %eax,%eax
  0.00 :        ffffffff810604ec:       pop    %rbp
  0.00 :        ffffffff810604ed:       retq
  0.00 :        ffffffff810604ee:       xchg   %ax,%ax
  0.00 :        ffffffffa0757430:       nopl   0x0(%rax,%rax,1)
  0.00 :        ffffffffa0757435:       push   %rbp
  0.00 :        ffffffffa0757436:       mov    %rsp,%rbp
  0.00 :        ffffffffa0757439:       push   %r15
  0.00 :        ffffffffa075743b:       push   %r14
  0.00 :        ffffffffa075743d:       push   %r13
  0.00 :        ffffffffa075743f:       push   %r12
  0.00 :        ffffffffa0757441:       mov    %rsi,%r12
  0.00 :        ffffffffa0757444:       push   %rbx
  0.00 :        ffffffffa0757445:       mov    %rdi,%rbx
  0.00 :        ffffffffa0757448:       mov    %gs:0xbd00,%rdi
  0.00 :        ffffffffa0757451:       sub    $0x88,%rsp
  0.00 :        ffffffffa0757458:       add    $0x1a80,%rdi
  0.00 :        ffffffffa075745f:       mov    %gs:0x28,%rax
  0.00 :        ffffffffa0757468:       mov    %rax,-0x30(%rbp)
  0.00 :        ffffffffa075746c:       xor    %eax,%eax
  0.00 :        ffffffffa075746e:       callq  0xffffffff81020af0
  0.00 :        ffffffffa0757473:       mov    0xa0(%rbx),%r11d
  0.00 :        ffffffffa075747a:       test   %r11d,%r11d
  0.00 :        ffffffffa075747d:       jne    0xffffffffa0757f00
  0.00 :        ffffffffa0757483:       cmpl   $0x1,0x2b4(%rbx)
  0.00 :        ffffffffa075748a:       je     0xffffffffa0757f22
  0.00 :        ffffffffa0757490:       mov    (%rbx),%rax
  0.00 :        ffffffffa0757493:       mov    0x2b38(%rax),%rax
  0.00 :        ffffffffa075749a:       test   %rax,%rax
  0.00 :        ffffffffa075749d:       je     0xffffffffa0758303
  0.00 :        ffffffffa07574a3:       mov    0x30b8(%rbx),%rax
  0.00 :        ffffffffa07574aa:       test   %rax,%rax
  0.00 :        ffffffffa07574ad:       jne    0xffffffffa07584e7
  0.00 :        ffffffffa07574b3:       cmpq   $0x0,0x19a0(%rbx)
  0.00 :        ffffffffa07574bb:       je     0xffffffffa0757bf4
  0.00 :        ffffffffa07574c1:       mov    $0x19f0,%esi
  0.00 :        ffffffffa07574c6:       mov    $0xffffffffa078932e,%rdi
  0.00 :        ffffffffa07574cd:       callq  0xffffffff810a0900
  0.00 :        ffffffffa07574d2:       mov    (%rbx),%rax
  0.00 :        ffffffffa07574d5:       lea    0x48(%rax),%r15
  0.00 :        ffffffffa07574d9:       mov    %r15,%rdi
  0.00 :        ffffffffa07574dc:       callq  0xffffffff81100700
  0.00 :        ffffffffa07574e1:       mov    %eax,0x28(%rbx)
  0.00 :        ffffffffa07574e4:       mov    %gs:0x14184,%rax
  0.00 :        ffffffffa07574ed:       mov    %rax,-0x80(%rbp)
  0.00 :        ffffffffa07574f1:       mov    %gs:0xbd00,%rax
  0.00 :        ffffffffa07574fa:       mov    %rax,-0x88(%rbp)
  0.00 :        ffffffffa0757501:       mov    %rax,-0x90(%rbp)
  0.00 :        ffffffffa0757508:       mov    %rax,-0x98(%rbp)
  0.00 :        ffffffffa075750f:       mov    0x2b4(%rbx),%r12d
  0.00 :        ffffffffa0757516:       test   %r12d,%r12d
```

Figure 5.6: A capture of real threat blocks

5.6. Summary

This chapter contains the details about the novel evaluation framework proposed by this research called the RECSRF. The granular information flow and the logical reasoning about the problem have laid a solid foundation in further building arguments. It has been identified that privilege escalations and control flow integrity based attacks are powerful attacks on any system and ensuring the control flow integrity has become a paramount.

The chapter has shown how the microprocessor of a system can be vulnerable to control flow integrity based attacks due to the nature of machine instruction sequence it executes. It also shows how a normal user in a system with least amount of privileges can drive microprocessor through risky states allowing a processor level privilege escalation attacks in the system.

The next section of the chapter has shown how to derive the RECSRF number of an execution. In addition to that, possible challenges of the method were also discussed with a data standardization methodology. Finally a justification has been done about the novel framework in the contest of its practical usages and applications.

CHAPTER 6

6. The Evaluation and the Test Results

6.1. Introduction

This chapter contains the results for the tests carried out in this research. Essentially, what RECSRF will evaluate is the risk factor a given program execution imposes on its underlying microprocessor. The number reflects the attack surface or the likelihood of an attack that the microprocessor undergoes due to the program execution. The technique can be used in risk evaluation under different conditions and constrains depending on the user requirement. At software design and baseline evaluation phases, there can be instances which requires answering questions of below format; "what is the best resource combination that provides maximum security?" This can be successfully answered with the RECSRF number.

This chapter provides the results for an evaluation performed in order to get an understanding about the security strength of programs executed as the sudo user and as the root user on a virtualized system. In that context, a given task has been performed using both methods and a comparative risk analysis has been done at the end with the RECSRF number.

The details of the underlying hardware platform used in this evaluation are listed below.

- **Processor:** Intel Core i5 microprocessor
- **Memory:** 8GB
- **Hypervisor:** KVM
- **Host OS:** Fedora 22
- **Guest OS:** RHEL 7

6.2. An Evaluation of Tools

This section provides an overview about the tools that has been considered in this research. Unfortunately, finding a proper tool that purely deals with disassembled machine instructions has become a challenge since, since disassembled code is rarely used in program analysis. However, few tools were closely evaluated in this research and those were derailed below.

- **Objdump utility**

Objdump is a utility in Linux is used to provide thorough information on object files. If an archive is specified, objdump displays information on each object file in the archive. However the tool cannot be used in determining the disassembled code at runtime, the utility has been omitted.

- **Gdb**

GUU debugger is a well know stable utility but it will be really effective when it is debugging a program that has debugging symbols linked in to it. Additionally gdb can only use debugging symbols that are generated by g++. The symbol generation should be performed at the time of compilation, and therefore gdb cannot be used in evaluating basically any program. Therefore gdb has been omitted in this research.

- **Linux Perf**

Perf is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command line interface. Perf is based on the perf_events interface exported by recent versions of the Linux kernel. Perf has been selected in this research mainly because it allows to collect run time disassemble code and also facilitate in creating control flow graphs. The results were obtained in this research with the help of Perf utility.

6.3. Theoretical Basis for the Evaluation

This section summarizes the background details of the aspects that have been taken in to account in this particular evaluation.

6.3.1. SUDO Access

Sudo stands for "super user do, which effectively allows a user to run a program as the root user without sharing the root password. The role of sudo is incredibly important and crucial to many Linux distributions, and is treated as a mechanism to achieve "best practice security" on Linux. When users are given access via sudo, they are prompted to enter their own password, in which upon authentication, the administrative command is executed as if run by the root user. Accessing privileged resources in a system via sudo has many advantages. The utility will come handy especially when it comes to server hardening, where the root user is completely locked. In such situations the privileged resources will be accessed via sudo. Apart from that, there can be situations which require different users to execute different privileged resources in the absence of the root user.

Considering these advantages and most of all due to its highly practical nature in application use, program execution with sudo and root has been taken in to account in this research. Additionally a virtualized system has been considered as the underlying platform, due to its high demand caused by its cost effective nature. In fact, what the research has taken in to account is the microprocessor level risk that privilege elevations will impose on a virtualized system when a given operation is performed either directly as the root user or as a sudo user.

6.3.2. Practicality of the Scenario

In a corporate network, many service oriented servers are running on the DMZ (Demilitarized Zone) on virtual machines (VMs), in which a given service will run solely on a dedicated VM. In that context, as an example a dedicated VM for the mail server, and another dedicated VM for the web server etc.

Due to the server hardening requirements, in most cases, it is required to completely lock down the root account. Sudo access will be the obvious choice in those situations, which allows the access to privileged system resources. Unfortunately a security evaluation will not be performed in such situations, i.e. it will not be checked whether the sudo execution is actually secure than that of the root execution. Detailed below is a RECSRF calculation performed on different activities executed as the sudo user and as the root user.

6.3.3. The Logic

Take H(S, P), where S = {s1, s2,…., sn} is the source alphabet, and P = {p1, p2…pn} is the probability distribution. The set details are listed below.

S = {all unconditional VMExit instructions}

P = {probability of each VMExit instruction}

$$\mathrm{H}(X) = \sum_{i=1}^{n} \mathrm{P}(x_i)\,\mathrm{I}(x_i) = -\sum_{i=1}^{n} \mathrm{P}(x_i)\log_b \mathrm{P}(x_i),$$

In this equation, b is the base of the logarithm used. Common values of *b* are *2*, Euler's number *e*, and *10*, and the unit of entropy is Shannon for b = 2, Nat for b = e, and Hartley for b = 10. When b = 2, the units of entropy are also commonly referred to as bits. The entropy has been calculated for VMExit instructions generated as a result of different program executions.

6.4. Test Results

Below mentioned tests were carried out with this evaluation.

- Modification of privileged log files
- Performing a SUID program – ping command
- Change of system level networking related file
- Tcp dump on an interface
- Modifications to the firewall status of the system
- Change SELinux modes
- Stopping security sensitive daemons

6.4.1. Modification of privileged Logs

System wide log files are a key resource in ensuring the integrity of a system. In that context, the integrity of dmesg utility has been taken in to account. Logs from dmesg are dumped in /var/log/messages, which contains all the system messages including from starting of the system. In this scenario a system-wide log file has been modified, as sudo user and as a root user separately and the risk has been calculated.

6.4.1.1. Program execution as sudo

- Total number of instructions = 20660
- Total number of Unconditional Control Transfer instructions = 2245
- The number of Treat Blocks generated in the execution = (2245+1) -1 = 2245

Table 6.1: Modification of privileged logs – sudo execution

Instruction	Total (fi)	Probability (pi)	Calculation Pi log (pi)
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	2	2/20660	-0.00038858667
INVEPT	0	0	0
INVVPID	0	0	0
VMCALL	0	0	0
VMCLEAR	0	0	0
VMLAUNCH	1	1/20660	-0.000208864
VMPTRLD	0	0	0
VMPTRST	0	0	0
VMREAD	108	108/20660	-0.01192760448
VMRESUME	1	1/20660	-0.000208864
VMWRITE	2	2/20660	-0.00038858667
VMXOFF	0	0	0
VMXON	1	1/20660	-0.000208864

H (SUDO) = - {-0.00038858667-0.000208864-0.01192760448-0.000208864-0.00038858667-0.000208864}

H (SUDO) = **0.01333136982**

6.4.1.2. Program execution as root

- Total Instructions = 36660
- Total number of Unconditional Control Transfer instructions =3743
- The number of Treat Blocks generated in the execution = (3743 +1) -1 = 3743

Table 6.2: Modification of privileged logs – root execution

Instruction	Total	Probability (pi)	Calculation Pi * log (pi)
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	3	3/36660	-0.00033445754
INVEPT	0	0	0
INVVPID	0	0	0
VMCALL	0	0	0
VMCLEAR	0	0	0
VMLAUNCH	1	1/36660	-0.00012450061
VMPTRLD	1	1/36660	-0.00012450061
VMPTRST	0	0	0
VMREAD	101	101/36660	-0.00705256355
VMRESUME	1	1/36660	-0.00012450061
VMWRITE	2	2/36660	0.00023257842
VMXOFF	0	0	0
VMXON	1	1/36660	-0.00012450061

H (ROOT) = - {-0.00033445754-0.00012450061-0.00012450061-0.00705256355-0.00012450061-0.00023257842-0.00012450061}

H (ROOT) =**0.00811760195**

6.4.1.3. Results Analysis

- **Threat Block (TB) normalization**

On sudo $= 2245 \text{ x } \{\frac{(20660)}{(36660+20660)}\}$

$= 809.171318911 \quad \sim 810$

On root $= 3743 \text{ x } \{\frac{(36660)}{(36660+20660)}\}$

$= 2393.90055827 \quad \sim 2394$

- **VMExit Information normalization**

On sudo $= 0.01333136982 \text{ x } \{\frac{(20660)}{(36660+20660)}\}$

$= 0.00480506106$

On root $= 0.00811760195 \text{ x } \{\frac{(36660)}{(36660+20660)}\}$

$= 0.00519175309$

- **RECSRF evaluation**

On sudo = 0.00480506106/810 = **5.932175e-6**

On root = 0.00519175309/2394 = **2.1686521e-6**

The ratio = 5.932175e-6/2.1686521e-6 **= 2.7354**

According to the results, it can be stated that the threat blocks under sudo execution contains more privilege elevations than that of the root execution. There is a significant difference, i.e. the activity via root execution can be treated as if the security risk involved is one third of that of the sudo execution.

6.4.2. A SUID program execution - ping command

Ping is a utility which indicates whether the connections among different computing resources are working correctly. It is used diagnostically to ensure that a host computer the user is trying to reach is actually operating. Ping works by sending an Internet Control Message Protocol (ICMP) Echo Request to a specified interface on the network and waiting for a reply. Ping can be used for troubleshooting to test connectivity and determine response time. It is interesting to note that ping is a SUID program, i.e. it can be executed even by less privileged users with root privileges. Due to this notable capability in UNIX, SUID has become an important feature that will be used in security sensitive operations. In this scenario, the ping to local host command has been issued as a sudo user and as the root user.

6.4.2.1. Program execution as sudo

- Total Instructions = 24073
- Total number of Unconditional Control Transfer instructions = 2735
- The number of Treat Blocks generated in the execution = (2735 + 1) -1 = 2735

Table 6.3: Ping command execution as a sudo user

Instruction	Total (fi)	Probability (pi)	Calculation Pi*log (pi)
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	2	2/24073	-0.00033901052
INVEPT	2	2/24073	-0.00033901052
INVVPID	2	2/24073	-0.00033901052
VMCALL	0	0	0
VMCLEAR	0	0	0
VMLAUNCH	1	1/24073	-0.00018201014
VMPTRLD	1	1/24073	-0.00018201014
VMPTRST	0	0	0
VMREAD	107	107/24073	-0.01045485269
VMRESUME	1	1/24073	-0.00018201014
VMWRITE	2	2/24073	-0.00033901052

VMXOFF	0	0	0
VMXON	1	1/24073	-0.00018201014

H (SUDO) = - {-0.00033901052-0.00033901052-0.00033901052-0.00018201014-0.00018201014 -0.01045485269-0.00018201014-0.00033901052-0.00018201014}

H (SUDO) = **0.01217491505**

6.4.2.2. Program execution as root

- Total Instructions = 23923
- Total number of Unconditional Control Transfer instructions = 2769
- The number of Treat Blocks generated in the execution = (2769 + 1) – 1 = 2769

Table 6.4: Ping command execution as the root user

Instruction	Total (fi)	Probability (pi)	Calculation Pi log (pi)
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	2	2/23923	-0.00034090922
INVEPT	0	0	0
INVVPID	0	0	0
VMCALL	0	0	0
VMCLEAR	0	0	0
VMLAUNCH	1	1/23923	-0.00018303789
VMPTRLD	0	0	0
VMPTRST	0	0	0
VMREAD	100	100/23923	-0.00994363432

VMRESUME	1	1/23923	-0.00018303789
VMWRITE	2	2/23923	-0.00034090922
VMXOFF	0	0	0
VMXON	0	0	0

H (ROOT) = - {-0.00034090922-0.00018303789-0.00994363432-0.00018303789-0.00034090922}

H (ROOT) = **0.01099152854**

6.4.2.3. Results Analysis

- **Threat Block (TB) standardization**

On sudo = 2735 x $\{\frac{24073}{(24073+23923)}\}$

= 1371.77379362 ~ 1370

On root = 2769 x $\{\frac{23923}{(24073+23923)}\}$

= 1380.17307692 ~1380

- **VMExit Information standardization**

On sudo = 0.01217491505 x $\{\frac{24073}{(24073+23923)}\}$

= 0.0061064824150613136605 = 6.12x10^{-3}

On root = 0.01099152854 x $\{\frac{23923}{(24073+23923)}\}$

= 0.00547858857 = 5.479x10^{-3}

- **RECSRF evaluation**

On sudo = 6.12x10^{-3}/1370 = 4.4573e-6

On root = 5.479x10^{-3}/1380= 3.97e-6

The ratio = 4.4573e-6/3.97e-6= **1.1227**

The test results have shown that there is not much deviation on RECSRF values when the activity is performed either via root or as sudo. The difference here is insignificant and this means that both the executions are equally secure.

6.4.3. Change of System Level Networking related file

Under this activity the /etc/resolve.conf file, which a critical file in DNS operations, has been modified as the sudo user as well as the root user.

6.4.3.1. Command execution as sudo

- Total Instructions = 46597
- Total number of Unconditional Control Transfer instructions = 5530
- The number of Treat Blocks generated in the execution = (5530 +1) -1 = 5530

Table 6.5: Modification of privileged network file as sudo

Instruction	Total (fi)	Probability (pi)	Calculation Pi log (pi)
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	2	2/46597	-0.00018745103
INVEPT	0	0	0
INVVPID	0	0	0
VMCALL	0	0	0
VMCLEAR	0	0	0
VMLAUNCH	1	1/46597	-0.0001001858
VMPTRLD	0	0	0
VMPTRST	0	0	0
VMREAD	107	107/46597	-0.00605983726
VMRESUME	1	1/46597	-0.0001001858

VMWRITE	2	2/46597	-0.00018745103
VMXOFF	0	0	0
VMXON	1	1/46597	-0.0001001858

H (SUDO) = - {-0.00018745103-0.0001001858-0.00605983726-0.0001001858-0.00018745103-0.00018745103}

H (SUDO) = **0.00682256195**

6.4.3.2. Command execution as root

- Total Instructions = 35604
- Total number of Unconditional Control Transfer instructions = 3711
- The number of Treat Blocks generated in the execution = (3711 + 1) – 1= 3711

Table 6.6: Modification of privileged network file as root

Instruction	**Total (fi)**	**Probability (pi)**	**Calculation Pi log (pi)**
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	3	3/35604	-0.00034330784
INVEPT	0	0	0
INVVPID	0	0	0
VMCALL	0	0	0
VMCLEAR	0	0	0
VMLAUNCH	1	1/35604	-0.00012783672
VMPTRLD	0	0	0
VMPTRST	0	0	0
VMREAD	109	109/35604	-0.0076967161

VMRESUME	1	1/35604	-0.00012783672
VMWRITE	2	2/35604	-0.00023876355
VMXOFF	0	0	0
VMXON	1	1/35604	-0.00012783672

H (ROOT) = - {-0.00034330784-0.00012783672-0.0076967161-0.00012783672-0.00023876355-0.00012783672}

H (ROOT) = **0.00866229765**

6.4.3.3. Results Analysis

- **Threat Block (TB) standardization**

On sudo = 5530 x $\{\frac{46597}{(46597+35604)}\}$

= 3134.77220471 ~ 3134

On root = 3711 x $\{\frac{35604}{(46597+35604)}\}$

= 1607.35810997 ~ 1607

- **VMExit Information standardization**

On sudo = 0.00682256195 x $\{\frac{46597}{(46597+35604)}\}$

= 0.00386748238

On root = 0.00866229765 x $\{\frac{35604}{(46597+35604)}\}$

= 0.00375193057

- **RECSRF evaluation**

On sudo = 0.00386748238/3134 = **1.2340e-6**

On root = 0.00375193057/1607= **2.3347e-6**

The ratio = 1.2340e-6/2.3347e-6 **= 0.53**

In this scenario the results have shown that the threat block on root execution contains more privilege gains than that of the sudo execution (almost twice). With that, it can be concluded that the more secure operation is sudo execution in this case.

6.4.4. TCP dump on an Interface

The tcpdump command is a powerful and widely used command-line packets sniffer used in packet analysis which is used to capture or filter TCP/IP packets that received or transferred over a network on a specific interface the utility is available under most of the Linux/Unix based operating systems. In this scenario, tcpdump command has been issued for the Ethernet interface and captured packets separately as sudo user and as the root user.

6.4.4.1. Command execution as sudo

- Total Instructions = 22262
- Total number of Unconditional Control Transfer instructions = 2577
- The number of Treat Blocks generated in the execution = (2577 + 1) = 2577

Table 6.7: Execution of TCP Dump command as sudo

Instruction	Total (fi)	Probability (pi)	Calculation Pi log (pi)
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	2	2/22262	-0.00036353734
INVEPT	0	0	0
INVVPID	0	0	0
VMCALL	0	0	0
VMCLEAR	0	0	0
VMLAUNCH	1	1/22262	-0.00019529081
VMPTRLD	0	0	0

VMPTRST	0	0	0
VMREAD	106	106/22262	-0.01105737944
VMRESUME	1	1/22262	-0.00019529081
VMWRITE	2	2/22262	-0.00036353734
VMXOFF	0	0	0
VMXON	1	1/22262	-0.00019529081

H (SUDO) = - {- 0.00036353734 - 0.00019529081 -0.01105737944 - 0.00019529081-0.00036353734 - 0.00019529081}

H (SUDO) = (2x0.00036353734) + (3x0.00019529081) + 0.01105737944

H (SUDO) = **0.01237032655**

6.4.4.2. Command execution as root

- Total Instructions = 20377
- Total number of Unconditional Control Transfer instructions= 2272
- The number of Treat Blocks generated in the execution = (2272 + 1) = 2272

Table 6.8: Execution of TCP Dump command as root

Instruction	**Total (fi)**	**Probability (pi)**	**Calculation Pi log (pi)**
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	2	2/20377	-0.00039339551
INVEPT	2	2/20377	-0.00039339551
INVVPID	2	2/20377	-0.00039339551
VMCALL	0	0	0
VMCLEAR	0	0	0

VMLAUNCH	1	1/20377	-0.00021147078
VMPTRLD	0	0	0
VMPTRST	0	0	0
VMREAD	100	100/20377	-0.0113320913
VMRESUME	1	1/20377	-0.00021147078
VMWRITE	2	2/20377	-0.00039339551
VMXOFF	0	0	0
VMXON	1	1/20377	-0.00021147078

H (ROOT) = - {-0.00039339551- 0.00039339551-0.00039339551-0.00021147078-0.0113320913-0.00021147078-0.00039339551-0.00021147078}

H (ROOT) = (4x0.00039339551) + (3x0.00021147078) + 0.0113320913

H (ROOT) = **0.01354008568**

6.4.4.3. Results Analysis

- **Threat Block (TB) standardization**

On sudo $= 2577 \text{ x } \{\frac{22262}{(22262+20377)}\}$

$= 1345.46246 \qquad \sim 1345$

On root $= 2272 \text{ x } \{\frac{20377}{(22262+20377)}\}$

$= 1085.7793 \qquad \sim 1085$

- **VMExit Information standardization**

On sudo $= 0.01237032655 \text{ x } \{\frac{22262}{(22262+20377)}\}$

$= 0.006558599161708764276835760688 57$

On root $= 0.01354008568 \text{ x } \{\frac{20377}{(22262+20377)}\}$

$= 0.0064707503905194774736743357020 6$

- **RECSRF evaluation**

  On sudo = 0.00645859916170876427683576068857/1345= **4.8019e-6**

  On root = 0.00647075039051947747367433570206/1085= **5.9638e-6**

  The ration = 4.8019e-6/5.9638e-6**= 0.8052**

The results have shown that threat block generated via root execution contains more privileged gains than that of the sudo execution. However there is no significant difference in between the two executions, and therefore it can be stated that both sections are on the same range.

6.4.5. Modifications to the Firewall Status of the System

A firewall is either hardware or software-based and controls incoming and outgoing network traffic based on a set of pre-defined rules. In this scenario the software based system firewall on the host system is disabled and enabled. The VMExit instruction distribution for this activity is as follows.

6.4.5.1. Command execution as sudo

- Total Instructions = 31276
- Total number of Unconditional Control Transfer instructions = 3631
- The number of Treat Blocks generated in the execution = (3631 + 1) – 1 = 3631

Table 6.9: Firewall modification as sudo

Instruction	Total (fi)	Probability (pi)	Calculation Pi log (pi)
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	3	3/31276	-0.00038541596
INVEPT	0	0	s0
INVVPID	0	0	0

VMCALL	0	0	0
VMCLEAR	0	0	0
VMLAUNCH	1	1/31276	-0.00014372717
VMPTRLD	0	0	0
VMPTRST	0	0	0
VMREAD	108	108/31276	-0.0085008647
VMRESUME	1	1/31276	-0.00014372717
VMWRITE	2	2/312761	-0.00026820445
VMXOFF	0	0	0
VMXON	1	1/31276	-0.00014372717

H (SUDO) = - {-0.00038541596-0.00014372717-0.0085008647-
0.00014372717-0.00026820445-0.00014372717}

H (SUDO) = 0.00038541596 + (3x0.00014372717) + 0.00026820445 +
0.0085008647

H (SUDO) = **0.00958566662**

6.4.5.2. Command execution as root

- Total Instructions = 25871
- Total number of Unconditional Control Transfer instructions= 3017
- The number of Treat Blocks generated in the execution = (3017 + 1) – 1 = 3017

Table 6.10: Firewall modification as sudo

Instruction	**Total (fi)**	**Probability (pi)**	**Calculation Pi log (pi)**
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	2	2/25871	-0.00031786813

INVEPT	0	0	0
INVVPID	0	0	0
VMCALL	0	0	0
VMCLEAR	0	0	0
VMLAUNCH	1	1/25871	-0.00017056987
VMPTRLD	0	0	0
VMPTRST	0	0	0
VMREAD	103	103/25871	-0.00955500471
VMRESUME	1	1/25871	-0.00017056987
VMWRITE	2	2/25871	-0.00031786813
VMXOFF	0	0	0
VMXON	0	0	0

H (ROOT) = - {-0.00031786813-0.00017056987-0.00955500471-0.00017056987-0.00031786813}

H (ROOT) = (2 x 0.00031786813) + (2 x 0.00017056987) + 0.00955500471

H (ROOT) = **0.01053188071**

6.4.5.3. Results Analysis

- **Threat Block (TB) standardization**

On sudo $= 3631 \times \{\frac{31276}{(31276+25871)}\}$

$= 1987.2112$ ~ 1987

On root $= 3017 \times \{\frac{25871}{(31276+25871)}\}$

$= 1365.8251$ ~ 1365

- **VMExit Information standardization**

On sudo $= 0.00958566662 \times \{\frac{31276}{(31276+25871)}\}$

= 0.00524614256578858032792622534866

On root = 0.01053188071 x $\{\frac{25871}{(31276+25871)}\}$

= 0.00476788433073319684322886590722

- **RECSRF evaluation**

  On sudo = 0.00524614256578858032792622534866/1987= **2.640e-6**

  On root = 0.00476788433073319684322886590722/1365= **3.4930e-6**

  The ratio = 2.640e-6/3.4930e-6**= 0.756**

The obtained results have shown that the average privilege information distribution on a threat block on root execution is higher than that of the sudo execution, and therefore it can be stated that sudo execution is more secure the root execution in this scenario.

6.4.6. Change SELinux Modes

SELinux is a security enhancement to Linux which allows users and administrators more control over access control. It provides a mechanism to enforce the separation of information based on confidentiality and integrity requirements. This allows threats of tampering and bypassing of application security mechanisms to be addressed and enables the confinement of damage that can be caused by malicious or flawed applications. In this scenario, the SELinux mode has been changed on runtime as the sudo user and as a root user separately.

6.4.6.1. Command execution as sudo

- Total Instructions = 18402
- Total number of Unconditional Control Transfer instructions = 2137
- The number of Treat Blocks generated in the execution =(2137 + 1) – 1 = 2137

Table 6.11: SELinux mode change as sudo

Instruction	Total (fi)	Probability (pi)	Calculation Pi log (pi)
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	2	2/18402	-0.0004308048
INVEPT	2	2/18402	-0.0004308048
INVVPID	2	2/18402	-0.0004308048
VMCALL	0	0	0
VMCLEAR	0	0	0
VMLAUNCH	1	1/18402	-0.00023176095
VMPTRLD	0	0	0
VMPTRST	0	0	0
VMREAD	105	105/18402	-0.01280219277
VMRESUME	1	1/18402	-0.00023176095
VMWRITE	2	2/18402	-0.0004308048
VMXOFF	0	0	0
VMXON	1	1/18402	-0.00023176095

H (SUDO) = - {-0.0004308048-0.0004308048-0.0004308048-0.00023176095-0.01280219277-0.00023176095-0.0004308048-0.00023176095}

H (SUDO) = (4x0.0004308048) + (3x0.00023176095) + 0.01280219277

H (SUDO) = **0.01522069482**

6.4.6.2. Command Execution as Root

- Total Instructions = 14909
- Total number of Unconditional Control Transfer instructions = 1707

- The number of Treat Blocks generated in the execution = (1707 + 1) – 1 = 1707

Table 6.12: SELinux mode change as root

Instruction	**Total (fi)**	**Probability (pi)**	**Calculation** **Pi log (pi)**
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	3	3/14909	-0.0007437777
INVEPT	0	0	0
INVVPID	0	0	0
VMCALL	0	0	0
VMCLEAR	0		0
VMLAUNCH	1	1/14909	-0.00027992813
VMPTRLD	0	0	0
VMPTRST	0	0	0
VMREAD	106	106/14909	-0.01527286342
VMRESUME	1	1/14909	-0.00027992813
VMWRITE	2	2/14909	-0.00051947394
VMXOFF	0	0	0
VMXON	1	1/14909	-0.00027992813

H (ROOT) = - {-0.0007437777-0.00027992813-0.01527286342-0.00027992813-0.00051947394-0.00027992813}

H (ROOT) = 0.0007437777 + (3x0.00027992813) + 0.00051947394 + 0.01527286342

H (ROOT) = **0.01737589945**

6.4.6.3. Results Analysis

- **Threat Block (TB) standardization**

On sudo = 2137 x $\{\frac{18402}{(18402+14909)}\}$

= 1180.5432 ~ 1180

On root = 1707 x $\{\frac{14909}{(18402+14909)}\}$

= 764.002 ~ 764

- **VMExit Information standardization**

On sudo = 0.01522069482 x $\{\frac{18402}{(18402+14909)}\}$

= 0.00840837039049082885533307315902

On root = 0.01737589945 x $\{\frac{14909}{(18402+14909)}\}$

= 0.00777692908949145927771606976674

- **RECSRF evaluation**

On sudo = 0.00840837039049082885533307315902/1180= **7.1257e-6**

On root = 0.00777692908949145927771606976674/764= **1.0179e-5**

The ration = 7.1257e-6/1.0179e-5**= 0.7000**

According to the results obtained, in this scenario the privilege information density on sudo threat block is lesser than that of the root threat block. Therefore, sudo execution can be treated as secure in this scenario.

6.4.7. Stopping Security Sensitive Daemons

In this scenario, the audit daemon of the system has been stopped and started separately as the sudo user and as the root user.

6.4.7.1. Command execution as sudo

- Total Instructions = 25874
- Total number of Unconditional Control Transfer instructions = 2972
- The number of Treat Blocks generated in the execution = (2972 + 1) – 1 = 2972

Table 6.13: Stopping the system wide audit daemon as sudo

Instruction	Total (fi)	Probability (pi)	Calculation {Pi*log (pi)}
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	2	2/25874	-0.00031783516
INVEPT	0	0	0
INVVPID	0	0	0
VMCALL	0	0	0
VMCLEAR	0	0	0
VMLAUNCH	1	1/25874	-0.00017055204
VMPTRLD	1	1/25874	-0.00017055204
VMPTRST	0	0	0
VMREAD	106	106/25874	-0.00978129075
VMRESUME	1	1/25874	-0.00017055204
VMWRITE	1	1/25874	-0.00017055204
VMXOFF	0	0	0
VMXON	1	1/25874	-0.00017055204

H (ROOT) = - {-0.00031783516-0.00017055204-0.00017055204-0.00978129075-0.00017055204-0.00017055204-0.00017055204}

H (ROOT) = 0.00031783516 + (5x0.00017055204) + 0.00978129075

H (ROOT) = **0.01095188611**

6.4.7.2. Command execution as root

- Total Instructions = 31084
- Total number of Unconditional Control Transfer instructions = 3953
- The number of Treat Blocks generated in the execution = (3953 + 1) -1 = 3953

Table 6.14: Stopping the system wide audit daemon as root

Instruction	Total (fi)	Probability (pi)	Calculation Pi log (pi)
CPUID	0	0	0
GETSEC	0	0	0
INVD	0	0	0
XSETBV	2	2/31084	-0.00026968903
INVEPT	0	0	0
INVVPID	0	0	0
VMCALL	0	0	0
VMCLEAR	0	0	0
VMLAUNCH	1	1/31084	-0.00014452891
VMPTRLD	0	0	0
VMPTRST	0	0	0
VMREAD	115	115/31084	-0.00899695958
VMRESUME	1	1/31084	-0.00014452891
VMWRITE	2	2/31084	-0.00026968903
VMXOFF	0	0	0
VMXON	1	1/31084	-0.00014452891

H (ROOT) = - {-0.00026968903-0.00014452891-0.00899695958-0.00014452891-0.00026968903-0.00014452891}

H (ROOT) = (2x0.00026968903) + (3x0.00014452891) + 0.00899695958

H (ROOT) = **0.00996992437**

6.4.7.3. Results Analysis

- **Threat Block (TB) standardization**

On sudo = 2972 x $\{\frac{25874}{(25874+31084)}\}$

= 1350.0742 ~ 1350

On root = 3953 x $\{\frac{31084}{(18402+31084)}\}$

= 2483.0266 ~ 2483

- **VMExit Information standardization**

On sudo = 0.01095188611 x $\{\frac{25874}{(25874+31084)}\}$

= 0.0049750535694747006566241792197 8

On root = 0.00996992437 x $\{\frac{31084}{(18402+31084)}\}$

= 0.00626248088584811865982298023683

- **RECSRF evaluation**

On sudo = 0.00497505356947470065662417921978/1350= **3.6852e-6**

On root = 0.00626248088584811865982298023683/2483= **2.5221e-6**

The ratio = 3.6852e-6/2.5221e-6**= 1.4612**

The test results have shown that privileged information density on root execution is lesser than that of the sudo execution.

6.5. The Evaluation of Test Results

The RECSRF ratio can be used along with a scale. This scale can be determined according to the need of the requirement. In order to compare the results, below scale has been used.

Take $X = \frac{\text{RECSRF(Sudo)}}{\text{RECSRF(Root)}}$

Table 8.15 shows the scale that has been adopted in this evaluation.

Table 6.15: RECSRF ratio evaluation scheme

Ratio	Risk	Note
X < 0.25	LOW	Both executions can be treated as equally secure; hence either method can be used in this scenario.
0.25 <= X < 0.75	MEDIUM	The risk factor ratio is moderate. However the execution with lower RECSRF number is preferred to be selected.
X >= 0.75	HIGH	There is a significant difference in the risk factor involved on two executions. Therefore it is highly recommended to select the execution with the lower RECSRF number.

The evaluation scheme given above is an example of a custom made scale. It essentially provides a base reference in quantifying the risk. The scale can be customized depending on the requirement.

The results obtained during the test have been summarized in Table 6.16 below in line with an evaluation according to Table 6.15 above.

Table 6.16: The Evaluation of Results

Test	RECSRF Ratio (sudo/root)	Risk Comparison	Comment
Modification of privileged logs	2.7354	HIGH	The results have shown that the per threat block privileged information density on root execution is almost 250% times compared to the sudo execution. Therefore the execution via root is the secure method.
SUID program execution (ping request)	1.1227	HIGH	The root execution is almost 100% times secure than the sudo execution. Hence the root execution is the recommended method.
Change of DNS config file	0.5300	MEDIUM	The ratio indicates that the per threat block privileged information density on the root execution is almost twice compared to that of the sudo execution. Therefore the sudo execution is preferred in this scenario.
Network analysis with tcpdump utility	0.8052	HIGH	The results obtained have shown that there exists high privilege density on a root threat block. Hence the sudo execution is more secure than the root execution.
Modification of software firewall	0.7560	HIGH	The sudo execution is more secure than the root execution.
Change of SELinux modes	0.7000	MEDIUM	The sudo execution is more secure than the root execution.
Stopping the audit daemon	1.4612	HIGH	The per-threat block privileged information density on sudo execution is higher than that of

			the root execution. Therefore it can be stated that the execution via root is more secure than the sudo execution.

Generally, the execution via sudo is treated as secure, since it does not require root password to execute programs. However the analysis has shown that there is a significant difference in risk factors involved, when the same activity is performed as sudo and as root. In some situations, the per-threat block privileged information density in sudo is significantly higher, compared to that of the root execution; which makes the sudo execution more susceptible to privilege escalation attacks. The evaluation carried out in this research has taken sudo vs. root execution. However, the RECSRF framework allows the other operations also to be evaluated comparatively and a quantitatively in order to get an understanding about different execution methods.

6.6. Other Observations

Apart from the main objective of the research; the execution evaluations via RECSRF, this section summarizes the other observations made during the research.

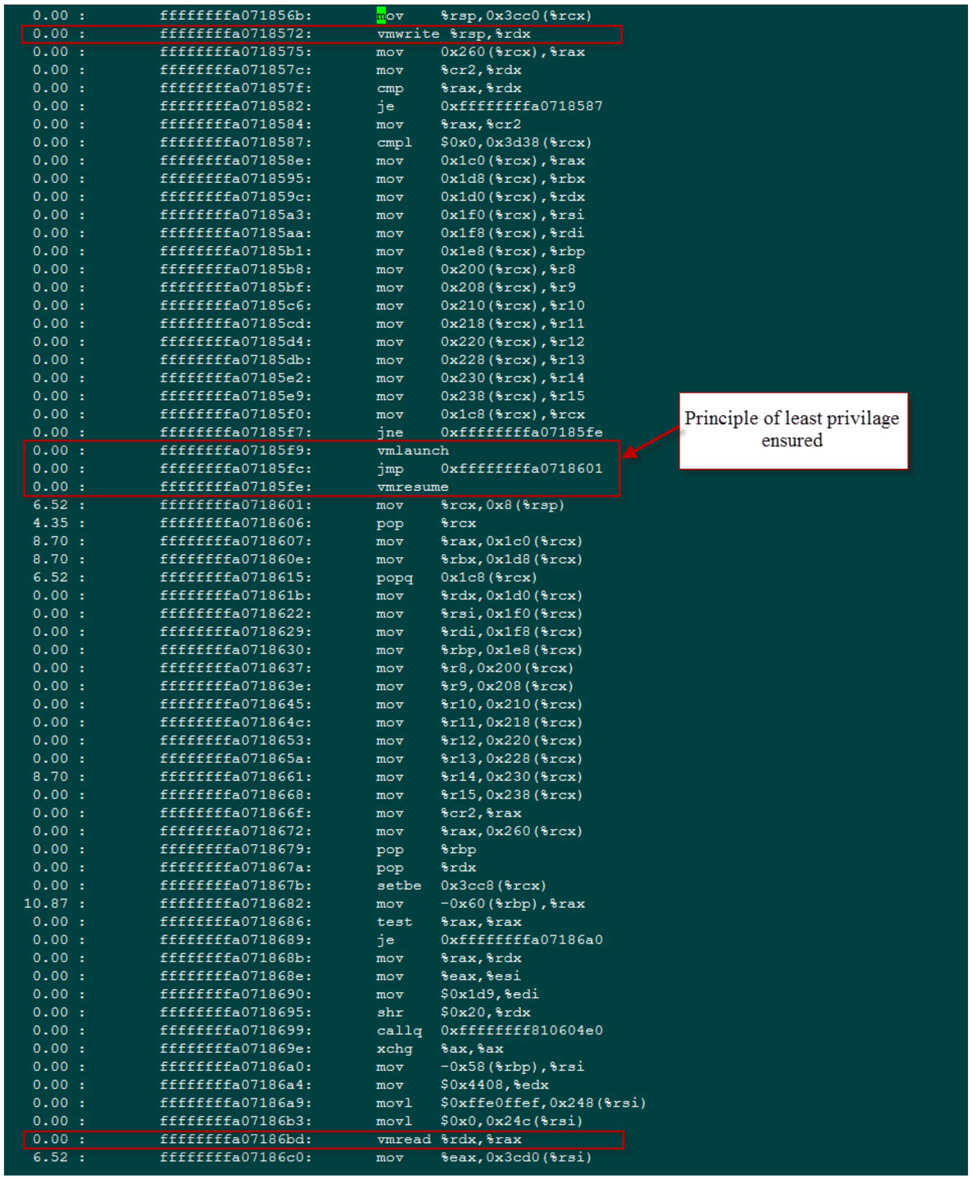
6.6.1. Observations related to VMX Instructions

The task of the VMLAUNCH is to start the VM pointed to by the loaded VMCS, whereas unloading the VM loaded by the VMCS is the task of the VMRESUME instruction. During the data collection and analysis phase of the research, it has been observed that the KVM hypervisor carefully handles VMX extensions provided by Intel-VT. The Figure 8.1 is a real disassembled instruction trace, and given below is an extract.

```
ffffffffa07185f7:     jne     0xffffffffa07185fe

ffffffffa07185f9:     vmlaunch

ffffffffa07185fc:     jmp     0xffffffffa0718601

ffffffffa07185fe:     vmresume

ffffffffa0718601:     mov     %rcx,%0x8(%rsp)
```

The disassembled code has shown that VMLAUNCH will be executed if and only of the JNE command is unsuccessful. The JNE instruction is directly pointing to the memory location of the VMRESUME instruction (0xffffffffa07185fe), which means, if the inequality is satisfied, it will directly execute the VMRESUME instruction.

On the other hand, after executing VMLAUNCH (which is a privilege reduction operation) a JMP instruction occurred and it directly points to the MOV instruction at ffffffffa0718601. The region in between VMLAUNCH and VMRESUME is a critical region because the microprocessor will perform a VMX transition along with these instructions. With VMLAUNCH a microprocessor level privilege switching will take place, i.e. the processor state will be changed from VMX-non-root mode to the VMX-root mode, causing a privilege reduction. However, the VMRESUME is the exact opposite of this, i.e. a VMX-non-root to VMX-root mode change will occur in that case causing a microprocessor level privilege elevation. Apart from that, interestingly the branch instruction in between VMLAUNCH and VMRESUME is an unconditional control transfer instruction (JMP), which means that there is no room for conditional executions. This has dramatically bought down the probability of control flow hijacking attacks via privilege elevations.



```
  0.00 :        ffffffffa071856b:       mov    %rsp,0x3cc0(%rcx)
  0.00 :        ffffffffa0718572:       vmwrite %rsp,%rdx
  0.00 :        ffffffffa0718575:       mov    0x260(%rcx),%rax
  0.00 :        ffffffffa071857c:       mov    %cr2,%rdx
  0.00 :        ffffffffa071857f:       cmp    %rax,%rdx
  0.00 :        ffffffffa0718582:       je     0xffffffffa0718587
  0.00 :        ffffffffa0718584:       mov    %rax,%cr2
  0.00 :        ffffffffa0718587:       cmpl   $0x0,0x3d38(%rcx)
  0.00 :        ffffffffa071858e:       mov    0x1c0(%rcx),%rax
  0.00 :        ffffffffa0718595:       mov    0x1d8(%rcx),%rbx
  0.00 :        ffffffffa071859c:       mov    0x1d0(%rcx),%rdx
  0.00 :        ffffffffa07185a3:       mov    0x1f0(%rcx),%rsi
  0.00 :        ffffffffa07185aa:       mov    0x1f8(%rcx),%rdi
  0.00 :        ffffffffa07185b1:       mov    0x1e8(%rcx),%rbp
  0.00 :        ffffffffa07185b8:       mov    0x200(%rcx),%r8
  0.00 :        ffffffffa07185bf:       mov    0x208(%rcx),%r9
  0.00 :        ffffffffa07185c6:       mov    0x210(%rcx),%r10
  0.00 :        ffffffffa07185cd:       mov    0x218(%rcx),%r11
  0.00 :        ffffffffa07185d4:       mov    0x220(%rcx),%r12
  0.00 :        ffffffffa07185db:       mov    0x228(%rcx),%r13
  0.00 :        ffffffffa07185e2:       mov    0x230(%rcx),%r14
  0.00 :        ffffffffa07185e9:       mov    0x238(%rcx),%r15
  0.00 :        ffffffffa07185f0:       mov    0x1c8(%rcx),%rcx
  0.00 :        ffffffffa07185f7:       jne    0xffffffffa07185fe
  0.00 :        ffffffffa07185f9:       vmlaunch
  0.00 :        ffffffffa07185fc:       jmp    0xffffffffa0718601
  0.00 :        ffffffffa07185fe:       vmresume
  6.52 :        ffffffffa0718601:       mov    %rcx,0x8(%rsp)
  4.35 :        ffffffffa0718606:       pop    %rcx
  8.70 :        ffffffffa0718607:       mov    %rax,0x1c0(%rcx)
  8.70 :        ffffffffa071860e:       mov    %rbx,0x1d8(%rcx)
  6.52 :        ffffffffa0718615:       popq   0x1c8(%rcx)
  0.00 :        ffffffffa071861b:       mov    %rdx,0x1d0(%rcx)
  0.00 :        ffffffffa0718622:       mov    %rsi,0x1f0(%rcx)
  0.00 :        ffffffffa0718629:       mov    %rdi,0x1f8(%rcx)
  0.00 :        ffffffffa0718630:       mov    %rbp,0x1e8(%rcx)
  0.00 :        ffffffffa0718637:       mov    %r8,0x200(%rcx)
  0.00 :        ffffffffa071863e:       mov    %r9,0x208(%rcx)
  0.00 :        ffffffffa0718645:       mov    %r10,0x210(%rcx)
  0.00 :        ffffffffa071864c:       mov    %r11,0x218(%rcx)
  0.00 :        ffffffffa0718653:       mov    %r12,0x220(%rcx)
  0.00 :        ffffffffa071865a:       mov    %r13,0x228(%rcx)
  8.70 :        ffffffffa0718661:       mov    %r14,0x230(%rcx)
  0.00 :        ffffffffa0718668:       mov    %r15,0x238(%rcx)
  0.00 :        ffffffffa071866f:       mov    %cr2,%rax
  0.00 :        ffffffffa0718672:       mov    %rax,0x260(%rcx)
  0.00 :        ffffffffa0718679:       pop    %rbp
  0.00 :        ffffffffa071867a:       pop    %rdx
  0.00 :        ffffffffa071867b:       setbe  0x3cc8(%rcx)
 10.87 :        ffffffffa0718682:       mov    -0x60(%rbp),%rax
  0.00 :        ffffffffa0718686:       test   %rax,%rax
  0.00 :        ffffffffa0718689:       je     0xffffffffa07186a0
  0.00 :        ffffffffa071868b:       mov    %rax,%rdx
  0.00 :        ffffffffa071868e:       mov    %eax,%esi
  0.00 :        ffffffffa0718690:       mov    $0x1d9,%edi
  0.00 :        ffffffffa0718695:       shr    $0x20,%rdx
  0.00 :        ffffffffa0718699:       callq  0xffffffff810604e0
  0.00 :        ffffffffa071869e:       xchg   %ax,%ax
  0.00 :        ffffffffa07186a0:       mov    -0x58(%rbp),%rsi
  0.00 :        ffffffffa07186a4:       mov    $0x4408,%edx
  0.00 :        ffffffffa07186a9:       movl   $0xffe0ffef,0x248(%rsi)
  0.00 :        ffffffffa07186b3:       movl   $0x0,0x24c(%rsi)
  0.00 :        ffffffffa07186bd:       vmread %rdx,%rax
  6.52 :        ffffffffa07186c0:       mov    %eax,0x3cd0(%rsi)
```

Figure 6.1: The Principle of Least Privilege Example

6.6.2. Observations related to VMX Instruction Distribution

The test results on section 8.3 have shown that all VMX instructions are not equally distributed. Some instructions were very rarely occurred, and some are frequently occurred. Instructions such as VMLAUNCH, VMRESUME, VMXON and VMXOFF were rarely used whereas VMWRITE was moderately used. VMREAD is the VMX instruction that is most frequently used. In the context of privilege elevations, this is another notable plus point ensured by the KVM hypervisor.

These observations have shown that KVM has adopted Intel-VT extensions in a way such that it will ensure reasonable execution resilience in the context of security. Commands such as VMLAUNCH and VMRESUME can be treated as highly volatile security commands, as those will contribute in VMCS manipulations which introduce a risk. Having a lesser amount of these instructions will help in ensuring system-wide integrity. The most frequently used VMX instruction in these executions; the VMREAD command will only performs read operations on VMCS, which is less volatile, hence it imposes a lower risk.

6.7. Summary

This chapter contains the detailed results of the tests that have been performed using the RECSRF framework. The question that was successfully addressed is in this chapter is "what is the most secure execution when a give task is performed as the sudo user and as the root user?" in that context, a set of security sensitive operations were selected and the same task has been executed as sudo and as root separately and finally a comparative analysis performed using the obtained RECSRF number.

At the end of the section a comprehensive analysis was performed with regard to the obtained results. Finally the chapter concludes with some observations related to program security form disassembled instruction point of view, including some notable feature about the KVM hypervisor and its interaction with Intel-VT.

Appendix

This section summarizes the categorization of Intel x86 machine instructions

Data Transfer Instructions

Instruction	Description
MOV	Move
CMOVE/CMOVZ	Conditional move if equal/Conditional move if zero
CMOVNE/CMOVNZ	Conditional move if not equal/Conditional move if not zero
CMOVA/CMOVNBE	Conditional move if above/Conditional move if not below or equal
CMOVAE/CMOVNB	Conditional move if above or equal/Conditional move if not below
CMOVB/CMOVNAE	Conditional move if below/Conditional move if not above or equal
CMOVBE/CMOVNA	Conditional move if below or equal/Conditional move if not above
CMOVG/CMOVNLE	Conditional move if greater/Conditional move if not less or equal
CMOVGE/CMOVNL	Conditional move if greater or equal/Conditional move if not less
CMOVL/CMOVNGE	Conditional move if less/Conditional move if not greater or equal
CMOVLE/CMOVNG	Conditional move if less or equal/Conditional move if not greater
CMOVC	Conditional move if carry
CMOVNC	Conditional move if not carry
CMOVO	Conditional move if overflow
CMOVNO	Conditional move if not overflow
CMOVS	Conditional move if sign (negative)
CMOVNS	Conditional move if not sign (non-negative)
CMOVP/CMOVPE	Conditional move if parity/Conditional move if parity even
CMOVNP/CMOVPO	Conditional move if not parity/Conditional move if parity odd
XCHG	Exchange
BSWAP	Byte swap
XADD	Exchange and add
CMPXCHG	Compare and exchange
CMPXCHG8B	Compare and exchange 8 bytes
PUSH	Push onto stack
POP	Pop off of stack
PUSHA/PUSHAD	Push general-purpose registers onto stack
POPA/POPAD	Pop general-purpose registers from stack

IN	Read from a port
OUT	Write to a port
CWD/CDQ	Convert word to doubleword/Convert doubleword to quadword
CBW/CWDE	Convert byte to word/Convert word to doubleword in EAX register
MOVSX	Move and sign extend
MOVZX	Move and zero extend

Binary Arithmetic Instructions

Instruction	**Description**
ADD	Integer add
ADC	Add with carry
SUB	Subtract
SBB	Subtract with borrow
IMUL	Signed multiply
MUL	Unsigned multiply
IDIV	Signed divide
DIV	Unsigned divide
INC	Increment
DEC	Decrement
NEG	Negate
CMP	Compare

Decimal Arithmetic

Instruction	**Description**
DAA	Decimal adjust after addition
DAS	Decimal adjust after subtraction
AAA	ASCII adjust after addition
AAS	ASCII adjust after subtraction
AAM	ASCII adjust after multiplication
AAD	ASCII adjust before division

Logic Instructions

Instruction	**Description**
AND	And
OR	Or
XOR	Exclusive or
NOT	Not

Shift and Rotate Instructions

Instruction	**Description**
SAR	Shift arithmetic right
SHR	Shift logical right
SAL/SHL	Shift arithmetic left/Shift logical left
SHRD	Shift right double
SHLD	Shift left double
ROR	Rotate right
ROL	Rotate left
RCR	Rotate through carry right
RCL	Rotate through carry left

Bit and Byte Instructions

Instruction	**Description**
BT	Bit test
BTS	Bit test and set
BTR	Bit test and reset
BTC	Bit test and complement
BSF	Bit scan forward
BSR	Bit scan reverse
SETE/SETZ	Set byte if equal/Set byte if zero
SETNE/SETNZ	Set byte if not equal/Set byte if not zero
SETA/SETNBE	Set byte if above/Set byte if not below or equal
SETAE/SETNB/SETNC	Set byte if above or equal/Set byte if not below/Set byte if not carry

SETB/SETNAE/SETC	Set byte if below/Set byte if not above or equal/Set byte if carry
SETBE/SETNA	Set byte if below or equal/Set byte if not above
SETG/SETNLE	Set byte if greater/Set byte if not less or equal
SETGE/SETNL	Set byte if greater or equal/Set byte if not less
SETL/SETNGE	Set byte if less/Set byte if not greater or equal
SETLE/SETNG	Set byte if less or equal/Set byte if not greater
SETS	Set byte if sign (negative)
SETNS	Set byte if not sign (non-negative)
SETO	Set byte if overflow
SETNO	Set byte if not overflow
SETPE/SETP	Set byte if parity even/Set byte if parity
SETPO/SETNP	Set byte if parity odd/Set byte if not parity
TEST	Logical compare

Control Transfer Instructions

Instruction	**Description**
JMP	Jump
JE/JZ	Jump if equal/Jump if zero
JNE/JNZ	Jump if not equal/Jump if not zero
JA/JNBE	Jump if above/Jump if not below or equal
JAE/JNB	Jump if above or equal/Jump if not below
JB/JNAE	Jump if below/Jump if not above or equal
JBE/JNA	Jump if below or equal/Jump if not above
JG/JNLE	Jump if greater/Jump if not less or equal
JGE/JNL	Jump if greater or equal/Jump if not less
JL/JNGE	Jump if less/Jump if not greater or equal
JLE/JNG	Jump if less or equal/Jump if not greater
JC	Jump if carry
JNC	Jump if not carry
JO	Jump if overflow
JNO	Jump if not overflow
JS	Jump if sign (negative)

JNS	Jump if not sign (non-negative)
JPO/JNP	Jump if parity odd/Jump if not parity
JPE/JP	Jump if parity even/Jump if parity
JCXZ/JECXZ	Jump register CX zero/Jump register ECX zero
LOOP	Loop with ECX counter
LOOPZ/LOOPE	Loop with ECX and zero/Loop with ECX and equal
LOOPNZ/LOOPNE	Loop with ECX and not zero/Loop with ECX and not equal
CALL	Call procedure
RET	Return
IRET	Return from interrupt
INT	Software interrupt
INTO	Interrupt on overflow
BOUND	Detect value out of range
ENTER	High-level procedure entry
LEAVE	High-level procedure exit

String Instructions

Instruction	**Description**
MOVS/MOVSB	Move string/Move byte string
MOVS/MOVSW	Move string/Move word string
MOVS/MOVSD	Move string/Move double word string
CMPS/CMPSB	Compare string/Compare byte string
CMPS/CMPSW	Compare string/Compare word string
CMPS/CMPSD	Compare string/Compare double word string
SCAS/SCASB	Scan string/Scan byte string
SCAS/SCASW	Scan string/Scan word string
SCAS/SCASD	Scan string/Scan double word string
LODS/LODSB	Load string/Load byte string
LODS/LODSW	Load string/Load word string
LODS/LODSD	Load string/Load double word string
STOS/STOSB	Store string/Store byte string
STOS/STOSW	Store string/Store word string
STOS/STOSD	Store string/Store double word string

REP	Repeat while ECX not zero
REPE/REPZ	Repeat while equal/Repeat while zero
REPNE/REPNZ	Repeat while not equal/Repeat while not zero
INS/INSB	Input string from port/Input byte string from port
INS/INSW	Input string from port/Input word string from port
INS/INSD	Input string from port/Input double word string from port
OUTS/OUTSB	Output string to port/Output byte string to port
OUTS/OUTSW	Output string to port/Output word string to port
OUTS/OUTSD	Output string to port/Output double word string to port

Flag Control Instructions

Instruction	**Description**
STC	Set carry flag
CLC	Clear the carry flag
CMC	Complement the carry flag
CLD	Clear the direction flag
STD	Set direction flag
LAHF	Load flags into AH register
SAHF	Store AH register into flags
PUSHF/PUSHFD	Push EFLAGS onto stack
POPF/POPFD	Pop EFLAGS from stack
STI	Set interrupt flag
CLI	Clear the interrupt flag

Segment Register Instructions

Instruction	**Description**
LDS	Load far pointer using DS
LFS	Load far pointer using FS
LGS	Load far pointer using GS
LSS	Load far pointer using SS

Miscellaneous Instructions

Instruction	**Description**
LEA	Load effective address
NOP	No operation
UB2	Undefined instruction
XLAT/XLATB	Table lookup translation
CPUID	Processor Identification

References

[1] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F., Anderson, A. V., ... & Smith, L. (2005). Intel virtualization technology. Computer, 38(5), 48-56.

[2] McCabe, Thomas J. "A complexity measure." Software Engineering, IEEE Transactions on 4 (1976): 308-320.

[3] E. Berlinger, "An Information Theory Based Complexity Measure",Proceedings of the 1980 National Computer Conference. pp. 773-779.

[4] Atkinson, Colin, and Thomas Kühne. "Reducing accidental complexity in domain models." Software & Systems Modeling 7.3 (2008): 345-359.

[5] Cook, C. "Information theory metric for assembly language." Proceedings of Third Annual Oregon Workshop on Software Metrics. 1991.

[6] Shannon, Claude Elwood. "A mathematical theory of communication." ACM SIGMOBILE Mobile Computing and Communications Review 5.1 (2001): 3-55.

[7] Oh, N., Shirvani, P. P., & McCluskey, E. J. (2002). Control-flow checking by software signatures. Reliability, IEEE Transactions on, 51(1), 111-122.

[8] Bardas, A. G. (2010). Static code analysis. Journal of Information Systems & Operations Management, 4(2), 99-107.

[9] Hsi, C. H., Bresee, R. R., & Annis, P. A. (2000). Characterizing fuzz on fabrics using image analysis. Textile Research Journal, 70(10), 859-865.

[10] Eder, J., Kappel, G., & Schrefl, M. (1994). Coupling and cohesion in object-oriented systems. Technical Reprot, University of Klagenfurt, Austria.

[11] Shin, Y., & Williams, L. (2008, October). Is complexity really the enemy of software security?. In Proceedings of the 4th ACM workshop on Quality of protection (pp. 47-50). ACM.

[12] Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3B: System Programming Guide, Part 2.

[13] Barth, Adam, Collin Jackson, and William Li. "Attacks on javascript mashup communication." Proceedings of the Web. Vol. 2. 2009.

[14] Goktas, E., Athanasopoulos, E., Bos, H., & Portokalidis, G. (2014, May). Out of control: Overcoming control-flow integrity. In Security and Privacy (SP), 2014 IEEE Symposium on (pp. 575-589). IEEE.

[15] Poon, Wing-Chi, and Aloysius K. Mok. "Improving the Latency of VMExit Forwarding in Recursive Virtualization for the x86 Architecture." System Science (HICSS), 2012 45th Hawaii International Conference on. IEEE, 2012.

[16] Sheldon, M., and Ganesh Venkitachalam Boris Weissman. "Retrace: Collecting execution trace with virtual machine deterministic replay." Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007). 2007.

[17] Dinaburg, A., Royal, P., Sharif, M., & Lee, W. (2008, October). Ether: malware analysis via hardware virtualization extensions. In Proceedings of the 15th ACM conference on Computer and communications security (pp. 51-62). ACM.

[18] Shannon, C. E. (1957). A universal Turing machine with two internal states. Automata studies, 34, 157-165.

[19] Nikhil, R. S. (1989, April). Can dataflow subsume von Neumann computing?. In ACM SIGARCH Computer Architecture News (Vol. 17, No. 3, pp. 262-272). ACM.

[20] Lee, D., Choi, Y., Jung, J., Kim, J., & Won, D. (2015). An efficient categorization of the instructions based on binary excutables for dynamic software birthmark. International Journal of Information and Education Technology, 5(8), 571.

[21] Dandamudi, Sivarama P. Introduction to assembly language programming: from 8086 to Pentium processors. Springer Science & Business Media, 2013.

[22] Boggs, D., Baktha, A., Hawkins, J., Marr, D. T., Miller, J. A., Roussel, P., ... & Venkatraman, K. S. (2004). The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. Intel Technology Journal, 8(1).

[23] Russinoff, D. M. (2000, November). A Case Study in Formal Verification of Register-Transfer Logic with ACL2: The Floating Point Adder of the AMD Athlon TM Processor. In Formal Methods in Computer-Aided Design (pp. 22-55). Springer Berlin Heidelberg.

[24] Petroni Jr, N. L., & Hicks, M. (2007, October). Automated detection of persistent kernel control-flow attacks. In Proceedings of the 14th ACM conference on Computer and communications security (pp. 103-115). ACM.

[25] Bhandarkar, D., & Clark, D. W. (1991, April). Performance from architecture: comparing a RISC and a CISC with similar hardware organization. In ACM SIGARCH Computer Architecture News (Vol. 19, No. 2, pp. 310-319). ACM.
[26] Hennessy, J., Jouppi, N., Przybylski, S., Rowen, C., Gross, T., Baskett, F., & Gill, J. (1982, October). MIPS: A microprocessor architecture. In ACM SIGMICRO Newsletter (Vol. 13, No. 4, pp. 17-22). IEEE Press.
[27] Tanenbaum, A. S., & Bos, H. (2014). Modern operating systems. Prentice Hall Press.
[28] Schneider, F. B., Morrisett, G., & Harper, R. (2001). A language-based approach to security. In Informatics (pp. 86-101). Springer Berlin Heidelberg.
[29] Andronick, J., Greenaway, D., & Elphinstone, K. (2010, October). Towards Proving Security in the Presence of Large Untrusted Components. In SSV.
[30] Liedtke, J. (1995). On micro-kernel construction (Vol. 29, No. 5, pp. 237-250). ACM.
[31] Roch, B. (2004). Monolithic kernel vs. Microkernel. TU Wien.
[32] Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R.,& Wiegert, J. (2006). Intel Virtualization Technology for Directed I/O. Intel technology journal, 10(3).
[33] Chiueh, T. C., Venkitachalam, G., & Pradhan, P. (1999). Integrating segmentation and paging protection for safe, efficient and transparent software extensions. ACM SIGOPS Operating Systems Review, 33(5), 140-153.
[34] van de Ven, A., Patel, B. V., Mallick, A. K., Neiger, G., Coke, J. S., Dixon, M. G., & Brandt, J. W. (2011). U.S. Patent Application No. 13/997,857.
[35] Arce, I. (2004). The shellcode generation. Security & Privacy, IEEE, 2(5), 72-76.
[36] Xiong, Haiquan, and Zhiyong Liu. "The Architectural Based Interception and Identification of System Call Instruction within VMM." (2013).
[37] Molnar, David, et al. "The program counter security model: Automatic detection and removal of control-flow side channel attacks." Information Security and Cryptology-ICISC 2005. Springer Berlin Heidelberg, 2005. 156-168.
[38] Peleg, A., & Weiser, U. (1996). MMX technology extension to the Intel architecture. Micro, IEEE, 16(4), 42-50.

[39] Katona, G. O., & Nemetz, T. O. (1976). Huffman codes and self-information. Information Theory, IEEE Transactions on, 22(3), 337-340.

[40] Crampton, J. (2005, June). A reference monitor for workflow systems with constrained task execution. In Proceedings of the tenth ACM symposium on Access control models and technologies (pp. 38-47). ACM.

[41] Smith, S. (2013). Trusted computing platforms: design and applications. Springer.

[42] Pfleeger, C. P., & Pfleeger, S. L. (2002). Security in computing. Prentice Hall Professional Technical Reference.

[43] Lin, T. Y. (1989, December). Chinese wall security policy-an aggressive model. In Computer Security Applications Conference, 1989., Fifth Annual (pp. 282-289). IEEE.

[44] Elliott Bell, D. (2011). Bell–La Padula Model. Encyclopedia of Cryptography and Security, 74-79.

[45] Ge, X., Polack, F., & Laleau, R. (2004, June). Secure databases: an analysis of Clark-Wilson model in a database environment. In Advanced Information Systems Engineering (pp. 234-247). Springer Berlin Heidelberg.

[46] Abadi, M., Budiu, M., Erlingsson, U., & Ligatti, J. (2005, November). Control-flow integrity. In Proceedings of the 12th ACM conference on Computer and communications security (pp. 340-353). ACM.

[47] Lombardi, F., & Di Pietro, R. (2011). Secure virtualization for cloud computing. Journal of Network and Computer Applications, 34(4), 1113-1122.

[48] Tsifountidis, Fotis. "Virtualization Security: Virtual Machine Monitoring and Introspection." Signature (2010).

[49 Smirnov, Alexey, and Tzi-cker Chiueh. "DIRA: Automatic Detection, Identification and Repair of Control-Hijacking Attacks." NDSS. 2005.

[50] Zhang, Chao, et al. "Practical control flow integrity and randomization for binary executables." Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013.

[51] Griswold, William G., et al. "Modular Software Design with Crosscutting Interfaces."

[52] Tsoutsos, N. G., & Maniatakos, M. (2014). Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation. Emerging Topics in Computing, IEEE Transactions on, 2(1), 81-93.

[53] Davi, Lucas, et al. "Poster: control-flow integrity for smartphones." Proceedings of the 18th ACM conference on Computer and communications security. ACM, 2011.

[54] Davi, Lucas, et al. "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones." NDSS. 2012.

[55] Bermudo, Nerina, Andreas Krall, and Nigel Horspool. "Control flow graph reconstruction for assembly language programs with delayed instructions." Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on. IEEE, 2005.

[56] Vigna, Giovanni. "Static disassembly and code analysis." Malware Detection. Springer US, 2007. 19-41.

[57] Abadi, M., Budiu, M., Erlingsson, U., & Ligatti, J. (2005, November). Control-flow integrity. In Proceedings of the 12th ACM conference on Computer and communications security (pp. 340-353). ACM.

[58] Tsifountidis, Fotis. "Virtualization Security: Virtual Machine Monitoring and Introspection." Signature (2010).