A Study on Kernel Memory Protection and Application Traffic Monitoring for Information Leakage Prevention

2020, March

Hiroki KUZUNO

Graduate School of Natural Science and Technology
(Doctor's Course)
OKAYAMA UNIVERSITY

Contents

1 Introduction **1**

1.1 Information Leakage 1

1.2 Multi-layered defense as an information leakage countermeasure 3

1.3 Background of the Problems 5

1.3.1 Threats to the Operating System 6

1.3.2 Threats to Applications 9

1.4 Research Problems 11

1.4.1 Operating System Monitoring 11

1.4.2 Application Traffic Monitoring 14

1.5 Related Work 15

1.5.1 Operating System Security 16

1.5.2 Application Information Leads Leakage Detection 19

1.6 Research Strategies 22

1.6.1 Detection of Kernel Memory Corruption 22

1.6.2 Detection of Information Leakage Network Traffic 23

1.6.3 Detection of Suspicious Application Library Leads Information Leakage . 24

1.7 Outline of the Dissertation 25

2 Detecting and Identifying Kernel Memory Corruption **26**

2.1 Introduction 26

2.2 Background 27

2.2.1 Virtual Memory Management 27

2.2.2 Separation of Virtual Memory 28

2.3 KMO Design 29

2.3.1 Design Goal 29

2.3.2 Switching Patterns and Detection Capability . . . 30
2.3.3 Design Approach . . . 31
2.4 KMO Implementation . . . 32
2.4.1 Secret Virtual Memory Space Management . . . 32
2.4.2 Switching of the Virtual Memory Space . . . 33
2.4.3 Monitoring of Virtual Memory Space . . . 34
2.4.4 Direct Mapping Management . . . 35
2.4.5 Kernel Vulnerability Attacking Case . . . 36
2.5 Evaluation . . . 37
2.5.1 Evaluation Purpose and Environment . . . 37
2.5.2 Monitoring System Call Argument . . . 38
2.5.3 Detection of Linux Security Module Overwrite . . . 38
2.5.4 Evaluation of Direct Mapping Access Validation . . . 40
2.5.5 Measurement System Call Interaction Overhead . . . 40
2.5.6 Measurement Application Overhead . . . 42
2.6 Discussion . . . 42
2.6.1 Performance Consideration . . . 42
2.6.2 KMO Detection Capability . . . 43
2.7 Conclusion . . . 44

3 Detection of Sensitive Information Leakage in Android Applications 45
3.1 Introduction . . . 45
3.2 Background . . . 46
3.2.1 Android Architecture . . . 46
3.2.2 Android Permissions . . . 47
3.3 Problem Description . . . 48
3.3.1 Application Request Permissions . . . 48
3.3.2 Application Traffic Analysis . . . 48
3.4 Approach . . . 50
3.4.1 Overview . . . 51
3.4.2 HTTP Packet Destination Distance . . . 52
3.4.3 HTTP Packet Content Distance . . . 53
3.4.4 Hierarchical Clustering . . . 54

3.4.5 Signature Generation . . . 55
3.4.6 Signature Screening . . . 55
3.5 Evaluation . . . 55
3.5.1 Experimental Setup . . . 55
3.5.2 Experimental Results . . . 56
3.6 Discussion . . . 58
3.6.1 Approach Consideration . . . 58
3.6.2 Complexity Analysis . . . 58
3.7 Conclusion . . . 59

4 Detecting and Characterizing of Mobile Advertisement Network Traffic 62
4.1 Introduction . . . 62
4.2 Background . . . 63
4.2.1 Permission Framework . . . 63
4.2.2 Advertisement Modules . . . 64
4.3 Problem Description . . . 64
4.3.1 Advertisement Modules Behavior . . . 64
4.3.2 Advertisement Modules Traffic Analysis . . . 65
4.4 Approach . . . 67
4.4.1 Graph Definition . . . 69
4.4.2 Graph of HTTP Sessions . . . 69
4.4.3 Graph Distance . . . 71
4.4.4 HTTP Session Distance . . . 71
4.4.5 Vertex and Edge Matching Rules . . . 72
4.5 Evaluation . . . 72
4.5.1 Experimental Setup . . . 72
4.5.2 Experimental Results . . . 73
4.5.3 Training Set Screening . . . 76
4.6 Discussion . . . 76
4.6.1 Detection Rate Consideration . . . 76
4.6.2 False Detection Rate Consideration . . . 80
4.6.3 Ad Module Network Traffic Consideration . . . 80
4.6.4 Complexity Analysis . . . 81

4.7 Conclusion 81

5 Conclusions **84**

5.1 Concluding remarks 84

5.2 Future directions 86

Acknowledgements **88**

References **89**

List of Figures

1.1 Annual statistics of information leakage cases in Japan [1]. 2
1.2 Scenario of cyberattack, threat, and countermeasure [20, 21, 22]. 4
1.3 Overview of threats and multi-layered defenses. 5
1.4 Number of kernel vulnerabilities registered to CVE. 11
1.5 Operating system security comparison of related work. 17
1.6 Application traffic monitoring and modeling comparison of related work. 20
1.7 Approach of this study to the problems of multi-layer security. 23

2.1 Multiple page table converts virtual address into physical address. 27
2.2 Overview of monitoring on the secret virtual memory space. 28
2.3 Virtual memory switching patterns 1, 2, and 3 29
2.4 Overview of secret virtual memory space for Linux kernel. 32
2.5 Virtual memory space switching on Linux kernel. 33
2.6 Position and unmap region for the virtual memory space on Linux x86_64. 34
2.7 Monitoring attacker process using the secret virtual memory space on Linux. . . . 35
2.8 Monitoring result for Linux system call arguments. 38
2.9 Monitoring result for LSM function 39
2.10 Preventing result for modification through direct mapping. 39

3.1 Overview of Android architecture 46
3.2 Frequency Distribution of HTTP Host Destinations. Out of 1,188 applications total, 81 (7%) have 1 destination, 885 (74%) have up to 10 destinations, and average number of destinations was 7.9. 50
3.3 (a) The architecture of proposed clustering and signature generation system. (b) The information flow control application that uses the signatures generated by (a). . 51
3.4 Detection Rate of Sensitive Information Leakage. 57

4.1 An overview of an ad modules' network behavior. Application bundles include ad modules, which connect to their suppliers' servers to download ad images or provide user statistics. 63
4.2 An overview of the organization and permissions of an application that includes an ad module. The ad module can use the application's permissions the access sensitive information on the device and send it over the network. 65
4.3 An overview of proposed approach. First, the known ad module network traffic is separated out. Next, ad graphs from the remaining network traffic is extracted by comparing the candidate graph distance from ad graph. Finally, new ad graphs is predicted. 67
4.4 Ad module HTTP session graphs. (a) Graph of doubleclick, consisting of 5 vertices and 4 edges. One image vertex connects to both JavaScript and HTML vertices, while the other image vertex connects only to the HTML vertex. (b) Graph of mydas, consisting of 8 vertices and 7 edges. All 3 image vertices share the same cookie, but connect to different HTML vertices. One HTML vertex connects to another HTML vertex, as well as a JavaScript vertex. 70
4.5 Graph distance statistics using N known ad graphs to classify other known ad graphs. 77
4.6 Graphs distance statistics for candidate ad graphs using N known ad graphs. 77
4.7 Detection rate of other known ad graphs using N known ad graphs. 78
4.8 Detection rate of candidate ad graphs using N known ad graphs. 78
4.9 False positive rate of standard graphs using N known ad graphs. 79
4.10 Improved Detection rate of know ad graphs and other known ad graphs using N known ad graphs except for only one vertex graphs. 79

List of Tables

1.1 Information leakage cases by type. . . . 3
1.2 Countermeasure guidelines and standards for information leakage. . . . 3
1.3 Types of Vulnerabilities [28]. . . . 6
1.4 Effects of kernel vulnerability attack [29]. . . . 7
1.5 Types of kernel vulnerability implementations [29]. . . . 7
1.6 Kernel exploit techniques [30]. . . . 8
1.7 Types of application threat. . . . 8
1.8 Effects of application-layer threats. . . . 9
1.9 Combinations of excessive permissions (✓is permission request). . . . 9
1.10 Methods of application analysis (✓is supported; △ is partially supported). . . . 10
1.11 Six categories of the 130 memory corruption vulnerabilities (CVE registered) for Linux kernel. . . . 13
1.12 PoC available Linux memory corruption vulnerabilities list since 2016. Types are referring to DoS: denial-of-service, Mem. Corr.: Memory Corruption, Priv: Gain Privileges . . . 13
1.13 Operating system security. . . . 16
1.14 Kernel monitoring feature comparison (✓is supported; △ is partially supported). . 18
1.15 Application traffic monitoring and modeling. . . . 19
1.16 Application traffic monitoring feature comparison (✓is supported; △ is partially supported). . . . 21
1.17 Solving problems in multi-layer security in this dissertation. . . . 22

2.1 Overhead of switching virtual memory space and monitoring (μs). . . . 40
2.2 ApacheBench overhead of virtual memory switching and monitoring on the Linux kernel (μs). . . . 41

2.3 ApacheBench overhead of virtual memory switching patterns 1 and 3 with monitoring (μs). 41

3.1 Number of applications with dangerous permission combinations. Out of 1,188 applications total, 55% required both the **INTERNET** permission and at least one permission for sensitive information. 47
3.2 HTTP packet destinations. This table shows the number of packets sent to each HTTP host destination, and the number of applications that send packets to each HTTP host destination. 60
3.3 Sensitive Information. This table shows for each type of information considered sensitive, the number of packets containing the information, the number of applications that send those packets, and the number of destinations to which those packets go. 61

4.1 The number of HTTP GET requests for each type of with content. The most commonly requested content types (in decreasing order) are image files, script files, and document files. 66
4.2 Content types banner image sizes in HTTP GET responses. Banners of size 320x50 are most commonly downloaded by ad modules. Banners of size 300x48 are the next most common. 67
4.3 The number of applications that include ad modules, and the methods those ad modules used to download images. The total number of applications exceeds the number of analyzed applications, because some applications contain more than one ad module. 68
4.4 The number of graphs associated with each known ad module, converted from the dataset traffic. Ad modules' traffic was selected using destination domain matching. 74
4.5 The number of candidate ad graphs, associated with various ad modules. The traffic to convert to graphs was selected using domain matching in the HTTP headers and patterns of ad image downloading. 75

Summary

Managing the information assets of public and private organizations is a rapidly growing and important field. Information leakage is a significant problem for these organizations, forcing them to develop information security and damage-mitigation methods. In Japan, an average of 1,231 incidents of information leakage occurred every year from 2005 to 2017. Electronic media have had over half of the incidents than did paper media since 2016.

Cyberattack is a the principal cause of information leakage occurs. Such attacks may be either internal or external. The adversary uses a vulnerability of the software, a missing implementation of the hardware, or a weak user password to intrude on the internal network and its constituent devices. The danger of cyberattacks has led to information leakage countermeasures, to the adoption of international standards and sector guidelines, and to the requirement of certification in incident protection for information systems and organization staffs. A coordinated response, and cooperation between affected organizations, is essential. Incident response teams are now commonly set up to prevent information leaks, to react rapidly to incidents when they occur, and to handle disclosure.

One useful strategy is to employ multiple layers of defense for protection against cyberattack, reduction of damage, and identification of perpetrators. Mutual complementation of multiple layers suppresses risk to an information system. It ensures that another mitigating security mechanism is already prepared when one has been compromised. Network, hardware, and software, in this strategy, all have their own security mechanisms with appropriate policies and run-time assurances. Additionally, penetration testing is constantly used to simulate cyberattacks on the information system. Incident handling and digital forensics units, working together, identify the effects of the incident on the information system.

In this dissertation, the focus is on defending the operating system (OS) and the applications. These layers manage and handle information assets on information systems, then detect and prevent information leakage. Moreover, in order to identify information leakage using appropriate

detection and prevention methods, it is difficult to monitor the actual behavior of OS, application traffic contents, and identification of the application from inside the network. The combination of OS monitoring, application traffic monitoring, and identification of suspicious libraries of applications is necessary to achieve information security through multiple layers of defense:

- OS monitoring
  To identify kernel memory corruption and illegally overwritten kernel code, and malevolent data in the kernel virtual memory

- Application traffic monitoring
  To identify information leakage and other traffic from applications requesting excessive permissions

- Suspicious application library identification
  To use network traffic modeling to identify applications requesting excessive permissions

Countermeasures against attacks targeting an OS kernel can be highly effective in preventing security compromises caused by kernel vulnerabilities. An adversary uses such attacks to overwrite credentials, thereby gaining the ability to overcome security features through arbitrary program execution. To protect the OS, CPU features such as supervisor mode access prevention, supervisor mode execution prevention, and the no-execute bit facilitate access-permission control and data execution in the virtual memory. Linux further reduces the danger from kernel exploits by using several other protective methods, such as kernel address space layout randomization, control flow integrity, and kernel page table isolation. A combination of these methods can indeed mitigate the attack, as kernel vulnerability relies on the interaction between the user and kernel modes; nevertheless, kernel virtual memory corruption can still occur (e.g., the eBPF vulnerability allows malicious memory overwriting in the kernel mode). The kernel memory observer (KMO), an alternative design for virtual memory proposed in this dissertation, uses a secret observation mechanism to monitor kernel virtual memory and detect illegal data manipulation/writing. KMO determines whether kernel virtual memory corruption has occurred, inspects system-call arguments, and forcibly unmaps the direct mapping area. An evaluation of KMO reveals that it can detect kernel virtual memory corruption and quickly identifies the failing security feature through actual kernel vulnerabilities. In addition, the results indicate that the system call overhead latency ranges from 0.002 μs to 8.246 μs, and the web application benchmark from 39.70 μs to 390.52

μs for each HTTP access, whereas KMO reduces these overheads by using tag-based translation lookaside buffers.

The applications layer also requires defense. Many applications that are "free" to the user carry advertisements. While advertisement (ad) modules provide useful services, they also track user behavior statistics for commercial or other purposes. Users generally accept this business model, but, in most cases, the applications do not require a user's permission to transmit sensitive information. Therefore, such applications' behavior potentially constitutes an invasion of privacy. Analyzing the network traffic and permissions of 1,188 Android applications, it was found that 93% connected to multiple destinations when using the network, and 55% required a permission combination that included both access to sensitive information and the use of networking services, with obvious potential for leakage. Of the 107,859 HTTP packets from these applications, 23,309 (22%) contained sensitive information, as identified by string matching with a device identification number and carrier name. To enable users to control the transmission of their private information, the proposed system uses a novel clustering method based on the HTTP packet destination and content distances to generate signatures and thereby detect the leakage of sensitive information from Android applications. This system does not require that the Android framework be modified. Thus, users can easily add the proposed system to their devices and manage the network behavior of suspicious applications in a fine-grained manner. In tests, the system accurately detected 97% of the sensitive information leakage from the applications evaluated, with only 3% false negative results and 3% false positive results.

Another goal is to identify suspicious application libraries likely to leak information. In analyzing the network traffic of 1,188 Android applications, the analyzing result indicates 797 applications that included previously known ad modules. It was noted that ad modules exhibit characteristic network traffic patterns for acquiring repeatedly used content, specifically images. These patterns are evident in network traffic graphs mapping the relationships among HTTP session data (such as HTML or JavaScript). In order to accurately differentiate between the ad modules' network traffic and valid application network traffic, the proposed system adopts a novel method based on the distance between session graphs and the graphs of known ad modules. This distance describes the similarity between the sessions. For evaluation, 20,903 graphs of applications were generated. The system separated the application graphs into those generated by known ad modules (4,698 graphs), those manually identified as ad modules (2,000 graphs), and standard application traffic (2,000 graphs). The system applied 1,000 graphs of known ad modules

to the other graph sets (the remaining 3,698 known ad graphs, the 2,000 manually classified ad graphs, and 2,000 standard graphs) to see how accurately ad graphs could be distinguished. The evaluation showed a 76% detection rate for known ad graphs, a 96% detection rate for manually classified ad graphs, and an under 10% false positive rate for standard graphs.

In sum, this dissertation presents several elements of a multi-layer defense against information leakage, concentrating on the OS and application layers. To guard the OS against cyberattack, the KMO virtual memory system is proposed. To guard mobile device users against various forms of information leakage, two new detection methods are developed. The efficacy of these approaches is supported by the observed results. This work should prove valuable to future researchers in the field of information security.

Chapter 1

Introduction

1.1 Information Leakage

The information systems of public and private organizations possess large amounts of information used for service development and financial management. The leakage of information assets via unauthorized access has a significant impact on government, business, and individual security. The rate of information leakage in Japan is shown in Figure 1.1. In 2018 alone, there were 443 reported information leaks. On average, 1,231 incidents were reported every year from 2005 to 2017. Since 2016, electronic media have proven more vulnerable to leakage than paper media [1]. The largest cause of information leakage from electronic media is a cyberattack, which may involve outside unauthorized access, insider misuse of valid accounts, or loss of media devices [2, 3]. Outside attackers break through protective mechanisms by exploiting software or hardware vulnerabilities or guessing a user's weak password. Inside attackers use valid credentials (whether their own or stolen from other users) to log in to a file server or database server. In either case, adversaries reach information assets, then expose them to parties outside the organization. Leakage of personally identifiable information has taken place in both the national public sector [4, 5] and the private sector [6, 7, 8]. Additionally, specific cyberattacks sometimes directly target an organization's intellectual or financial property [9, 10, 11]. In this dissertation, information assets are categorized into three types: personal information, intellectual property, and financial property (see Figure 1.1). The International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC) 27000 series provides precise classification guidelines for information assets using stored media, secrecy level, and category attribution [12].

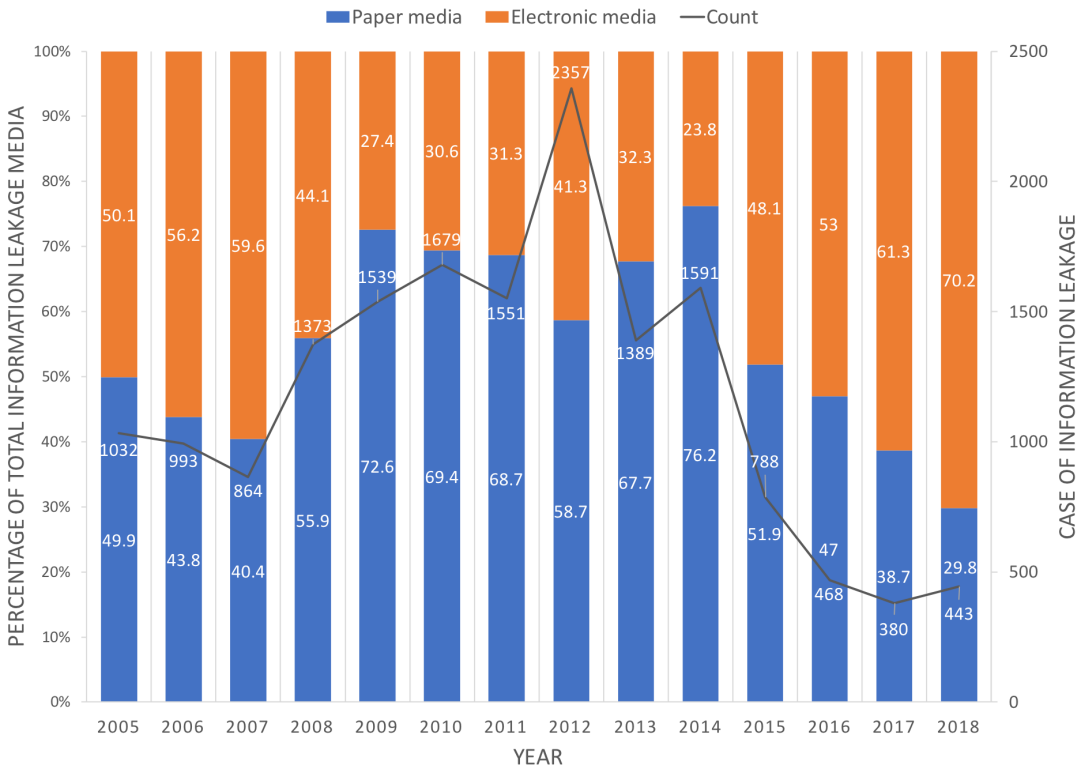

Figure 1.1 Annual statistics of information leakage cases in Japan [1].

Information security relies on the use of national, international, and industry-wide standards [2, 12, 13, 14, 15, 16]. These establish the combination of protective methods and organizational policies necessary to ensure security. Certification also indicates that staff have the education, psychology, and motivation that minimize the risk of internal illegality [17]. A computer security incident response team (CSIRT) and product security incident response team (PSIRT) manage the total security of the information system and products, and respond by handling any risks (e.g., vulnerability disclosure, patch management, and information leaking). Such handling requires a systematic incident response with related organizations [18, 19].

An information leakage scenario containing the elements of attack strategy, threat, and countermeasure is indicated in Figure 1.2. Lockheed Martin's cyber kill chain is a representative example of an advanced persistent threat (APT) attack; it defines seven stages of attack [20], from reconnaissance through device compromise [21]. MITRE ATT&CK provides a knowledge base of adversary activity, techniques, and incident details sufficient to identify attack paths and damage to information system without difficulty [22].

Table 1.1 Information leakage cases by type.

Type	Detail	External attack	Internal attack
Personal Information	Personally Identifiable Information	Ecuador [4]	Yahoo US [6]
		Japan Pension Service [5], Adobe [7]	Benesse [8]
Intellectual Property	Technical information, Software	Shadowbroker [9]	Trade Secret Disclosure
	Management Information		
Financial Property	Legal Currency, Securities	Bangladesh Bank [10]	Embezzlement
	Crypto asset	Coincheck [11]	

Table 1.2 Countermeasure guidelines and standards for information leakage.

Grade	Countermeasure document
Standard	NISC [2], NIST SP800 [13], ISO / IEC 27000 [12], and ISO 28000 [26]
Industry Standard	FISC [14], PCI DSS [15], HIPAA [16], and P Mark [17]
Education	Security Guideline of Targeted Email [21], Cloud [24], and IoT [25]

Multiple layers of defense are effective countermeasures against an attack and the subsequent compromised-device stage [23]. They include preparing suitable security quality of inbound, outbound , and internal of information asset control and management at the network and device levels and training management staff to have and retain cybersecurity knowledge and to take quick action in response to any incident.

1.2 Multi-layered defense as an information leakage countermeasure

Many computer devices connect through the network infrastructure; the constructing of recent information systems used the complex software to that manages user credential information, and information assets. This complexity means that multiple layers of defense are important for preventing information leakage with certainty, for incident response, and for digital forensics and damage reduction.

A multi-layered defense against several types of threats is summarized in Figure 1.3. The main purpose of the multiple layers of security is the prevention and detection of information leakage. Security mechanisms must assume a level of risk exists in each layer. It requires yet another layer that the adversary must break, another layer for reducing the damage, and one for verification and for tracing the attack sequence on the compromised information system.

Security mechanisms must adequately cover the entire information system: network, hardware,

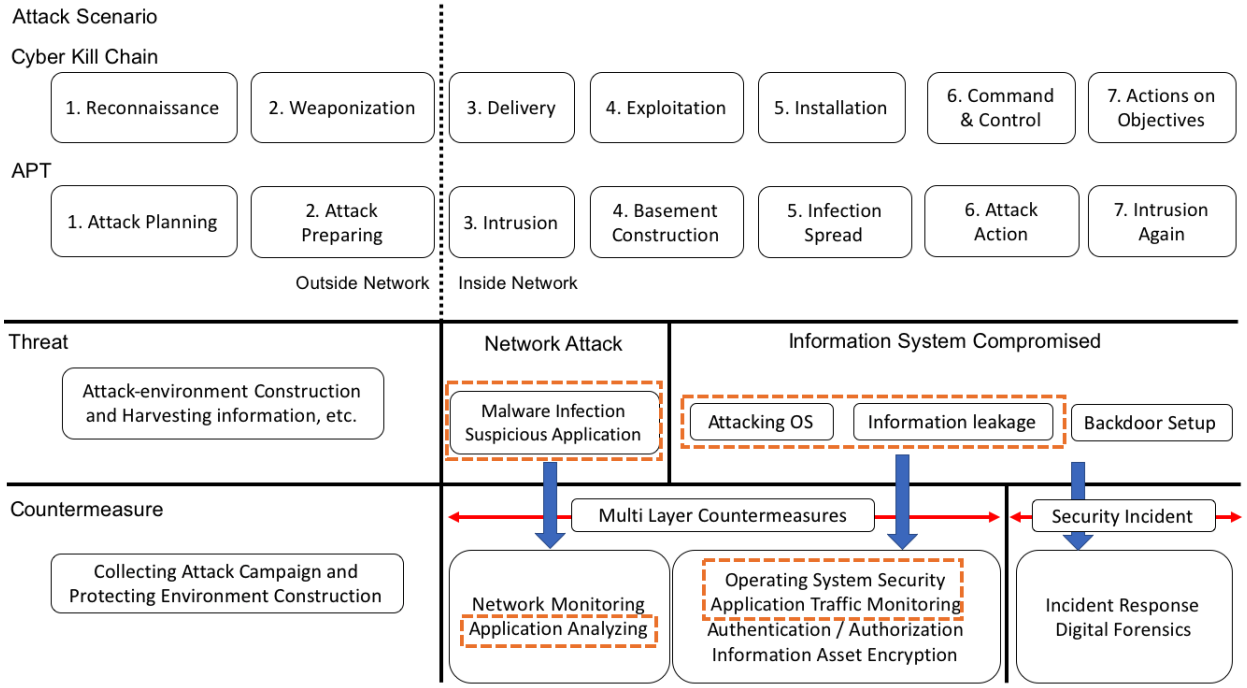

Figure 1.2 Scenario of cyberattack, threat, and countermeasure [20, 21, 22].

and software. For example, a Root-of-Trust supports the integrity of the hardware and software, while access control manages the relationship between an information asset and its potential users. Any practical information system requires a penetration test in which a cyberattack is simulated, incident-handling vulnerabilities are revealed to support patch management by the CSIRT / PSIRT, and the forensics are analyzed. Information on the system should be logged for sharing with law enforcement agencies. Such a combination of security mechanisms should support the gathering of clues from the compromised information system about the adversary's command and network activity and help with quick detection of the damage of disclosure due to the specific incident.

Attack techniques evolve as information systems do. Thus, new security guidelines are required for the Cloud [24], the Internet of Things (IoT) [25], and supply chain management [26]. With both old and new technologies, it is important to ensure that systems are constructed to ensure trust, the fundamental common component for the construction of system and service [27].

The operating system (OS) and the applications are necessary layers for storing and accessing an information asset. To identify that an outside adversary is attempting to seize direct control of these layers, appropriate monitoring is the most practical security mechanism. Combined monitoring of OS and application traffic to detect information leakage at these levels has not been attempted in

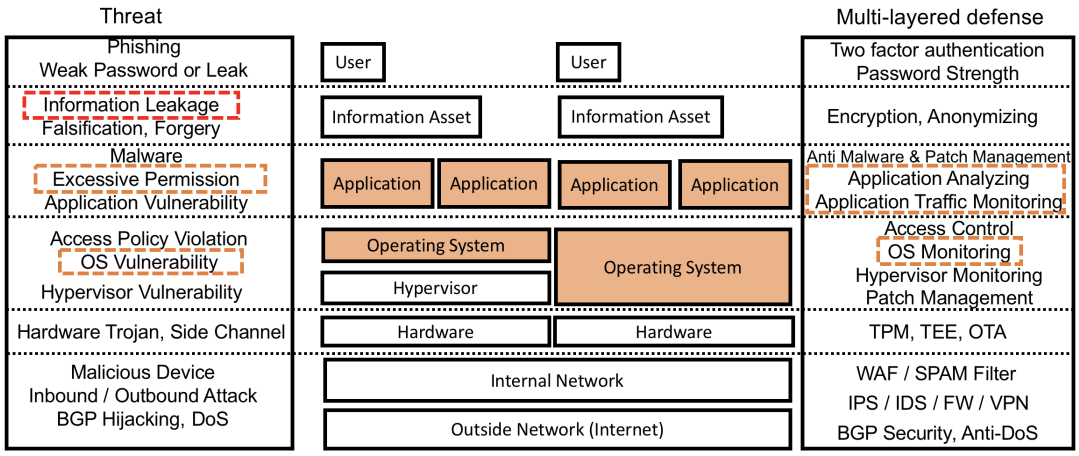

Figure 1.3 Overview of threats and multi-layered defenses.

previous research works. It is difficult to identify suspicious activity on the inside of OS kernel behavior, and it is also hard to supply the appropriate detection of information leakage, and then to characterize the application traffic model to identify the suspicious component of an application.

In this dissertation, appropriate OS monitoring, application traffic monitoring, and the identification of suspicious libraries of applications are used to solve the problems associated with an information security mechanism based on multiple layers of defense:

- OS monitoring

  To identify kernel memory corruption and illegally overwritten kernel code, data is kept in the kernel virtual memory.

- Application traffic monitoring

  To identify information leakage from excessive permission application.

- Suspicious application library identification

  To identify a suspicious application that gains excessive permission application, by network traffic modeling

1.3 Background of the Problems

OS and applications are the basic software for constructing information systems. Threats to one or the other include kernel vulnerabilities, application malware, application vulnerabilities,

Table 1.3 Types of Vulnerabilities [28].

Exploit	Content
DoS	Forcing shutdown of software
Code Execution	Arbitrary program execution
Overflow	Breaking of stack or heap memory space
Memory Corruption	Illegal overwriting in software memory space
SQL Injection	Arbitrary SQL insertion and execution
XSS	Arbitrary HTML or JavaScript code in Web application
Directory Traversal	Reading arbitrary directory
HTTP Response Splitting	Poisoning of cache by illegal HTTP response
Bypass Something	Evading of access limitations
Gain Information	Information gaining through illegal control
Privilege Escalation	Getting of administrator privilege
CSRF	Illegal HTTP request is accepted by Web application
File Inclusion	Forced reading of malicious file reading

and excessive permissions. The appropriate definition of these threats will provide a better understanding of the security approach detailed in section 1.4.

1.3.1 Threats to the Operating System

The main threat to the OS is an attack through kernel vulnerabilities, that is, software mis implementations in the kernel, which manages hardware devices and application execution. As seen in Table 1.3, the common vulnerabilities and exposures (CVE) classification system states that software has 13 types of vulnerabilities [28]. Kernel attack is involved in four of them, as indicated in 1.4: kernel memory corruption, policy violation, Denial of Service, and OS information leakage. Table 1.5 shows that Linux kernel implementations address 10 types of vulnerabilities [29]. Table 1.6 lists 16 exploitation techniques based on kernel vulnerabilities [30].

Suppose that the adversary finally achieves control of the information asset, then sends it to the outside environment. An attacking threat to the OS has two routes: remote attack through a network, and local attack through a user account after logging in to the device. Either requires appropriate permission from the OS that manages the information asset, whether on local storage or a remote file server. The adversary might also use privilege escalation to acquire an administrator account with the highest privilege level, able to read files and use network features.

Table 1.4 Effects of kernel vulnerability attack [29].

Effect	Content
Memory corruption	Overwriting or reading of kernel code / data on virtual memory
Policy violation	Miss implementation of access control decision
Denial of Service	Forcing kernel to stop running
OS information leakage	Information leakage from uninitialized data variables

Table 1.5 Types of kernel vulnerability implementations [29].

Type	Content
Missing pointer check	Lack of pointer variable verification
Missing permission check	Lack of permission verification
Buffer overflow	Overwriting of stack or heap space
Uninitialized data	Lack of initialization at variable creation
Null deference	Access to Null variable
Divide by zero	Zero dividing calculation
Infinite loop	Occurrence of infinite loop process
Data race / deadlock	Occurrence of race condition or deadlock
Memory mismanagement	Inconsistent allocation of memory allocation and free
Miscellaneous	Other wrong implementations

User account information is controlled on virtual memory by the kernel. Virtual memory is the mechanism by which the OS uses flexible memory space rather than physical memory space. The kernel prepares a virtual memory containing two types of page tables: user page-tables for each running user process, and kernel page-tables for the running kernel itself. The reading and writing of process, kernel code, and data require translation from a virtual address to a physical address, i.e., directory traversal of the page tables.

Privilege escalation means overwriting from a user account to an administrator account with the highest level of privilege (e.g., a root account on UNIX). A user account is usually managed by a virtual address on the kernel page tables that are protected by the kernel feature. The adversary may use a kernel vulnerability such as a memory corruption exploit that allows modification of kernel page tables, and, in this way, can insert any program code to the kernel. This can turn a user account into an administrator account, because kernel privilege allows the modification of

Table 1.6 Kernel exploit techniques [30].

Technique	Content
Metadata Corruption	Illegal modification of meta data on the file system
Control-Flow Hijack	Arbitrary control function call chain
Allocator Data Corruption	Overwriting of stack
Heap Layout Control	Control heap allocation mechanism
Userspace Data Access	Access to user data from kernel code
Finding Kernel Objects	Identification of kernel code or data virtual address
JIT Abuse	Illegal use of JIT feature (e.g., eBPF)
W^X Area Abuse	Malicious modification of writing or execution exclusive space
Changing Kernel Image	Malicious modification of kernel image
Bad Module Loading	Installation of malicious kernel module (e.g., Rootkit)
Unbalanced set_fs()	Misconfiguration of kernel and user address borderline

Table 1.7 Types of application threat.

Threat	Description
Malware	Malicious software leads to damage to information system or user device
Application Vulnerability	Mis implementation of software is used on attack
Excessive Permission	Suspicious software acquires multiple privileges

any virtual address on the kernel page tables.

An adversary who achieves privilege escalation, then obtains an administrator account, controls access to information assets, and can transfer them to the outside environment, or attack other computer systems through the compromised OS of the device. This dissertation proposes to circumvent this kernel-layer threat by supporting the appropriate detection of kernel page-table modification after privilege escalation.

- OS monitoring

  To support the appropriate detection of kernel page table modification after the privilege escalation through kernel vulnerability attack at the kernel layer.

Table 1.8 Effects of application-layer threats.

Threat	Effect
Ransom demanded for information asset	Loss of access to information asset
Sending information asset to outside	Information leakage and breach
Infection of other computer devices	Risk and damage to internal and external systems

Table 1.9 Combinations of excessive permissions (✓is permission request).

	Permission			Effect
	Information asset access	Application interaction	Network	
Combination 1	✓		✓	Sending information asset to outside
Combination 2	✓	✓		Sharing information asset with other applications
Combination 3	✓	✓	✓	Sending information asset to outside Sharing information asset with other applications

1.3.2 Threats to Applications

The threats to applications listed in Table 1.7 include malware, application vulnerability, and excessive permission. These exploits are summarized in Table 1.15.

Malware is explicitly malicious software, of which there are several types. Worms infect other computer devices via a network; ransomware demands a ransom to decrypt files; and bots send spam mail, causing a denial of service and information leakage from the command and control server. Malware infection typically spreads through the downloading and execution of malicious files attached to an email message or placed on a compromised web site, although remote attacks through kernel or application vulnerabilities can also involve malware.

Application vulnerabilities, like kernel vulnerabilities, are software mis implementations. Their effects are shown in Table 1.3. After an attack's success, an adversary could use application privileges to cause damaging effects, just as with malware.

Excessive permissions allow access to the information asset and use of the network feature of the device at the same time. Excessive permission combinations are summarized in Table 1.9. The application contains the main component; third party libraries require different permission to run on the device. As a result, application requests to the user involve a combination of privileges. This leads to the risk of information leakage, as pointed out in previous research work [31].

The granularity of privileges is an important principle. The UNIX access control allows users

Table 1.10 Methods of application analysis (✓is supported; △ is partially supported).

Techniques	Malware	Application vulnerability	Excessive permission	Content
Static analysis	✓	△	△	Disassemble and analyze application code and binaries
Dynamic analysis	△	✓	✓	Analyze application internal behavior and network traffic

and groups to have the read, write, and execute permissions. Mandatory access control (MAC) is stricter, designating category and level for users and files. Additionally, the capability and permission allow or disallow the right to using of computer device resources (e.g., network, store, and information asset). Several combinations of privileges may be required by the application; this potentially increases the risk of damage from malware.

Methods for identifying application risk may be either static or dynamic (see Table 1.10). The static analysis method disassembles an application's execution file into source code or assembly language, then analyzes the program structures and components. The dynamic analysis method directly monitors API calls, system call invocation, memory usage, and network traffic content, then analyzes this historical information while the application is running.

Static analysis is appropriate for extracting details about the application source code, such as whether it contains a known vulnerability or grants excessive permissions. Moreover, static analysis takes less time to analyze the entire application than does dynamic analysis. On the other hand, it has difficulty identifying the actual use being made of the application's features and privileges. Static analysis cannot support file, memory, and network usage that is determined at the running time. Additionally, obfuscation, packing techniques, hardening of an application using complex modification and encryption are countermeasures by which an adversary can thwart a static analysis.

Dynamic analysis is available to identify whether an application has a specific vulnerability to attack, and information assets are contained on file reading, writing, and transferring via a network. The weak points of dynamic analysis are that, if it cannot cover entirety of a running application because of time limitations, it does not collect the pertinent information when the application camouflages its behavior or does not connect to a suspicious server.

An effective countermeasure for the threat to applications requires the combination of static analysis and dynamic analysis to cover the entire application and its actual behavior to analyze the risk posed by the threat. Specifically, as dynamic analysis is an effective approach to identifying information leakage, this dissertation focuses on the following approaches:

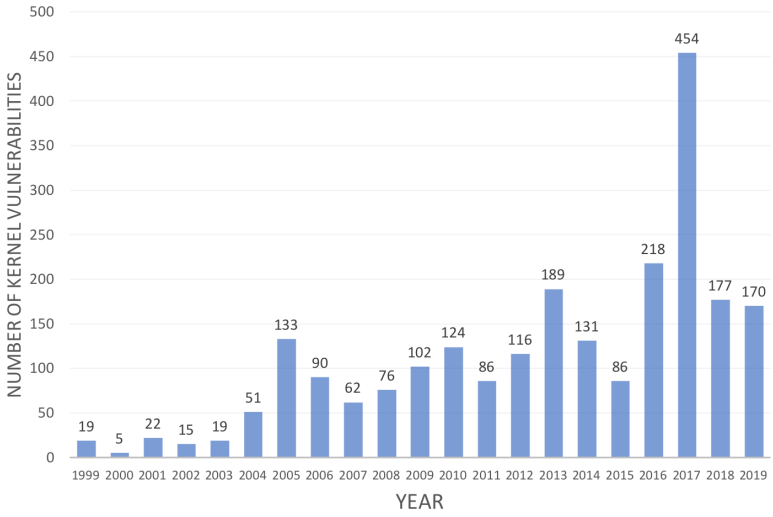

Figure 1.4 Number of kernel vulnerabilities registered to CVE.

- Application traffic monitoring
  application traffic monitoring on a suitable monitoring system to detect and reduce the risk of information leakage that is caused by excessive permissions;

- Suspicious application library identification
  suspicious application library identification through the analysis and modeling of application network traffic.

1.4 Research Problems

In this section, the research problems addressed in the dissertation are enumerated. The first concerns information leakage owing to the privilege escalation through kernel vulnerability attacks on OS security. The second concerns information leakage due to excessive permission application. The third concerns third party library of application.

1.4.1 Operating System Monitoring

The concerning of information leakage from the information system. OS has responsible for access control between a user account and an information asset. Adversaries can exploit the OS kernel through such vulnerabilities. Preventive countermeasures must be developed to mitigate their attacks. The adversary uses kernel vulnerabilities to modify the credentials in the

kernel virtual memory; such privilege escalation can force the OS to grant root privilege to a non-privileged user account. It allows adversaries access information assets and network features for achieving to information leakage using root privilege.

Kernel vulnerabilities in OS have become a significant security risk [29, 32]. There are 2,240 kernel vulnerabilities in the Linux that were listed on the CVE database in 2019 (Figure1.4) [28]. Full privilege is restricted by two OS features, namely the capability [33] and MAC (e.g., SELinux [34]) mechanisms.

The kernel has several countermeasures to prevent kernel vulnerability and data misuse. Kernel address space layout randomization (KASLR) distributes the kernel functions and data positions in the kernel virtual memory to conceal the virtual addresses of vulnerable functions [35] Additionally, control flow integrity (CFI) enforces kernel function flow validation between call and return relationships to prevent the injection of malicious code [36]. Return address monitoring of the stack is one method to detect kernel memory corruption to prevent malicious code initialization [37].

The CPU also has virtual memory access and an execution permission mechanism. The NX bit (No eXecute bit) is the execution permission flag for a virtual address in a page table entry [38]. Supervisor mode access prevention (SMAP) and supervisor mode execution prevention (SMEP) are in the CR4 register. SMAP prevents access to the user memory region, whereas SMEP prevents code execution in the user memory region of the virtual memory at the supervisor [39]. Meltdown vulnerability uses a side channel approach to expose directory kernel functions and data virtual addresses. Therefore, kernel page table isolation (KPTI) has been proposed as a means of isolating the virtual address space between the user and kernel modes in Linux [40].

Privilege restriction methods separate root privilege features to minimize the damage to the OS environment in the event of a successful attack. Kernel and CPU countermeasures complicate (from the attacker's point of view) the availability of kernel vulnerabilities based on the interaction between the user and kernel modes. However, these methods cannot prevent attacks that exploit kernel vulnerabilities in the kernel mode alone [41, 42, 43, 44]. The adversary can avoid many security countermeasures by executing a kernel exploit code in the kernel mode to override the security feature functions in the kernel virtual memory (e.g., some kernel exploits disable SELinux via memory corruption [49, 50]).

Table 1.11 Six categories of the 130 memory corruption vulnerabilities (CVE registered) for Linux kernel.

DoS	Code Execution	Overflow	Bypass Feature	Gain Information	Gain Privileges
114	6	73	1	4	25

Table 1.12 PoC available Linux memory corruption vulnerabilities list since 2016. Types are referring to DoS: denial-of-service, Mem. Corr.: Memory Corruption, Priv: Gain Privileges

CVE ID	Types	PoC	Publish Date	Description
CVE-2017-16995 [44]	DoS, Overflow, Mem. Corr.	✓	2017-12-27	A boundary check error in kernel/bpf/verifier.c
CVE-2017-1000112 [45]	Mem. Corr.	✓	2017-10-04	A race condition in net/ipv4/ip_output.c
CVE-2017-7533 [46]	DoS, Priv, Mem. Corr.	✓	2017-08-05	A race condition in the fsnotify implementation
CVE-2016-9793 [47]	DoS, Overflow, Mem. Corr.	✓	2016-12-28	A boundary check error in net/core/sock.c
CVE-2016-4997 [48]	DoS, Priv, Mem. Corr.	✓	2016-07-03	A boundary check error in setsockopt implementation

Problem 1: Kernel Memory Corruption

Kernel vulnerabilities are implemented in several ways. Privilege escalation uses malicious programs to overwrite the credential variable in the kernel virtual memory and gain root privilege.

The OS utilizes privilege level management to protect the kernel code or data in the kernel virtual memory from the user mode, while KASLR / CFI reduces the success of kernel exploitation attacks, and SMAP / SMEP restricts the kernel mode execution of malicious code in the user virtual memory. Nevertheless, 128 memory corruption vulnerabilities were reported for the Linux kernel in the first half of 2019 (Table 1.11) [28]. Memory corruption vulnerabilities (e.g., the eBPF vulnerability and others in Table 1.12) are still available whereby the directory allocates malicious code to the kernel virtual memory through a kernel vulnerability, and continue to pose a problem.

In the postulated threat model, an adversary exploits kernel vulnerability only in the kernel mode, aiming to compromise the OS and become capable of running any program (e.g., shell command) without security restrictions. The attacker first attempts to avoid the security features and gain full administrator capability, changing the Linux security modules (LSM) hook function pointer variable to disable MAC in Linux.

The assumption of the threat model of a memory corruption kernel vulnerability involving overwriting of the kernel virtual memory space is that this occurs only in the kernel vulnerability target memory region that includes the security feature functions pointer, kernel module management data, and a direct mapping region. In addition, it is assumed that the BIOS, MMU,

TLB, and other hardware are safe.

1.4.2 Application Traffic Monitoring

With the increasing popularity of smartphones and tablets, development of mobile device operating systems (particularly for Apple's iOS and Google's Android, which are the most popular choices) has drastically increased, as has the development of applications for online marketplaces such as the AppStore and Google Play. Google's Android is currently the most popular operating system for smartphones, tablet devices, and application marketplaces. In February 2014, Google Play had over 1,100,000 applications [51]. Applications available in such marketplaces are categorized as either free or paid. This dissertation will primarily address free applications, which often come with advertisement modules.

A smartphone retains various kinds of personal information, such as location tracking data, the contents of the user's address book, and the device's unique identifier. Android provides a framework that requires applications to have specific permissions for accessing restricted resources. However, the Android permission framework does not completely protect the user's sensitive information. Applications or advertisement modules with certain permission combinations can send the user's sensitive information to outside servers using the network [31, 52, 53, 54, 55, 56]. There remain some problems:

- Excessive Permission Application
  In order to decouple the features of permission granularity (e.g., network access, camera, sensitive information), and thus maintain security.

- Suspicious Application Library
  While the user's sensitive information is generally used for targeted advertising, it can also be discovered and used by malicious parties without the original user's awareness.

Problem 2: Excessive Permission Application

The users can use a number of free applications that have network and sensitive information permissions. It is highly risky for users to run applications that have the possibility of sensitive information leakage. However, the users cannot determine if sensitive information is present or absent in their network traffic.

Therefore, Android provides the notification page of required permission details for the users when the application is installed. This is effective information that helps the users. Although they can understand what kinds of resources are accessed by the runtime of the application, the users cannot realize for what purpose the application obtains this information. Thus, they decide whether to install an application or not on the basis of the permission list.

In addition, Android's current model, an application requests permissions only once, on installation. After the application is installed, all its transmissions are opaque to the users, who have no way of determining if sensitive information is present in their network traffic. Users would ideally prefer that they could use an application without interruption when it is only transmitting benign data but be prompted for confirmation when the application shes to send sensitive information over the network.

Problem 3: Suspicious Applications Library

Some free applications include an advertisement (ad) module, which provides a targeted ad to the user and collects statistics about user behavior. Since the ad module providers pay the application developers to include these ads in their modules, these applications are offered at no cost to the user. This business model is widely accepted in the mobile market for "free" applications [57].

Android provides a permission framework for privilege management. To access restricted resources, an application needs specific permissions. Generally, ad modules are offered as a software development kit (SDK) library that an application includes in its distribution. Thus, an application bundle combines the original application permissions with the permissions of the ad module, and the user cannot determine which permission applies to which. Privilege separation methods have been proposed to address this problem [58, 59], but they require modifying the Android permission framework and presenting an accurate representation of the permissions to the users.

1.5 Related Work

In this section, works related to the above two threat models and three problems are reviewed.

Table 1.13 Operating system security.

Category	Security measures
CPU feature	NX-bit [38], SMAP / SMEP [39], MPK [63], TCB [92]
Access control	Capability [33], SELinux [34]
Kernel protection	CFI [36], Dataflow integrity [69]
	Stack checking [37], Variable checking [72]
Kernel image protection	kRˆX [74]
Kernel memory protection	KASLR [35], KPTI [40], PT-Rand [73], Reliability [80]
Memory monitoring (hardware)	GRIM [91]
Memory monitoring (virtualization)	SecVisor [87], TrustVisor [90]
Memory monitoring (in-kernel)	SIM (requires hardware support) [93]
	ED-Monitor (focuses a hypervisor kernel) [94]

1.5.1 Operating System Security

Operating system security. The OS provides several security mechanisms that mainly focus on access management of the relation between a subject and an object, and whether a policy exhibits granularity of privilege range or type. Moreover, isolation architectures and models that control separation combine in multiple layers from hardware to software [60, 61].

Access control. Linux also has security mechanism implementations such as SELinux [34, 62] and a capability [33] to restrict privileges.

CPU feature. The CPU already possesses the NX-bit [38] for execution management, and SMAP / SMEP [39] for access and the execution of joint control by a supervisor and a user of the virtual memory space. The CPU feature MPK supports virtual memory protection [63, 64], page-based separation instructions [65, 66], and physical memory isolation for each process [67].

Kernel protection. Prior studies have presented kernel security methods such as CFI [36] for code flow integrity checking, and Code Pointer Integrity for the verification of a function's return address [68]. In addition, running kernel protection methods focus on invalid overwriting of kernel code and data, including control flow or data flow tracing [36, 69, 70], monitoring of stack status [37, 71], and verifying of the privilege variable [72].

Kernel image protection. Kernel protection methods often involve randomized page table positions in the physical memory [73]. kRˆX restricts the permission of the kernel memory layout [74]. KMO has different merits: switching of the virtual memory space has no effect for attacks via

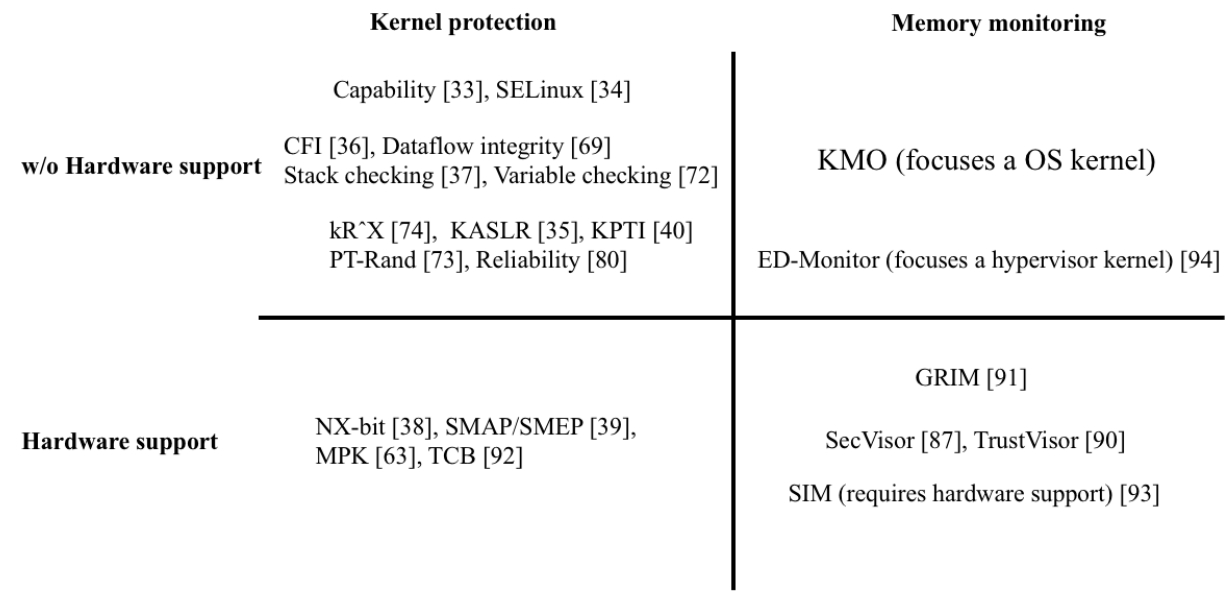

Figure 1.5 Operating system security comparison of related work.

kernel vulnerabilities, and it causes no interruption when running the kernel code. This suggests that several existing kernel security mechanisms ought to be used in coordination.

Kernel memory protection. KASLR [35] for virtual memory randomization, and KPTI or another method that separates the virtual memory space between the user and the kernel, can reduce the effects of attacks on the kernel memory [40, 75]. Moreover, virtual memory protection methods separate the memory space by the domain and granularity of memory access control [76, 77, 78, 79]. Additionally, the separation of the device driver code from the kernel provides granularity monitoring points [80, 81].

Kernel attack. Several attack concepts target the kernel virtual memory [40, 82, 83] to evade the above security mechanisms by the side channel attack. The kernel attack method uses both return oriented programming and anti-CFI [84], whereas the direct mapping space method can execute the attack code only in the kernel mode [32, 85]. The device driver has a directory threat surface [86]. Kernel virtual memory monitoring is essential to mitigating these attacks in kernel mode.

Memory monitoring (vitalization). Memory monitoring mechanisms are run under the kernel layer and are unaffected by kernel vulnerability. Kernel monitoring mechanisms have a hypervisor, and a secure mode has been proposed [61, 87, 88, 89]. In particular, SecVisor [87] and TrustVisor [90] ensure that only the verified kernel code is running.

Table 1.14 Kernel monitoring feature comparison (✓is supported; △ is partially supported).

Feature	**SecVisor** [87]	**SIM** [93]	**ED-Monitor** [94]	**KMO**
Memory Corruption Detection	✓		✓	✓
Memory Corruption Protection		△		△
System Call Argument Inspection		✓		✓
In Kernel Interception		✓	△	✓
Kernel Integrity	✓		✓	△
Cloud Environment Deployment		△	△	✓

Memory monitoring (hardware). GRIM also has a verified kernel code at the GPU layer [91], and Trusted Computing Base [92] verifies the integrity of the kernel code at the boot sequence.

Memory monitoring (in-kernel). Monitoring of the same layer as the OS or hypervisor has a low overhead when hardware assistance is available [93, 94, 95].

The summarized of comparison between related works and KMO's monitoring feature that resides in the kernel (Figure 1.5). KMO requires relatively small overhead with the existing hypervisor methods. These methods target invalid overwriting of the kernel code or privilege variable in the kernel virtual memory with hardware assistance. These are effective methods to reduce or trigger kernel monitoring, as needed. Additionally, the evaluation of the KMO mechanism's capability to protect existing kernel vulnerabilities and maintain stable operation on the system, and the co-operation with other security mechanisms (e.g., SELinux, KASLR, and SMAP / SMEP) should be undertaken.

Comparison with Related Work

The security features of KMO have been compared with those of three existing kernel-monitoring methods (Table 1.14) [87, 93, 94]. KMO satisfies almost all the identified requirements for the running kernel and the cloud environment.

SecVisor [87], which completely monitors the kernel from the hypervisor layer, intercepts device access to maintain kernel integrity; however, the inspection granularity has the limitation of being dependent on hardware assistance. Secure in VM (SIM) [93] directly inserts an alternative address space into the guest kernel from the hypervisor to monitor the kernel. The event driven (ED)

Table 1.15 Application traffic monitoring and modeling.

Category	Method
Network traffic monitoring	Attack detection [96, 97, 98, 99, 100, 101, 102]
	Signature generation [103, 104]
Network traffic modeling	Malware detection [108, 109, 110, 111]
Information leakage (host)	Framework modification [115, 116]
	Permission separation [55, 58, 59, 123]
	Policy enhancement [117, 118, 119, 120, 121, 122]
Suspicious library (static analysis)	Application analyzing [31, 54, 127, 128, 129, 130]
	Network traffic investigation [56]

Monitor [94] ensures hypervisor integrity as the same layer focuses on memory protection by having the kernel module insert a hooking placement.

Although this privilege layer monitoring approach is similar to the KMO architecture, KMO provides finer inspection points for memory protection and detection through system calls or the insertion of a kernel function flow. Finally, although KMO may struggle to set a suitable inspection point on a kernel, users can monitor kernels effectively by using a combination of existing methods. KMO helps protect device drivers or other potentially vulnerable regions by reducing the attack surface of the system.

1.5.2 Application Information Leads Leakage Detection

Network traffic monitoring. Previous work on signature generation using clustering focused on the similarity of network traffic or on the characteristics of applications, specifically targeting malware and malicious network traffic [96, 97, 98, 99, 100, 101, 102]. Other studies proposed clustering network destination and traffic separately to comprehend some aspect of an application's behavior [103], or to generate signatures for computing HTTP packet statistics to improve detection rates [104]. Other researchers proposed that probabilistic signatures might improve the detection of information leakage on Android applications [105, 106, 107].

Network traffic modeling. Analysis of applications' behavior has been used to target malware or detect bot traffic [108, 109, 110, 111]. Some papers use models of network protocol to automatically detect bot behavior or protocol vulnerabilities [112] or generate an application's

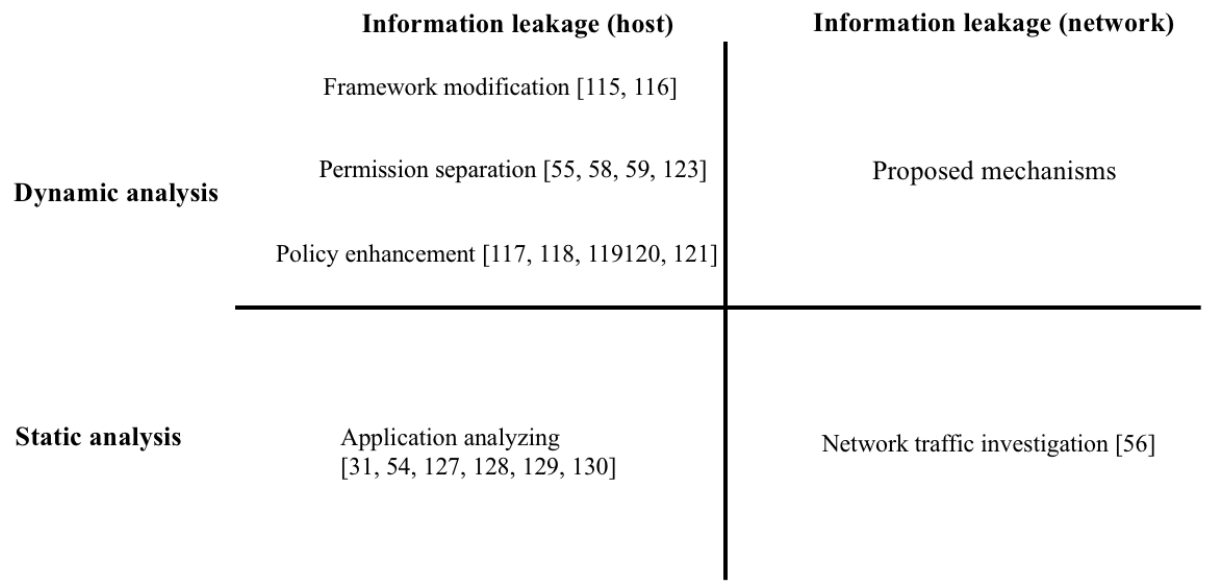

Figure 1.6 Application traffic monitoring and modeling comparison of related work.

information flow and system call sequence graphs to distinguish malware behavior on computers [113]. Modeling a sequence of ad links on malicious web sites has also been suggested [114]; this has some similarities to the present proposal.

Information leakage (host). Taint tracking can also accurately detect leakage of sensitive information and control the information flow of applications [115]. Observation of intent can be applied to finding malicious activity in applications' communication [116]. These approaches have shown that dynamic analysis of the traced details of applications' behavior on the Android framework can ameliorate the problem, and it has the advantage of having low overhead with very few false positives.

Application capability. To address information leakage in applications, fine-grained access control techniques have been proposed [117, 118]. These projects implement enhancements to the Android permission policy. Studies have suggested enhancements to the Android permission framework, and new policies that which would limit applications' behavior [119, 120, 121, 122]. Separating ad modules from applications can reduce the privacy risk [58, 59, 123] and alert users to details regarding permissions usage [124]. If permissions are not granted to the advertisement module, the user can be sure that the advertisement module does not access sensitive information on the device and send it over the network. These approaches are practical countermeasures against information leakage, but they all require Android framework modifications or application support.

Table 1.16 Application traffic monitoring feature comparison (✓is supported; △ is partially supported).

Feature	**TaintDroid** [115]	**AdSplit** [59]	**AdRisk** [55]	**Proposed mechanisms**
Excessive permission application	✓	✓	✓	✓
Host information leakage	✓			
Network information leakage				✓
Host suspicious library	✓	✓	✓	
Network suspicious library			△	✓

Privacy preserving. Privacy preserving advertisement approaches [125, 126], in which users' behavioral information is not collected from devices for targeted advertising, have also been proposed. Although these proposals reduce users' privacy risks, they are not in practical use.

Suspicious library (static analysis). Several studies have analyzed the security and privacy concerns of potentially sensitive information leakage from Android and iOS applications from suspicious ad module behavior [31, 54, 127, 128, 129]. Some work has focused on ad modules' access to device identification number and location information, and their ability to send raw or hashed sensitive information over the network [56]. Such studies have included tracking the device identification number when it appears in network traffic by analyzing over 100 ad modules in applications (AdRisk) [55] and observing the number of permissions that are required by ad modules [130].

The summarized of comparison between related works and the proposed mechanisms from analyzing method and points of information leakage detection (Figure 1.6). The proposed mechanisms generating signatures and graph of ad module traffic that can identify sensitive packets and actual ad module traffic with a small percentage of false positives, does not require any modifications to the Android framework, or any escalation of user privilege on the Android device, making the system simple and immediately applicable. It can also be used in conjunction with other methods should the proposed modifications be implemented, or with anti-virus applications designed to detect malware.

Table 1.17 Solving problems in multi-layer security in this dissertation.

Problem	Number
Detection of kernel memory corruption	Problem 1
Detection of information leakage network traffic	Problem 2
Detection of suspicious application library leads information leakage	Problem 3

Comparison with Related Work

Table 1.16 compares the security features of the proposed mechanisms to those of three existing mechanisms [55, 59, 115].

The TaintDroid [115] and AdSplit [55] approaches require Android framework modifications. AdRisk [59] attempts to identify ad modules within applications. These methods disassemble the application, or track information flow on the device. The proposed mechanisms use only the network traffic relationships to detect ad modules' activity and is thus applicable even in environments that do not allow static analysis (e.g., the large quantities of data in corporate or WiFi networks). Additionally, while there are a small number of major ad service providers, many smaller providers are unlikely to be detected by static analysis. However, the experiments indicate that manually identified ad graphs are similar to known ad graphs, and thus previously unknown modules could be detected by the proposed mechanisms using only a few known ad graphs.

1.6 Research Strategies

This section presents the research strategy in the dissertation. Target problems are summarized in Table 1.17 and Figure 1.7. It mentioned in section 1.4.1, 1.4.2, and approaches in the research.

1.6.1 Detection of Kernel Memory Corruption

This dissertation describes the design of a novel security mechanism called the "kernel memory observer" (KMO). The proposed mechanism can identify illegal data manipulation in the kernel virtual memory, which could result in the security features being defeated. KMO provides a secret observation mechanism that equips an alternative kernel virtual memory as a secret monitor of the original kernel virtual memory. Although the kernel has one kernel virtual memory at the KPTI implementation, the design of KMO is such that the kernel virtual memory is completely separated

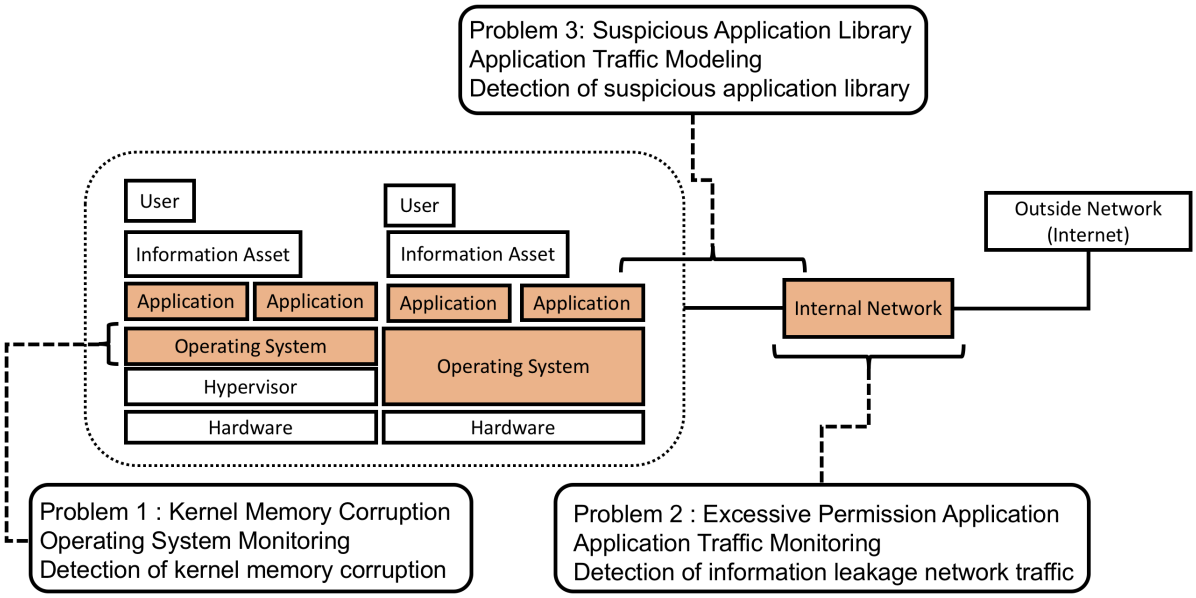

Figure 1.7 Approach of this study to the problems of multi-layer security.

to maintain its secrecy, and it is responsible for kernel monitoring, code execution, and valid data storage.

More specifically, KMO controls a virtual memory switching function that changes the kernel virtual memory space to the secret virtual memory space at various times during monitoring. KMO aims to prevent two scenarios: (i) malicious parameters at system call arguments that induce the injection of suspicious code targeting kernel vulnerabilities; (ii) a kernel vulnerability attack that overwrites the kernel virtual memory, leading to a modification of KMO monitoring code and valid data. KMO can successfully identify kernel virtual memory corruption.

1.6.2 Detection of Information Leakage Network Traffic

Various methods employing tracking-information flow and privilege separation have been used to address the problem of detecting information leakage network traffic [58, 59, 115, 120, 131]. While these approaches all successfully expose leaks of sensitive information, they all require extensive modifications to the Android framework. It should be sufficient to force applications to notify users of information usage details, thus enabling them to control the handling of sensitive information dynamically. One goal of the present work is to find a practical method, requiring no modification of the Android framework, which can identify when applications leak sensitive

information. Another is to create a system where users can easily manage the transmission of sensitive information by applications, thus reducing possible violations of their privacy.

More specifically, a novel method of clustering is presented, which uses selected HTTP packets to generate signatures that can accurately identify new packets containing sensitive information. The primary concern is not malware, but free applications that risk leaking sensitive information. Additionally, a system that can protect against information leakage (without Android modification) by generating signatures from applications' network traffic has also been proposed in a second research strategy.

1.6.3 Detection of Suspicious Application Library Leads Information Leakage

Ad modules are developed worldwide and have a variety of implementations. In order to determine what kind of ad module is included in an application, the proposed approach must be founded on an accurate comprehension of the behavior of ad modules. Static analysis of applications is the most efficient approach to this and has been widely used in ad module detection [55]. Dynamic analysis, which targets real-time behavior of the application's API calls and network traffic, is also important. With access to all the data from a corporate or WiFi network, analyzing only the network traffic should be possible in practice.

More specifically, a novel method is presented for detecting ad module network traffic intermixed with other application network traffic. The proposed approach is effective in the practical situation where only network traffic (e.g., from a corporate or WiFi network) can be monitored. In the proposed approach, a graph of ad module network traffic, extracted from identifying patterns for each HTTP session, is generated first. For example, ad modules often download an ad image using a particular sequence of HTML or JavaScript requests. This representative ad module graph can be compared, using graph distance, to the current network traffic. Graph distance measures the similarity between any two graphs, so the proposed approach can use the distance between the current and ad module graphs to determine whether network traffic is generated by the ad module or not.

1.7 Outline of the Dissertation

This dissertation is organized as follows:

Chapter 2 proposes a security architecture that provides a secret virtual memory to monitor the kernel virtual memory. The details of the design, its implementation, and the results of the evaluation are shown.

Chapter 3 proposes a clustering-based signature generation method, which uses selected HTTP packets to identify new HTTP packets containing sensitive information. The results of the evaluation are shown.

Chapter 4 proposes a detection method for ad module network traffic intermixed with application network traffic. This can be used to determine whether or not network traffic is generated by the ad module. The results of the evaluation are shown.

Chapter 5 presents the conclusions of this research, and it indicates the direction of future problems and research in this field.

Chapter 2

Detecting and Identifying Kernel Memory Corruption

2.1 Introduction

Kernel Memory Observer (KMO) prevents the modification of the secret virtual memory and monitors valid data in the direct mapping region, which contains physical memory for effective allocation or collection. KMO forces the unmapping of the virtual memory region and relates KMO information from the direct mapping onto the kernel virtual memory.

In the evaluation, KMO was able to detect the eBPF kernel vulnerability [44], the prototype of illegal kernel modules that corrupt the kernel virtual memory to bypass the SELinux security feature. In addition, KMO has a low overhead for each system call round time and for application running cost.

The main contributions of this study is summarized below:

- Designing a cutting edge security architecture, KMO, that provides a secret virtual memory to monitor the kernel virtual memory. KMO has a secret observation mechanism that provides three switching patterns between the secret virtual memory and the kernel virtual memory, whereas the unmapping method provides protection from direct mapping. Although kernel protection is being examined in multiple studies, no study has addressed the monitoring of kernel virtual memory at the kernel level. KMO provides the advantage of enhanced safety for the kernel, thereby combining the features of existing security mechanisms without virtualization. Moreover, it can be applied to the OS on a bare machine and to a guest OS on a cloud environment.

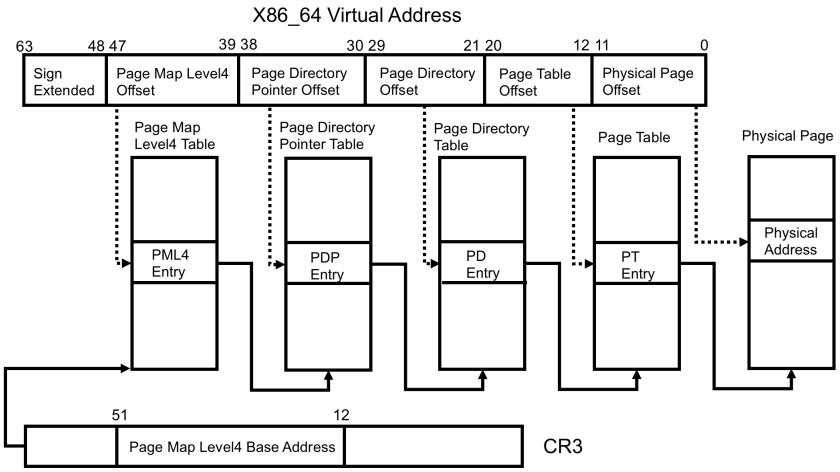

Figure 2.1 Multiple page table converts virtual address into physical address.

- Implement of KMO on the latest Linux kernel with KPTI. For the evaluation, examined KMO detection capability with regard to eBPF kernel vulnerability [44] and the prototype of illegal kernel modules that corrupt the kernel virtual memory to bypass the SELinux security feature. Additionally, KMO protects monitoring code and information from invalid access via the direct mapping region. The evaluation results for KMO revealed that its overhead is from 0.002 μs to 8.246 μs for each system call round time, whereas the application overhead is from 39.70 μs to 390.52 μs for each switching pattern for 100,000 HTTP accesses. KMO adopts the PCID of tag-based Translation Lookaside Buffers (TLBs) to mitigate these overheads and improve performance.

2.2 Background

2.2.1 Virtual Memory Management

Virtual memory is an illusion that provides domestic memory and memory isolation to running processes. It provides massive memory space on the kernel as compared to the physical memory space. Figure 2.1 shows that Linux x86_64 allows each process to have its own virtual memory space. A multiple page table converts virtual addresses into physical addresses on the virtual memory space. CR3 has a physical address of the page table of the current process, which refers to a traversal of the page table on the Memory Management Unit (MMU) and cache hit on the TLB.

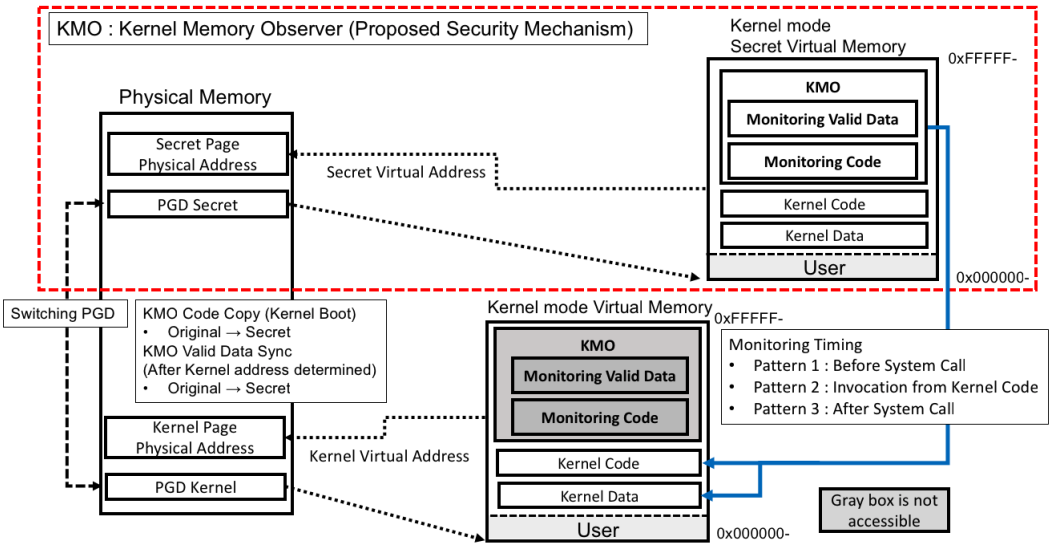

Figure 2.2 Overview of monitoring on the secret virtual memory space.

The virtual memory space layout and virtual address length differ for each CPU architecture. Linux x86_64 has a 48-bit virtual address length, which implies that the virtual address space has a size of 256 TB. The user space is 128 TB, and the kernel space is 128 TB, which contains the kernel function, data, module, direct mapping, and memory allocation address space (vmalloc, etc.).

2.2.2 Separation of Virtual Memory

Kernel and user processes share virtual memory to enable high-speed management. Virtual memory access control relies on the protection of the privilege level in the kernel and the CPU, which restricts cross access between the user and kernel modes. A meltdown attack overcomes this protection, and a user process can then easily access the kernel virtual memory through a combination of out-of-order, exception, and cache latency on side channel attacks.

One meltdown attack countermeasure involves the separation of the virtual memory used for the kernel and user modes. Here, a process has two virtual memory spaces. The OS automatically changes the virtual memory during any privilege level transition from user mode to kernel mode. The user mode virtual memory only contains a small amount of kernel code that switches to the virtual memory to minimize the access range of any meltdown attack. KPTI [40] is the separation method for Linux, and the other OSs are equipped with similar mechanisms [132].

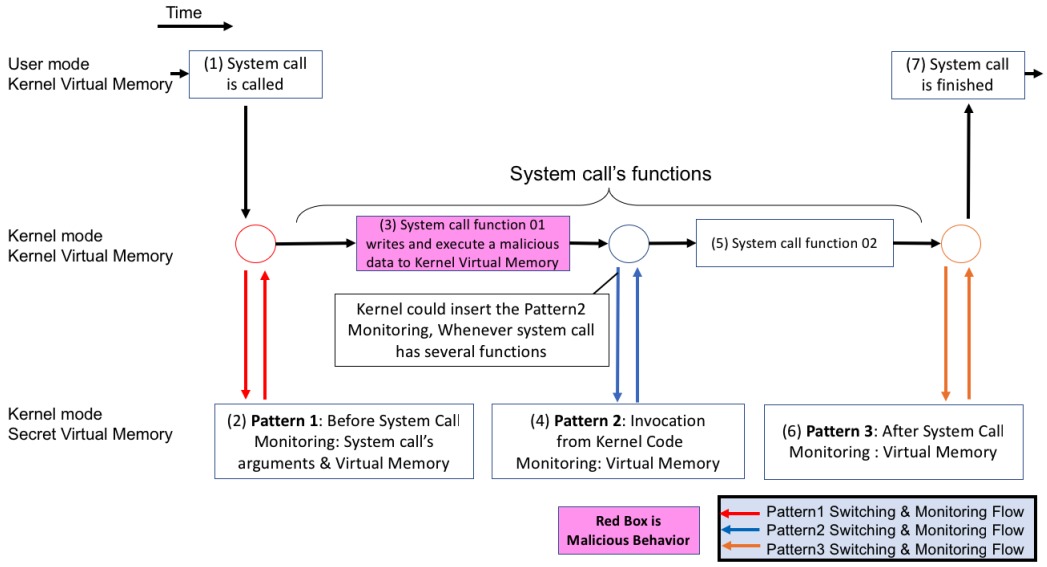

Figure 2.3 Virtual memory switching patterns 1, 2, and 3

2.3 KMO Design

Figure 2.2 shows that designed "Kernel Memory Observer" (KMO) that creates a secret virtual memory in the kernel mode. This supports the execution of monitor code in the kernel virtual memory. It is established at a different location of the kernel virtual memory management from the latest kernel (e.g., Linux with KPTI). In the KMO's kernel, it has two kernel virtual memories (i.e., original and secret).

2.3.1 Design Goal

The goal of KMO is to protect the kernel security feature code, data, and kernel module on the kernel virtual memory; then, KMO monitors these memory regions to detect invalid overwriting.

The kernel virtual memory permits reading, writing, and execution in the kernel mode, but not in the user mode. latest kernel (e.g., Linux with KPTI), which enables the isolation of virtual memories, involves the user and the kernel virtual memory for a process. Whenever a process invokes a system call processing, the kernel automatically switches both virtual memories during privilege transitions between the user mode and kernel modes. All processes share the kernel virtual memory; therefore, the kernel still provides one virtual memory space that is available for the various features at the kernel layer. Kernel memory corruption vulnerability can potentially

lead to invalid overwriting of specific kernel virtual memory region occurs only when the kernel code is running in kernel mode.

KMO creates a secret virtual memory space isolated from the original kernel virtual memory. This separation ensures that access violation is impossible between the secret and the original kernel virtual memory. KMO places the valid monitoring data and the monitoring code on the secret virtual memory, which is unaffected by the memory corruption on the kernel virtual memory. KMO generates the valid monitoring data from an original benign kernel code and data at the kernel boot. KMO then executes the monitoring code on the secret virtual memory. It verifies the kernel code and data for modification by comparing them with the valid monitoring data.

2.3.2 Switching Patterns and Detection Capability

Figure 2.3 denotes that KMO is a monitoring mechanism that adopts three virtual memory switching patterns depending on the kernel manages process transition between the user and kernel modes.

Pattern 1: **Inspection point is undertaken before the system call execution.** Pattern 1 involves inspecting whether the system call argument is suspicious data input before the adversary can execute malicious code by using the kernel vulnerability.

Pattern 2: **Inspection points during system call function or kernel code processing.** Pattern 2 inspects the kernel code and data in the kernel virtual memory. There may be inspection points having multiple functions during a system call consisting of multiple functions. Pattern 2 involves the direct detection of memory corruption in the kernel virtual memory for any timing during the kernel function flow.

Pattern 3: **Inspection point is undertaken after the system call execution.** Pattern 3 inspects the kernel code and data in the kernel virtual memory. It reliably detects memory corruption after an attack completes a system call execution.

Therefore, KMO automatically switches from every pattern for system call invocation at the kernel layer. It combines multiple inspection point of patterns at one system call invocation. Although it is effective in detecting kernel memory corruption and attacks, the number of inspections results in a significant overhead. Examination of simulated attacks on KMO mechanism to identify suitable inspection points in a running system.

Upon identifying the attack, the kernel handles the interruption of system calls by returning the error number to the user process. Additionally, the kernel is considered available to fix the modified memory region.

2.3.3 Design Approach

KMO overcomes three challenges facing the monitoring of kernel virtual memory in kernel mode.

Challenge 1: The monitoring code has access permission for monitored data and will be executed in the secret virtual memory. KMO has three virtual memory switching patterns with different inspection points on a running kernel with system calls. The inspection timing at which memory corruption is detected is also differs for each switching pattern. It efficiently monitors the already implemented kernel security feature and the module space in the kernel virtual memory to detect memory corruption attacks.

For virtual memory switching, KMO writes the physical address of the multiple-page table of the secret virtual memory into a specific register (i.e., CR3 register points to the page table for x86_64). The monitoring code executes in the secret virtual memory space. After monitoring, the KMO writes the physical address of the original kernel virtual memory into a specific register (i.e., the CR3 register for x86_64), and then continues the processing of the kernel code before the switching event occurs.

Challenge 2: The kernel code cannot access the secret virtual memory space.
KMO fully copies the secret memory space from the original one such that both memory spaces contain the same kernel code, kernel data, monitoring code, and monitoring data.

The monitoring code and valid monitoring data are not accessed through the page table flag management for the original kernel virtual memory. Therefore, in kernel mode, the original and secret virtual memory are completely isolated in KMO, ensuring that the kernel code acts on the original kernel virtual memory space by using its virtual addresses. Furthermore, it ensures that the monitored kernel code cannot access the kernel mode secret virtual memory space.

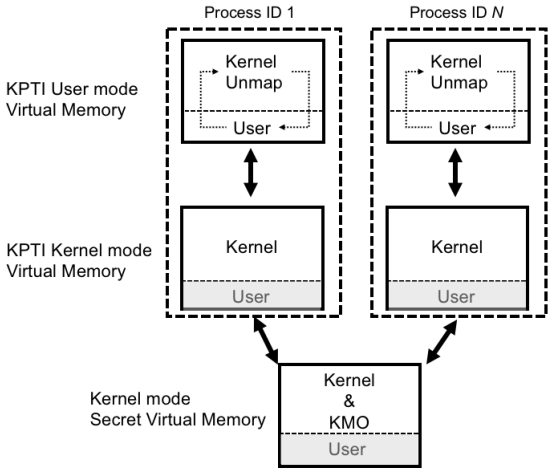

Figure 2.4 Overview of secret virtual memory space for Linux kernel.

Challenge 3: The monitoring code and valid data are not affected through a direct mapping space. The kernel virtual memory management provides a direct mapping space containing the physical memory for effective page-based memory allocation and collection. KMO shares the physical memory between the kernel and the secret virtual memory, which can be abused by allowing direct access to the monitoring code and valid data KMO modifies the allocation mechanism of direct mapping to prevent memory corruption via the direct mapping space. To exclude the monitoring code and valid data from the direct mapping of the kernel virtual memory, KMO forces the unmapping of these in the kernel virtual memory.

2.4 KMO Implementation

Implementation of KMO is on Linux as the target OS and x86_64 as the CPU architecture.

2.4.1 Secret Virtual Memory Space Management

KMO can monitor the kernel virtual memory (Figure 2.4). The latest Linux kernel has the KPTI feature, which already provides each process has 2 virtual memory spaces.

For the kernel, the `pgd` variable of `init_mm` in `mm_struct` points to the physical memory address of the kernel virtual memory. KMO creates an additional virtual memory space on the

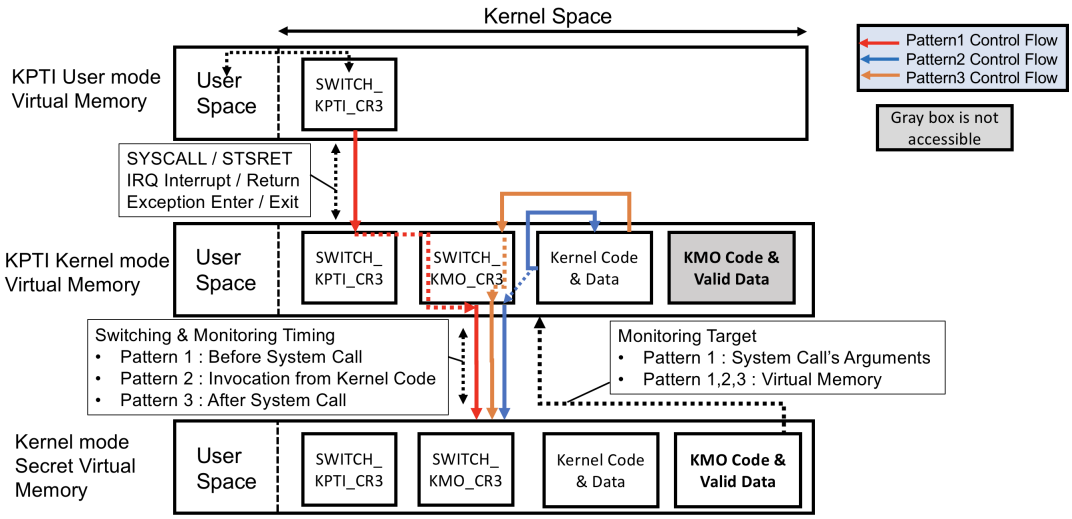

Figure 2.5 Virtual memory space switching on Linux kernel.

kernel whose physical address is a 4 page (16 KB on x86_64) logical conjunction from the physical address of the `pgd` variable of `init_mm`. Moreover, the kernel code and data are duplicated from the `pgd` variable. KMO uses the physical address of the created virtual memory to switch from the kernel virtual memory to the monitoring of each process in the kernel mode.

2.4.2 Switching of the Virtual Memory Space

Implementation of KMO is to provide a virtual memory switching mechanism for the secret virtual memory space in kernel mode (Figure 2.5).

In user mode, Interrupt (SYSCALL, IRQ) and Exception are triggered for the transition to kernel mode. It calls the SWITCH_KPTI_CR3 function on the virtual memory of the user and then changes to the kernel virtual memory space.

In kernel mode, KMO fulfills **challenge 1**, as the kernel calls the SWITCH_KMO_CR3 function, which calculates a 4-page offset to the physical address of the secret virtual memory space from the `pgd` variable of `init_mm`. The kernel writes this value to the CR3 register, followed by automatically switching the virtual memory space for monitoring. After the monitoring process, the SWITCH_KMO_CR3 function writes the physical address of the `pgd` variable in the `active_mm` of the `current` (`task_struct`) variable to the CR3 register, which can change the virtual memory space for the currently running process in kernel mode. The kernel then calls

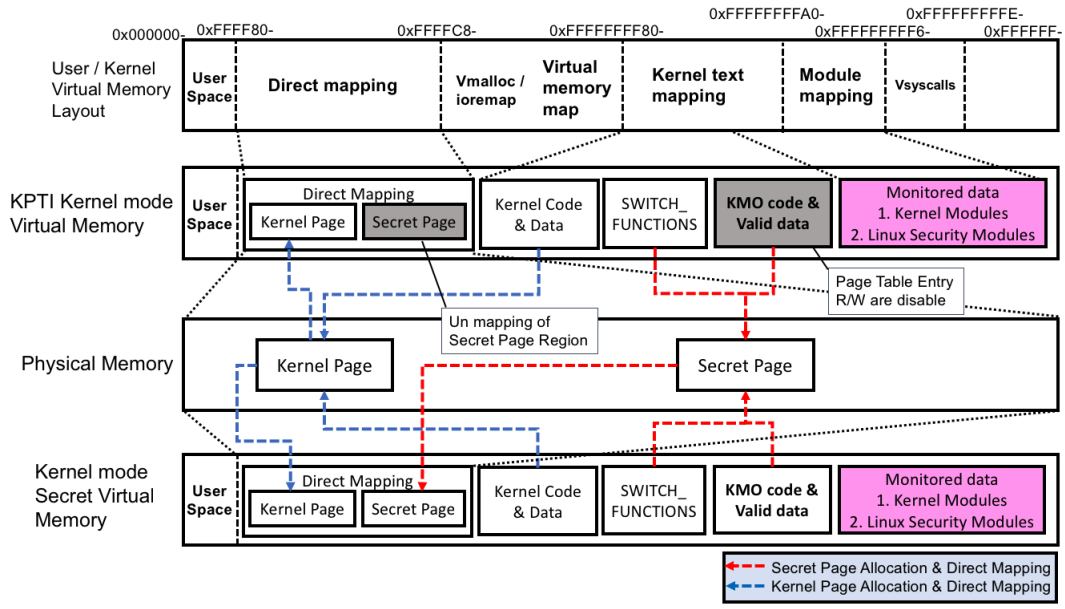

Figure 2.6 Position and unmap region for the virtual memory space on Linux x86_64.

SWITCH_KPTI_CR3 to change the virtual memory space for the user, and the system changes to user mode via an interrupt (SYSRET Interrupt return) or exception (Exception exit).

KMO currently supports the Process Context ID (PCID) during CR3 register writing. It enables the cache of TLB entry (lower 12 bits of CR3 value) by using the conversion of caches between the virtual and physical addresses on the specific CPU. If the CPU or environment does not support PCID, KMO uses TLB flush after the CR3 register writing. However, the virtual address conversion is accompanied by overheads.

2.4.3 Monitoring of Virtual Memory Space

KMO has almost the same virtual memory space layout both original and secret virtual memory on Linux x86_64 (Figure 2.6). KMO monitors the `security_hook_list` variable for LSM on the kernel text mapping and the module list variable `modules` in the kernel virtual memory. Additionally, KMO disables Copy on Write of the monitored data, whereas it supports targeted kernel space reading after virtual memory switching occurs. KMO fulfills **challenge 2** as both the monitoring code and the valid monitoring data have a designated flag setting that does not accept reading and writing at the supervisor level on the Page Table Entry.

KMO keeps the secret virtual memory space in the kernel boot sequence and then starts the

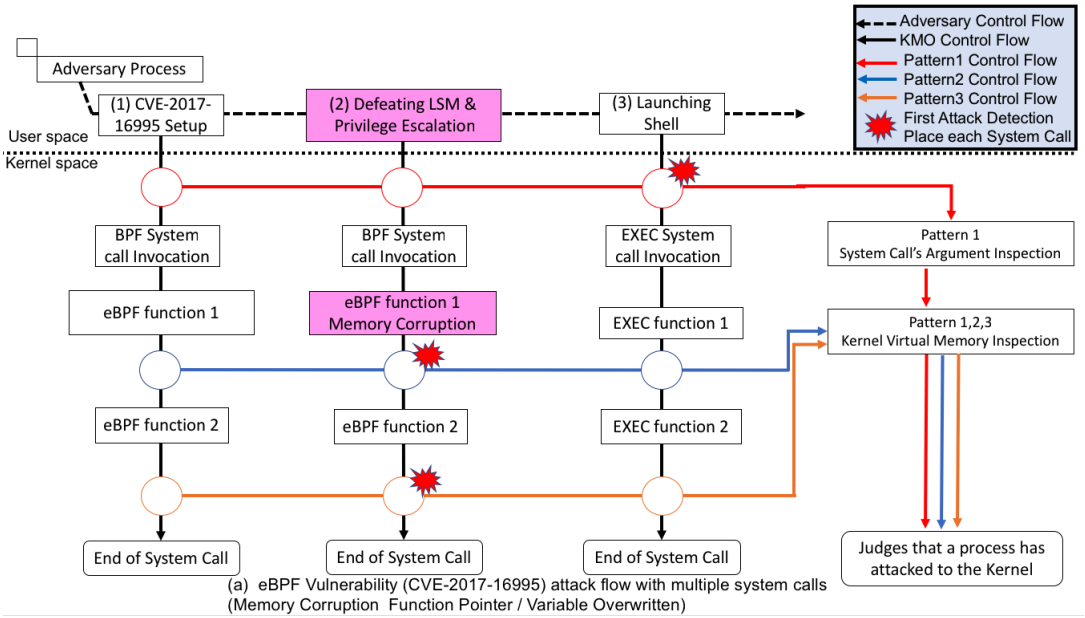

Figure 2.7 Monitoring attacker process using the secret virtual memory space on Linux.

monitoring feature according to the following sequence.

(1) The `mm_init` function initializes the kernel virtual memory, whereas the `kaiser_init` of KPTI function initializes the virtual memory for the user on the kernel boot sequence.

(2) KMO initializes the secret virtual memory in physical memory.

(3) The `security_init` function initializes the LSM and MAC mechanism.

(4) The `load_default_modules` function executes the module reading process on the kernel.

(5) KMO duplicates the valid monitoring data between the secret and kernel virtual memory spaces.

(6) KMO starts the monitoring feature in the secret virtual memory space.

2.4.4 Direct Mapping Management

Linux 4.4 (x86_64) has a direct mapping space of 64 TB. Therefore, the machine physical memory is mapped to a space of less than 64 TB, and the kernel manages physical page allocation by using direct mapping. Thus, it is possible to access the kernel code and the data virtual and direct mapping virtual addresses.

Linux uses the `init_mem_mapping` function to create the virtual memory direct mapping space. The `kernel_physical_mapping_init` function then maps the physical addresses to virtual memory. KMO covers **challenge 3**, as KMO uses the `remove_pagetable` function to unmap secret pages of the monitoring code and valid monitoring data from the direct mapping space in the kernel virtual memory after establishing a secret virtual memory setup (Figure 2.6). Any access to the unmapped pages occurs through a page fault. Subsequently, KMO registers an original page fault handler of unmapped memory region for panic processing.

2.4.5 Kernel Vulnerability Attacking Case

In one of the memory corruption kernel vulnerability cases, the adversary uses the eBPF vulnerability[44] to modify the targeted data on the kernel virtual memory. The adversary finally takes the shell as the root capability without any LSM limitation after memory corruption overrides the SELinux function pointer, as well as credential information. KMO monitors these modifications and detects the following sequences (Figure 2.7):

(1) The adversary executes the PoC code of the eBPF vulnerability with the user privilege. The PoC code inserts malicious BPF code into the kernel virtual memory via the `sys_bpf` system call. Although KMO traps the system calls, this does not lead to suspicious behavior at the time.

(2) The adversary overwrites the LSM function pointer and performs privilege escalation through memory corruption via the `sys_bpf` system call at the kernel mode. KMO also traps the issued system calls. The KMO's Pattern 2 monitoring identifies the LSM function pointer modification on the kernel control flow. It compares the `security_hook_list` variable with the monitoring data containing valid data and determines whether the monitored data is invalid. If the KMO's Pattern 2 is not invoked at kernel, KMO's Pattern 3 also traps it and identifies the same malicious behavior after the eBPF system call execution.

(3) The adversary launches the shell program from the PoC code. In KMO's Pattern 1, it traps the `sys_exec` system call and then determines whether it constitutes malicious behavior. System call arguments contain the shell program name, and memory corruption is already identified upon modification of the variable `security_hook_list`.

2.5 Evaluation

2.5.1 Evaluation Purpose and Environment

Evaluation of KMO's effectiveness in term of detection capability and overhead. The evaluation items and objectives are described below:

E1: Monitoring system call argument experiment
Evaluation of switching Pattern 1 of KMO to verify whether the target system call argument is valid before system call execution.

E2: Detection of overwriting of LSM function
Assessment of switching Patterns 2 and 3 of KMO to check whether or not they correctly identify an eBPF vulnerability PoC that modifies the LSM function's virtual address. Determining of the timing at which the attacks on the kernel virtual memory are detected.

E3: Evaluation of the prevention of access to direct mapping
Examined whether KMO prevents access to valid monitoring data in the direct mapping space after KMO unmaps secret pages.

E4: Measurement of overhead of system call interaction with KMO
Monitoring of the effect of kernel availability with KMO by switching the virtual memory space. The measurement of the overhead by benchmark software to calculate the system call latency.

E5: Measurement of the overhead of application with KMO
Measurement of the performance overhead of a user process by using benchmark software on KMO, which adopts several virtual memory switching patterns.

Evaluation of KMO on a physical machine with an Intel(R) Core(TM) i7-7700HQ (2.80 GHz, x86_64) and 16 GB DDR4 memory. The implementation targeted Linux Kernel 4.4.114 on Debian 9.0. 17 source files are modified regarding alternative virtual memory, virtual memory switching and monitoring functions, which required 1,653 lines in the Linux kernel. eBPF kernel vulnerability [44] PoC is customized for the modification of any virtual address on the kernel virtual memory. Of the evaluations E4 and E5, the comparison of the vanilla kernel with PCID (K0), KMO kernel without PCID (K1), and KMO kernel with PCID (K2). K0 supports TLB

```
// Switching to secret virtual memory and monitoring, pattern 1
1. [58.690804] target system call
2. [58.690821] system call number: 0000000000000139
3. [58.721702] module name: malicious_module
4. [58.721731] Invalid malicious_module
// Switching to kernel virtual memory, pattern 1
5. [58.727898] malicious_module: module license 'unspecified' taints kernel.
6. [58.728542] Disabling lock debugging due to kernel taint
7. [58.772438] Attack Module Init
```

(a) Monitoring of Linux System Call Arguments

Figure 2.8 Monitoring result for Linux system call arguments.

caches for user and kernel virtual memory. Although K1 does not enable a TLB cache of secret virtual memory, K2 pushes user, kernel, and secret virtual memory into TLB cache mechanism.

2.5.2 Monitoring System Call Argument

Assumption of rootkit installation. KMO monitors the module installation mechanism that uses the `init_module` and `finit_module` system calls. It inspects the kernel module binary image from the system call argument and then outputs whether the module is invalid as the detection result. In the log message, switching Pattern 1 denotes the monitoring system call as "target system call" and the invalid module as "invalid module name".

KMO correctly identifies the invalid kernel from the system call argument (Figure 2.8). The monitoring function detects invalid module names via the module binary for only 0.05 ms before the kernel executes the system call and then invokes the module initial function.

Confirmation of switching Pattern 1 yields the correct evaluation results for the monitoring and inspection of the system call argument. Although the module executes its initialization function, the module installation process is not yet completed at the time of detection in Patterns 1. This indicates that KMO interrupts the kernel code, specifically the system call invocation mechanism to determine if the validation is possible before system call processing.

2.5.3 Detection of Linux Security Module Overwrite

The original kernel module that replaces the LSM hook function. The custom eBPF vulnerability PoC forces the exchange of one LSM hook function in the `selinux_hooks` variable to the kernel module function on the kernel virtual memory. KMO stores the valid data at kernel boot. It then automatically identifies this memory corruption on switching patterns 2 and 3. These patterns

```
// Install LKM
1.  [78.654425] lkm_address_module: module license 'unspecified' taints kernel.
2.  [78.654853] Disabling lock debugging due to kernel taint
3.  [78.718498] dummy_hook_function Address Value ffffffffa0000000
4.  [78.718427] selinux_hooks[56].hook.file_permission Address ffffffff81e77c18
5.  [78.718444] selinux_hooks[56].hook.file_permission Address Value ffffffff812f3f20

// CVE-2017-16995 attack overrides LSM function address via sys_bpf()
6.  [*] attaching bpf backdoor to socket
7.  [*] UID from cred structure: 33, matches the current: 33
8.  [*] hammering cred structure at ffff88001c4399c0
9.  [*] replacing target function at ffffffff81e77c18
10. [*] credentials patched, launching shell...

// Switching to secret virtual memory and monitoring, pattern 2
11. [100.772834] Invalid lsm function is detected
12. [100.772854] Address ffffffff812f3f20    (Valid), ffffffffa0000000    (Invalid)
// Switching to kernel virtual memory, pattern 2

// Switching to secret virtual memory and monitoring, pattern 3
13. [204.010413] Invalid lsm function is detected
14. [204.010457] Address ffffffff812d5c70    (Valid), ffffffffa0000000    (Invalid)
// Switching to kernel virtual memory, pattern03

// LKM automatically outputs the target function virtual address
15. [108.769250] skpt_selinux_hooks[56].hook.file_permission Address ffffffff81e77c18
16. [108.769291] skpt_selinux_hooks[56].hook.file_permission Address Value ffffffffa0000000
```

(b) Monitoring of Linux Security Module's Function

Figure 2.9 Monitoring result for LSM function

```
// Install LKM
1.  [  143.610533] calling change_kernel_physmap_data_attack()
2.  [  143.612688] valid data 1st virtual address: ffffffff820de600 (address), ffffffff812d5c80 (data)
3.  [  143.612922] valid data pfn: 00000000000020de
4.  [  143.612989] valid data pfn phys: 00000000020de000
5.  [  143.613009] valid data pfn virtual address: ffff8800020de000
6.  [  143.613218] valid data 2nd virtual address: ffff8800020de600 (address)
7.  [  143.613234] valid data 2nd virtual address: ffffffff812d5c80 (data)
// Unmapping valid data on direct mapping, and attack via 2nd virtual address
8.  [  143.614117] BUG: unable to handle kernel paging request at ffff8800020de6005>] string.isra.4+0x65/0xd0
```

(c) Unmapping of the Valid Data Variable from Direct Mapping Region

Figure 2.10 Preventing result for modification through direct mapping.

compare the target LSM hook function's virtual address with the valid monitoring data, and then outputs the result as a log message. An invalid case is denoted by "Invalid lsm function is detected" and "Virtual Address (Invalid)" in the detection.

KMO's detection result is successful on Patterns 2 and 3 (Figure 2.9). Patterns 2 and 3 determine whether the illegal memory is overwritten after the LSM function is modified for detection.

Confirmation of switching patterns 2 and 3 determine the illegal memory corruption at suitable detection timings. Therefore, KMO has an effective detection capability for kernel vulnerability against attacks that modify the LSM function's virtual address to prevent its existence in the kernel virtual memory.

Table 2.1 Overhead of switching virtual memory space and monitoring (μs).

System call	Vanilla kernel (K0)	KMO kernel		Overhead	
		NO PCID (K1)	PCID (K2)	K1-K0	K2-K0
fork+/bin/sh	914.900	946.758	925.269	31.858	10.369
fork+execve	260.357	274.589	265.324	14.232	4.967
fork+exit	238.784	255.276	244.128	16.492	5.344
fstat	0.359	0.384	0.377	0.025	0.018
open / close	7.245	7.598	7.293	0.353	0.048
read	0.356	0.358	0.358	0.002	0.002
write	0.309	0.312	0.310	0.003	0.001
stat	2.322	2.408	2.351	0.086	0.029

2.5.4 Evaluation of Direct Mapping Access Validation

Evaluation of KMO when preventing the overriding of valid monitoring data with invalid data via the direct mapping region in the kernel virtual memory. The KMO unmaps the specific page of the valid data on direct mapping at kernel boot. Subsequently, the kernel module installation attempts to write the invalid data and output the result to a log message.

Figure 2.10 shows the log information after the unmapping process. The kernel module calculates the virtual address on direct mapping from the correct virtual address of the valid data and then accesses it. Next, the kernel issues the page request for the unmapped page having the virtual address on direct mapping. Thus, overwriting fails from the kernel module's write access via direct mapping.

2.5.5 Measurement System Call Interaction Overhead

Comparison of the Linux kernel, including KMO's mechanism, with a vanilla Linux kernel to measure the performance overhead. Adopted benchmark software, lmbench, and execute it 10 times to calculate an average score and determine whether each system call has an overhead effect.

The overhead results are the measurement switching virtual memory features. The results are the switching of the virtual memory for each system call execution (Table 2.1). lmbench shows different counts of system calls invoked for each benchmark. fork+/bin/sh has approximately

Table 2.2 ApacheBench overhead of virtual memory switching and monitoring on the Linux kernel (μs).

File size (KB)	Vanilla kernel (K0)	KMO kernel		Overhead	
		NO PCID (K1)	PCID (K2)	K1-K0	K2-K0
1	1,041.26	1,080.96	1,050.60	39.70	9.34
10	1,878.02	1,962.70	1,895.78	84.68	17.76
100	9,621.70	10,012.22	9,718.02	390.52	96.32

Table 2.3 ApacheBench overhead of virtual memory switching patterns 1 and 3 with monitoring (μs).

File size (KB)	Vanilla kernel (P0)	KMO kernel (PCID)		Overhead	
		Pattern 1 (P1)	Pattern 3 (P3)	P1-P0	P3-P0
1	1,041.26	1,051.89	1,050.01	10.63	8.75
10	1,878.02	1,892.16	1,893.10	14.14	15.08
100	9,621.70	9,711.18	9,678.76	89.48	57.06

54 invocations; fork+execve has 4 invocations; fork+exit has 2 invocations; open / close has 2 invocations; and the others have 1 invocation.

Table 2.1 shows that the overhead of the KMO (NO PCID) and KMO (PCID) versions. In KMO (NO PCID), the system calls with the highest overheads are fork+exit (8.246 μs) and fork+execve (3.558 μs). The system calls with lower overheads are read (0.002 μs) and write (0.003 μs). A kernel with KMO (NO PCID) exhibits an overhead of 0.002 μs to 8.246 μs for each system call invocation. Otherwise, KMO (PCID) has the highest overheads are fork+exit (2.672 μs), and fork+execve (1.2417 μs). The system calls with lower overheads are write (0.001 μs) and read (0.002 μs) improvement. KMO (PCID) exhibits a range of overhead from 0.001 μs to 2.672 μs for each system call invocation.

The lmbench overhead results indicate that the performance of KMO is affected by the switching virtual memory and monitoring process cost. The PCID contributes to the context switching of processes and kernel thread memory access to reduce the overhead of KMO.

2.5.6 Measurement Application Overhead

The comparison of the application overhead among the vanilla kernel, KMO (NO PCID), and KMO (PCID) kernels. Additionally, Evaluation of the effect of PCID on both kernels with switching patterns 1 and 3 Running of Apache 2.4.25 process. The benchmark software is ApacheBench 2.4. The environment includes a 100-Mbps network, one connection, and benchmark file sizes of 1 KB, 10 KB, and 100 KB. The ApacheBench calculates one download request average of 100,000 accesses to each file. The client machine is an Intel(R) Core(TM) i5 4200U (1.6 GHz, two cores), with 8 GB of memory and running Windows 8 as the OS.

The virtual memory switching patterns do not call the monitoring process because the evaluation measures the performance effect of the kernel on each switching pattern. Comparison of the KMO (NO PCID) and KMO (PCID) kernels. KMO (NO PCID) switches the virtual memory every 200 system call invocations, whereas KMO (PCID) switches it every 100 system call invocations (Table 2.2). In KMO (PCID) for Patterns 1 and 3, the monitoring function is called to evaluate the differences of the two patterns' overheads for every 100 system call invocations (Table 2.3).

KMO (NO PCID) has an overhead ranging from 39.70 μs to 390.52 μs, and KMO (PCID) has an overhead from 9.34 μs to 96.32 μs at each HTTP access. Additionally, the KMO (PCID) of Pattern 1 is from 10.63 μs to 89.48 μs, and that of Pattern 3 is from 8.75 μs to 57.06 μs at each HTTP access.

The overhead of ApacheBench depends on the total number of system call invocations in the process. The ApacheBench result shows that Patterns 1 and 3 considerably increase the overhead factor for a large file. When used on the benchmark, the overhead cost becomes relatively small at the application processing time. Consideration of Pattern 1 to require an argument inspection of register transfer cost with a high impact. Pattern 3 incurs the same overhead cost, indicating that the switching of virtual memory and the memory monitoring have a constant of inspection cost.

2.6 Discussion

2.6.1 Performance Consideration

Consideration of the performance overhead, whereby KMO enables the reduction of the performance overhead by using tag-based TLBs that provide an Address Space Identifier. The PCID on x86 is the cache for the virtual address to physical address conversion. The cache on

TLBs improves physical memory access without a page table walk to identify targeted page after a CR3 register update. Moreover, mitigation of overhead of KMO (PCID) relies on the number of TLB cache hits for page accesses while monitoring the secret virtual memory. Additionally, the Linux KPTI mechanism uses the PCID of TLB. Moreover, KMO (PCID) enables the storage of KPTI's caches without a TLB flush to perform a quick virtual memory switch. A virtual machine feature or cloud service may not provide the PCID of the virtual CPU, and KMO requires the performance penalty for calling the TLB flush for CR3 register updation in that environment.

Moreover, the application process has no overhead in user mode. Nearly the entire performance effect involves the switching of the virtual memory, followed by the monitoring feature in kernel mode. The overhead cost in the system call latency evaluation is identical for all types of system calls. Estimation of the actual application performance is proportional to the switching virtual memory and the monitoring process in the kernel mode after system call invocation in user mode.

The monitoring system call has a different cost for the type of system call argument. A string variable or address information has low overhead, but reading the data from the user mode has a high overhead. KMO requires nearly the same cost as an actual system call. In addition, the monitoring function's cost depends on the monitoring data size. The target kernel code or data affects the comparison process with the valid data.

2.6.2 KMO Detection Capability

Kernel vulnerabilities that enable privilege escalation have two effect types in the kernel layer. One type induces memory corruption on the kernel virtual memory (e.g., eBPF vulnerability [44]), whereas the other type does not create any kernel memory side effects (e.g., Dirty COW vulnerability [133]). If an adversary attempts to gain full administrator privilege of the OS, kernel memory corruption vulnerability is high priority to execute on attack scenarios for the defeating of security features. KMO provides a combination of switching virtual memory patterns having different inspection timings. Its feature detection capabilities compensate for the memory corruption of kernel vulnerability attacks for the kernel mode. During the evaluation, the eBPF vulnerability attack overwrites the SELinux functions' virtual address of the LSM hook variable that was automatically detected on KMO for protecting the security features on the kernel

Moreover, KMO identifies an attack code starting point from the user space and kernel space via using multiple system calls for the prevention of kernel vulnerability attacks leading to memory corruption. At an actual attack detection point, Pattern 1 determines the attack before system call

execution on the kernel and prevents memory corruption. Although Pattern 2 identifies memory corruption, it interrupts the kernel execution flow of multiple functions having one system call. The user inserts a suitable detection point to reduce the effect of the kernel vulnerability attack. Preventing the execution of malicious code on one system call invocation for Pattern 3 is difficult because its checkpoint is just before switching back to the user mode. Therefore, Pattern 3 reliably detects memory corruption during kernel processing for multiple functions.

2.7 Conclusion

Most kernel vulnerability attacks that focus on kernel virtual memory corruption aim to perform privilege escalation or defeat security features. The OS kernel should reduce the effect of the attack on the adoption capability and MAC restrict privileges. Although kernel adopts KASLR, CFI, KPTI, and SMAP / SMEP of CPU features to prevent kernel vulnerability attacks for memory corruption occurs privilege escalation or the avoidance of security features, kernel have been potential to compromise at only kernel layer. The proposed KMO is contributed as follows.

- The KMO provides secret observation mechanism for monitoring of the original kernel virtual memory. To determine invalid kernel virtual memory overwriting, KMO supports multiple inspection points, identifies malicious system call arguments, and prevents attacks through the direct mapping region.

- On the evaluation of Linux with KMO that successes to inspect system call arguments and identify the memory corruption of security features. The performance overhead from 0.002 μs to 8.246 μs in terms of each system call invocation on KMO kernel. The web application overhead for KMO monitoring from 39.70 μs to 390.52 μs at the running process. Additionally, the adoption of PCID on TLB for KMO, which reduces the overhead requirement of KMO, to adopt the actual load effect for monitoring and virtual memory switching.

Chapter 3

Detection of Sensitive Information Leakage in Android Applications

3.1 Introduction

In many cases, malware is detected by the anti-virus software. Various other methods have been proposed, such as detecting anomalous traffic generated by malware [96, 97, 98, 99, 100, 101, 102, 103, 104]. However, free software that allows sensitive information leakage is not malware (and thus is not discovered by anti-virus software) but still presents a threat to the user's privacy. Furthermore, sensitive information leakage may show different network patterns than malware, and thus sensitive information leakage detection in Android applications should be handled separately from malware detection.

Of evaluation, 1,188 free Android applications are analyzed from the Top 100 and the Recent Uploads lists in Google Play Japan and collected 107,859 HTTP packets that these applications generated. Examination of the number of HTTP packets that included sensitive information and sent it to outside servers using string matching. Considered sensitive information to be unique device identifiers (UDIDs) (Android ID, IMEI, IMSI and SIM Serial ID), hashed UDIDs (MD5, SHA1), and carrier names. In this trace, 23,309 HTTP packets contained such information. Some advertisement modules obfuscate or hide sensitive information in packets. Therefore, string matching is not sufficient for detecting all information leakage. After sensitive packets had been identified, the clustering is applied to a sample of the refined data to generate signatures (individual device is information is, of course, unique to each device, and thus not part of matched string used

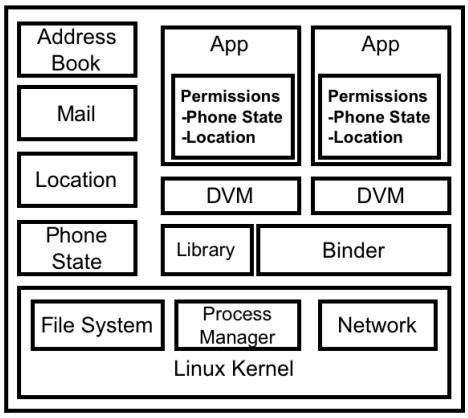

Figure 3.1 Overview of Android architecture

to generate the signature). And re-applied these signatures to the entire dataset. This method resulted in a high percentage of true positives, and a low percentage of false positives. Thus, the conclusion indicates that generated signatures have sufficient accuracy for detecting sensitive information transmissions.

The main contributions is considered following:

- A novel clustering method using HTTP packet distance that identifies the similarity between the two of Android application network packets.
- A system, using the proposed clustering method followed by signature generation, that can detect sensitive information leakage without altering the Android framework.

3.2 Background

3.2.1 Android Architecture

Figure 3.1 shows an overview of the Android architecture, which consists a Linux kernel, Middleware, Android applications, and sensitive information (Address Book, Location, Mail, and Phone State are shown). Middleware includes the Binder, the Library framework, and one Dalvik Virtual Machine (DVM) per application.

The Linux kernel provides some fundamental features for the upper layers: process management, a file system, and network services. Middleware provides DVMs, which are used

Table 3.1 Number of applications with dangerous permission combinations. Out of 1,188 applications total, 55% required both the **INTERNET** permission and at least one permission for sensitive information.

Internet	Location	Phone	Contacts	# Apps
x				302
x		x		329
x	x	x		153
x	x			148
x	x	x	x	23
				233

run applications. The Binder supports IPC and checks an application's permission list when it tries to access sensitive information via the Library. Applications on Android each have a unique Linux UID and resources have Linux UID specific permissions. This environment paradigm is called sandboxing. The application can only access resources within the bounds of its privilege permissions.

3.2.2 Android Permissions

Android provides the permission framework for managing an application's privileges. In order to access resources on Android, an application needs a specific set of permissions which link to these resources. For instance, the **INTERNET** permission allows an application to connect to any outside server using network. The **READ_PHONE_STATE** permission allows an application to access the unique device identifier and line number on the device. At the time of writing, there are 151 privileges permissions defined by Android API level 21 [134].

When an application accesses a controlled resource object, the Binder takes charge of the reference monitor to manage the application's request. The Binder verifies that the application has the appropriate permissions to bind to the requested resource.

3.3 Problem Description

In this section, the explanation that is how particular combinations of application permissions can allow a violation of user privacy. Analysis of the network traffic of 1,188 free applications – how many servers are connected to by an application; what, if any, sensitive information is included in the traffic – to show that this problem is a practical concern.

3.3.1 Application Request Permissions

Previous studies show that many applications require the **INTERNET** permission [127]. Table 3.1 shows the permissions held by collected 1,188 applications: 302 applications (25%) require only the **INTERNET** permission, while 653 applications (55%) require the **INTERNET** and some combination of sensitive information permissions. The consideration of sensitive information permissions to include **LOCATION**, **READ_PHONE_STATE**, and **READ_CONTACTS**. Those 653 applications can access sensitive resources on the device and send information gathered from those sensitive resources using the network feature, all without user confirmation, putting the user's privacy at risk.

3.3.2 Application Traffic Analysis

In fact, previous researches have been shown that some applications transmit sensitive information to external servers [31, 54]. One of the main reasons for this is that developers build an advertisement module into the free version of their applications for revenue. In order to collect statistical information of the device usage and to provide a targeted advertisement for users, advertisement modules take advantage of their ability to access sensitive information.

UDIDs are most commonly used by advertisement modules [55]. The types of UDIDs include the Android ID, the International Mobile Equipment Identity (IMEI), the International Mobile Subscriber Identity (IMSI) and the SIM Serial ID. Additionally, some modules compute UDID's hash with a cryptographic hash function at the time of transmission [56]. These UDIDs are immutable and linked to a user's real name and bank account. Unlike Internet cookies and IP addresses, UDIDs are hard (if not impossible) to change or erase, so it can be very dangerous for a user to have an advertisement module leaking his UDIDs. If an advertisement module generates a UDID's hash value from only a UDID, the hash value is same all the time, thus the

user cannot change the UDID's hash value without changing the original UDID, so the hash values have similar security problems. An advertisement module should use an application's unique user ID value (e.g., UUID value) rather than its UDID. If UUIDs (which are mutable) were used instead of UDIDs, harvested information would be restricted to the transmitting application, and the user would have the ability to alter his device's ID if he were concerned by the accumulation of information.

Selected 1,188 free applications from the popular ranking in Japan's Google Play from January to April, 2012. Since many users choose their applications from this ranking, consideration of this to be a good sample of applications used in Japan. Investigated the network traffic generated by these applications. The applications sent 107,859 GET / POST HTTP packets. The experiment environment was a Nexus S, Android 2.3.6.

Table 3.2 shows the number of HTTP packets destined for the most common hosts and the number of applications that send to each destination domain. Note that many applications send HTTP packets to the same destinations, and that some of these domains, such as "admob.com" and "ad-maker.info", are clearly advertisement services. Other domains are Web API service providers. Many of investigated applications send information to advertisement servers. During this experiment that several applications have multiple advertisement modules (e.g., AdMob, AdMaker, Adlantis, and MicroAd). Suspecting of applications switch from one module to another, depending on the user's device environment, such as country or carrier, to improve the revenue.

Figure 3.2 shows the cumulative frequency distribution of HTTP host destinations of investigated applications. From this, confirmed most of the targeted applications connect to multiple servers. From examination of the HTTP host destinations, 81 applications (7%) have 1 destination, 885 applications (74%) have up to 10 destinations, and 1,006 (90%) application have up to 16 destinations. The average number of destinations was 7.9. One application included an embedded browser, and thus had the largest number of destinations at 84.

Table 3.3 shows the number of HTTP packets, applications, and HTTP host destinations that are touched by sensitive information, where sensitive information is considered to be: UDIDs (IMEI, IMSI, SIM Serial ID, and Android ID), UDIDs' hashed values, and carrier name. IMEI refers to the assigned device number, IMSI to the assigned telephone service subscriber number in the SIM card, SIM Serial ID to the assigned SIM card number, and the Android ID to the assigned Android instance number, which is generated at Android's initial boot. The Android ID is the most frequently used identifier. Many examples of sensitive information being sent to the same

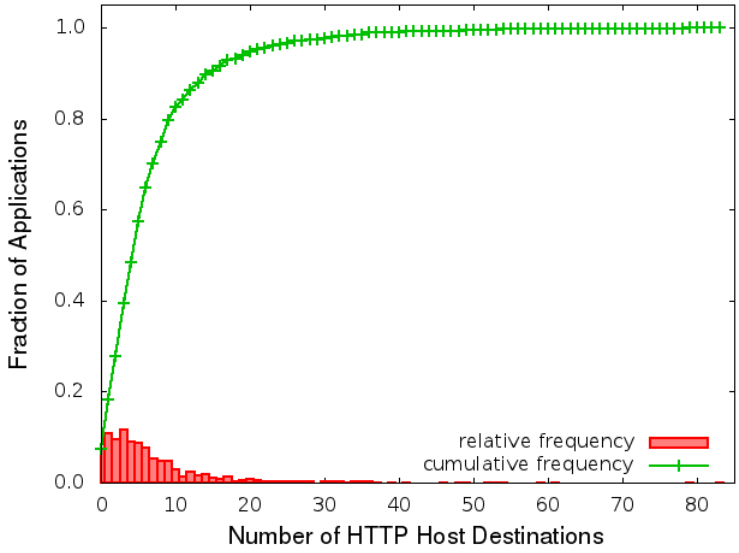

Figure 3.2 Frequency Distribution of HTTP Host Destinations. Out of 1,188 applications total, 81 (7%) have 1 destination, 885 (74%) have up to 10 destinations, and average number of destinations was 7.9.

destination. For example: “ad-maker.info”, “mydas.mobi”, “medibaad.com”, and “adlantis.jp” expect IMEI and Android ID; “zqapk.com” expects IMEI, SIM Serial ID, and carrier name; and “googlesyndication.com” and “admob.com” expect only Android ID.

From these results, the user’s sensitive information is accessed by applications which send it to outside servers via the network. Since Android does not provide the usage history of runtime applications’ permissions, the users cannot observe the application’s network behavior, and thus cannot prevent the sensitive information leakage.

3.4 Approach

Presented the following HTTP packet clustering and signature generation methods to address the problem described in Section 4.3. The objective of proposed approach is to without an Android framework modification, detect suspicious network behavior, specifically the transmission of sensitive information by an application to an outside server. Additionally, proposed system should be practical and lightweight for users to apply. Ideally, users would install the application component of proposed system to handle all the network transmissions generated by other

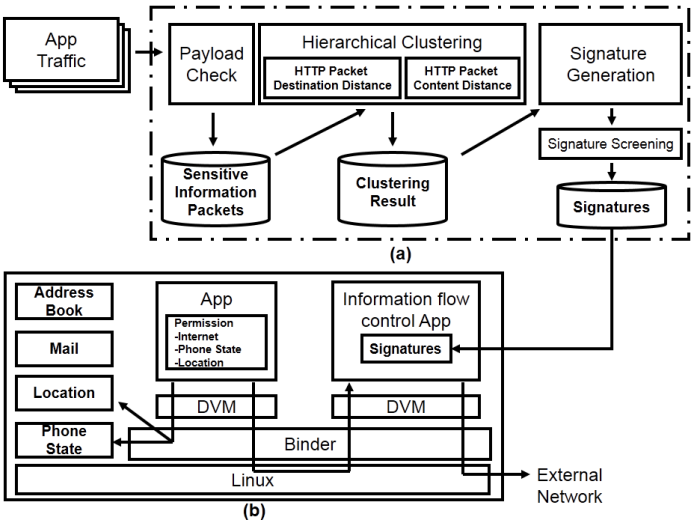

Figure 3.3 (a) The architecture of proposed clustering and signature generation system. (b) The information flow control application that uses the signatures generated by (a).

applications. Proposed approach is to collect network traffic and generate signatures from the clustering of the traffic. If sensitive information is sent unencrypted over the network, it is a fairly simple matter to detect such transmission, and signature generation can help to counteract leakage even when sensitive information is nondeterministically altered or obfuscated by an application's Software Development Kit (SDK) such as the Apperhand [135]) which is an advertisement SDK acted unwanted behavior for user. It is also effective against encrypted traffic that uses the same encryption key over a variety of modules or applies a cryptographic hash function to sensitive information.

3.4.1 Overview

Figure 3.3 shows proposed approach, which consists of two parts. First, a separate server (shown in Figure 3.3a) collects application traffic, clusters the data, and generates signatures. Second, an information flow control application on the user's device (shown in Figure 3.3b) fetches signatures from the server and manages the transmission of other applications' network traffic.

The server generates signatures by the following process. First, it performs a payload check, which separates application network traffic into two groups: one containing packets with sensitive information, and the other not. Second, the server clusters the group containing sensitive

information based on packet destination distance and content distance. Finally, it constructs a set of signatures from the clustering result using conjunction signatures [105], then sifts out overly general signatures and the actual sensitive information values, which assumption is to be device or advertisement service provider specific, and thus not useful for identifying future leakage. The screening process is most effective when the seed packets contain accurate patterns of sensitive information leakage, and proposed clustering and signature choices reflect that. Using the HTTP packet distance emphasizes patterns in HTTP packets, allowing us to distinguish trends and distributions of HTTP packets. Thus a packet with sensitive information will be clustered with other packets containing the same sensitive information, generating a useful signature. For that reason, definition of distance to include both packet content and packet destination. This broader definition causes packets sent to the same server to be clustered together, creating advertisement module specific signatures.

The information flow control application inspects network traffic using the Android API VpnService on Android 4.0 and later, which does not require any special privileges. On Android 2.3 special privileges for iptables are required for packet inspection.

3.4.2 HTTP Packet Destination Distance

The HTTP packet destination distances are calculated by the packets' destination IP addresses, port numbers, and HTTP host domains. Given two HTTP packets p_x and p_y, definition of the HTTP packet destination distance to be:

$$d_{dst}(p_x, p_y) = d_{ip}(p_x, p_y) + d_{port}(p_x, p_y) + d_{host}(p_x, p_y).$$

Let HTTP packet p_n destination be defined as $p_n = \{ip_n, port_n, host_n\}$, where ip_n is a destination IPv4 address, $port_n$ is the port number, $host_n$ is HTTP host. The distance in the above equation is defined as follows:

- Destination IP Address Distance: The distance between destination IP addresses' high bit is the longest matching prefix of the binary representations. IPv4 addresses have a 2^{32} bit space, and IP address blocks are denoted approximately by the upper 8 bit range. IP address blocks are allocated to organizations by the National Internet Registry and if the upper bits of IP addresses match on separate packets, there is a high possibility that the two destinations are managed by the same organization. Therefore, definition of the destination IP address

distance on packets p_x, p_y as

$$d_{ip}(p_x, p_y) = lmatch(ip_x, ip_y)/32 \in [0, 1]$$

where *lmatch* is a function returns a number of common upper bits in two IP address.

- Port Number Distance: The distance between port numbers is a Boolean: matching or not. Port numbers have a 2^{16} bit space, and usually specific port number is reserved for services. Definition of the port number distance on packets p_x, p_y as

$$d_{port}(p_x, p_y) = match(port_x, port_y) \in \{0, 1\}$$

where *match* is a function returns 1 on matching ports, and 0 on different ports.

- HTTP Host Distance: definition of the HTTP host as the character string of the Fully Qualified Domain Name (FQDN). Thus, the distance between HTTP host domains can be computed using the generality method to determine their edit distance. Definition the HTTP Host distance on packets p_x, p_y as

$$d_{host}(p_x, p_y) = \frac{ed(host_x, host_y)}{max(len(host_x), len(host_y))} \in [0, 1]$$

where *ed* is a function which returns an edit distance result, *len* is a function which returns a length of character strings, and *max* is a function which returns the greater of its two input values.

3.4.3 HTTP Packet Content Distance

The HTTP packet content distance is computed using the request-line, cookie, and message-body fields of the HTTP header. Given two HTTP packets p_x and p_y, definition of the HTTP content distance $d_{header}(p_x, p_y)$ as

$$d_{header}(p_x, p_y) = d_{rline}(p_x, p_y) + d_{cookie}(p_x, p_y) + d_{body}(p_x, p_y).$$

Let HTTP packet p_n contents be defined as $p_n = \{rline_n, cookie_n, body_n\}$, where $rline_n$ is request-line, $cookie_n$ is cookie, $body_n$ is message-body. These contents are character or binary strings. In order to accurately compute a distance, the applying of the normalized compression distance (NCD) algorithm [136], which is based on Kolmogorov's complexity, to calculate the

closeness of two strings without any context dependency. The NCD of any two character strings is defined as

$$ncd(x, y) = \frac{C(xy) - min(C(x), C(y))}{max(C(x), C(y))}$$

where $C(x)$ is a function which compresses a character string x, then returns its length. Definition of the distance between content components of HTTP packets p_x, p_y as

$$d_{data}(p_x, p_y) = ncd(data_x, data_y) \in [0, 1]$$

where *data* corresponds to request-line, cookie, and message-body respectively. After each *data* has been computed, they are combined into the overall distance.

3.4.4 Hierarchical Clustering

Hierarchical clustering uses group averages for iterative calculation and computes the proximity of clusters with HTTP packet distance (HTTP packet destination distance and HTTP packet content distance) as a heuristic. It then assigns a cluster to each HTTP packet, and iteratively composes new clusters from the nearest distance of HTTP packet pairs until there is only one cluster. Given two HTTP packets p_x and p_y, definition of the HTTP packet distance as

$$d_{pkt}(p_x, p_y) = d_{dst}(p_x, p_y) + d_{header}(p_x, p_y)$$

using the formula from sections 3.4.2 and 3.4.3 to compute d_{dst} and d_{header}. Given two clusters C_x and C_y, group average distance is defined as

$$d_{group}(C_x, C_y) = \frac{1}{|C_x||C_y|} \sum_{p_x \in C_x} \sum_{p_y \in C_y} d_{pkt}(p_x, p_y).$$

In a dataset of N HTTP packets $\mathbf{H} = \{p_i\}_{i=1..N}$, hierarchical clustering is applied to a subset $\mathbf{P}$ of size M: $\mathbf{P}_{j,j=1..M} \subset \mathbf{H}$, using the following method:

(1) Assign each HTTP packet $p_k \in \mathbf{P}$ to cluster C_k. At the end of this step, $\mathbf{C} = \{C_k\}_{k=1..M}$ is the set of defined clusters.

(2) Chose any cluster $C_x \in \mathbf{C}$, and compute the distance to all other clusters $C_{y,y=1..M} \in \mathbf{C}, x \neq y$ using the cluster distance d_{group}.

(3) Select the cluster C_y that is the closest to C_x. Create a new cluster $C_z = \{C_x, C_y\}$ and add it to $\mathbf{C}$, then remove C_x and C_y.

(4) Repeat until $\mathbf{C}$ has one cluster.

3.4.5 Signature Generation

The generating of a conjunction signature set from the hierarchical clustering result, which is a dendrogram of the HTTP packet group. A conjunction signature set contains the invariant tokens that describe the longest common substrings (LCS) in the dendrogram. Signatures each represent a feature of the cluster. That is, they reflect sensitive information as an invariant token. Given a dataset of N HTTP packets $\mathbf{H} = \{p_i\}_{i=1..N}$ and a subset $\mathbf{P}_{j,j=1..M} \subset \mathbf{H}$ used to generate (as described in Section 3.4.4) dendrogram $\mathbf{C}$, which has the nesting structure characteristic of clusters, generating of the conjunction signature set using the following process:

(1) Select the top of cluster $C_i \in \mathbf{C}$.

(2) Compute a signature S_i as LCS of HTTP contents in C_i.

(3) Remove C_i from $\mathbf{C}$ and repeat for all clusters in $\mathbf{C}$.

3.4.6 Signature Screening

A conjunction signature set contains many general signatures such as **POST, GET, HTTP/1.1** which match most network packets. Proposed system screens signature candidates by checking whether a candidate determines most non-sensitive packets to be sensitive. Such signatures are considered too general and are removed from the conjunction signature set. If the signatures are not screened in this manner, proposed system blocks almost all applications' network traffic. Additionally, proposed system screens for patterns of UDIDs and hashed UDIDs from the conjunction signature set. Since different each devices generally have different UDIDs, these signatures are not useful when detecting network traffic.

3.5 Evaluation

3.5.1 Experimental Setup

Collection of network traffic is from 1,188 free applications running on an Android 2.3.6, Nexus S, from January to April, 2012. The application set was as described in Section 4.3. Each application was run manually for 5 to 15 minutes on the device. Attempting is to test every possible application function. Generated the network traffic manually, since it is difficult

to automatically test an application that requires user interaction such as entering passwords and other user identification, or correct screen taps for a game.

The resulting dataset of application network traffic contained 107,859 GET / POST HTTP packets. For this experiment, the dataset is manually separated into a suspicious group and a normal group. In practice, this separation task would be automatically processed by the payload checking system, using previously generated signatures. Manual separation would only be needed to evaluate the signatures' detection rate and at the initial start-up. The suspicious group consisted of packets containing sensitive information. The normal group was made up of those not containing sensitive information. Again, considered UDIDs (Android ID, IMEI, IMSI, and SIM Serial ID), hashed UDIDs (MD5, SHA1), and carrier name to be sensitive information.

In this experiment, encrypted packets were not concerned, and did not observe any applications which nondeterministically changed or obfuscated sensitive information. In addition, the dataset did not contain mail addresses, address lists or changeable parameters such as location. Therefore, this evaluation does not include nondeterministic or obfuscation signatures except the hashes mentioned above. The suspicious group consisted of 23,309 HTTP packets, and the normal group contained 84,550 HTTP packets. The details of the suspicious group are shown in Table 3.3. Selected N HTTP packets at random out of the suspicious group for signature generation and screening, where N was increased from 100 up to 600 in intervals of 100. Finally, the generated signatures is applied to the dataset in its entirely to see how accurately they could identify packets containing sensitive information.

3.5.2 Experimental Results

Figure 3.4 shows the results of this experiment. Evaluation of the percentage of true positives, false positives, and false negatives for varying values of N.

True Positive: a correctly detected packet containing sensitive information. The percentage of true positives was calculated according to the following equation:

$$\mathrm{TP} = \frac{\text{number of detected sensitive information packets} - N}{\text{number of sensitive information packets} - N}$$

There were 23,309 sensitive information packets in the dataset for this evaluation. Proposed system produced 85% true positives at sampling size $N = 100$. It grew to $> 90\%$ by $N = 200$, with the best result being 97% at $N = 600$. These results show that true positives rise as the number of signature

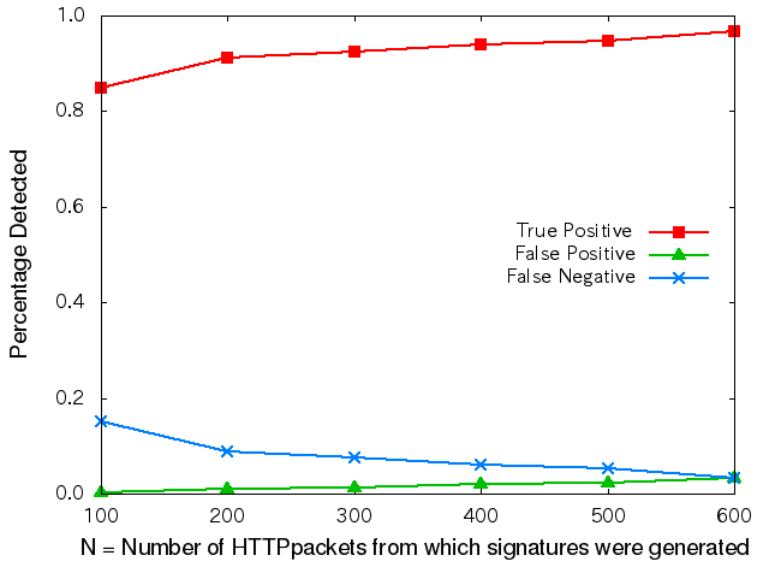

Figure 3.4 Detection Rate of Sensitive Information Leakage.

generating sensitive information packets increases, implying that signatures generated from more packets cover a wider common pattern of information leakage.

False Negative: a sensitive packet that was not correctly detected. The calculation is percentage of false negative results using following equation:

$$\text{FN} = \frac{\text{number of undetected sensitive information packets}}{\text{number of sensitive information packets} - N}$$

As stated above, there were 23,309 sensitive packets in the dataset. In this experiment, there were 15% false negatives at $N = 100$, only 8% or less for $N \geq 200$, and finally 3% at $N = 600$. Thus, effective detection of information leakage is improved by increasing the number of sensitive information packets used for generating signatures.

False Positive: a non-sensitive packet incorrectly detected as sensitive. the percentage of false positives is calculated using following equation:

$$\text{FP} = \frac{\text{number of detected non-sensitive information packets}}{\text{number of non-sensitive information packets}}$$

This value is important for an evaluation of proposed system's signatures detection rate in terms of practicality. If proposed system produces many false positives, users will be continually bothered by unnecessary warnings and prompts. This dataset had 84,550 non-sensitive information packets. The signatures from this dataset produced 0.3% false positives at $N = 100$, 0.9% at $N = 200$, and

eventually 3.2% at $N = 600$. More general signatures are generated by increasing the number of clustering packets. Lrge signature generating sets produce signatures that detect packets without relation to their information leakage.

3.6 Discussion

3.6.1 Approach Consideration

Other approaches to preventing sensitive information leakage include taint tracking and permission framework modifications. In this section, comparing of proposed approach with current research results, and discuss the limitations of proposed scheme.

These clustering methods use malware traffic with characteristic patterns for clustering. Focused sensitive information in benign traffic using novel packet distance measures. Proposed approach is applied to a real dataset to test for sensitive information leakage.

Clustering in general is useful for pulling together patterns in large amounts of data, but the number of the generated signatures tend to increase with cluster size, and can produce signatures that match most network packets (e.g., **POST ***, **GET ***, *** HTTP/1.1**), if signature generation is applied carelessly. For this reason, it has been difficult for a signatures approach to achieve high detection rates using a real dataset.

3.6.2 Complexity Analysis

the time and space complexity of the worst case for clustering and signature generation algorithms are analyzed.

Take the clustering input to be size N packets. The time complexity is $O(N^3)$, as there are N steps and at each step the packet distance matrix of at worst size N^2 must be updated and searched. The space complexity is $O(N^2)$ because the packet distance matrix must be stored.

Take the signature generation input to consist of M clusters. The time complexity is $O(M)$, as there are M steps each with a comparison of a cluster's LCS to that of its children. The space complexity is $O(M^2)$, as the matrix of the clusters' LCS must be stored.

3.7 Conclusion

Application developers adopt advertisement service revenue mechanisms. These application with regards to security and privacy risk, then Investigation of application behavior is important to protect personal information. Many Android applications permission require sensitive information access and network features, and that among them are applications that connect to many outside servers without the user's acknowledgment. Moreover, observation of applications' network behavior contains a large amount of sensitive information, particularly immutable identifiers such as UDIDs. The proposed information leakage protection method is contributed as follows.

- In order to detect sensitive information leakage from applications, proposed novel clustering method using HTTP packet distances contains both the distance between HTTP packet destinations and the distance between HTTP packet contents. For improving detection accuracy, approach uses clustering method in combination with signature generation and signature screening.

- On experimental results with dataset consists with 1,188 applications and 107,859 packets. It has 23,309 sensitive information packets, approach shown that 97% accurate detection of packets containing sensitive data with only 3% false positives.

Table 3.2 HTTP packet destinations. This table shows the number of packets sent to each HTTP host destination, and the number of applications that send packets to each HTTP host destination.

HTTP Host Destination	*# Packets*	*# Apps*
doubleclick.net	5786	407
admob.com	1299	401
google-analytics.com	3098	353
gstatic.com	1387	333
google.com	3604	308
yahoo.co.jp	1756	287
ggpht.com	940	281
googlesyndication.com	938	244
ad-maker.info	3391	195
nend.net	1368	192
mydas.mobi	332	164
amoad.com	583	116
flurry.com	335	119
microad.jp	868	103
adwhirl.com	548	102
i-mobile.co.jp	3729	100
adlantis.jp	237	98
naver.jp	3390	82
adimg.net	315	72
mbga.jp	1048	63
rakuten.co.jp	502	56
fc2.com	163	52
medibaad.com	1162	49
mediba.jp	427	48
mobclix.com	260	48
gree.jp	228	45

Table 3.3 Sensitive Information. This table shows for each type of information considered sensitive, the number of packets containing the information, the number of applications that send those packets, and the number of destinations to which those packets go.

Sensitive Information	# Packets	# Apps	# Destinations
ANDROID ID	7590	21	75
ANDROID ID MD5	10058	433	21
ANDROID ID SHA1	1247	47	12
CARRIER	2095	135	44
IMEI (Device ID)	3331	171	94
IMEI MD5	692	59	15
IMEI SHA1	1062	51	13
IMSI (Subscriber ID)	655	16	22
SIM Serial ID	369	13	18

Chapter 4

Detecting and Characterizing of Mobile Advertisement Network Traffic

4.1 Introduction

Advertisement (ad) modules access sensitive user data and send this information to outside servers using the device's network connection [31, 54, 56]. If the user does not have the opportunity to accept or deny the ad module behavior, this is a privacy violation. Grace et al showed that more than 100 ad modules distribute users' sensitive information [55].

Examined the applications' network traffic, and found that 797 applications included 45 known ad modules [31, 54, 55, 56]. These have characteristic network traffic patterns for acquiring ad content, specifically images. In order to identify ad modules' network traffic, a novel method based on the distance between network traffic graphs mapping the relationships between HTTP session data (such as HTML or JavaScript). The similarity between the sessions that derives a detection of ad modules' traffic by comparing session graphs with the graphs of already known ad modules.

Of proposed approach that converts the network traffic into graphs. The result shows that dataset consisted of 4,698 graphs from ad network sessions (called ad graphs), and 16,205 graphs of other network sessions. Using 1,000 randomly selected known ad graphs as training set, proposed approach were able to identify 76% of the other known ad graphs. This training set does not includes one vertex graphs. Since some ad modules have different network patterns, a real world system would need to carefully select ad graphs that covering ad modules' network

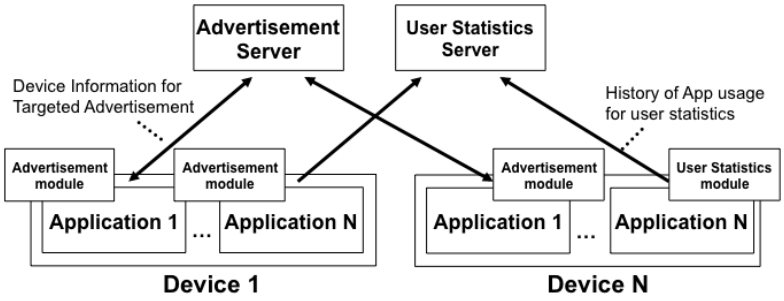

Figure 4.1 An overview of an ad modules' network behavior. Application bundles include ad modules, which connect to their suppliers' servers to download ad images or provide user statistics.

behavior. Evaluation of proposed approach using 6,093,682 packets generated by 1,188 free Android applications from the Top 100 and the Recent Uploads lists in Google Play Japan. Known ad graph subset to 2,000 other session graphs (referred to as candidate ad graphs) and 2,000 graphs from standard application traffic (referred to as standard graphs) which were manually identified as representing as yet unknown ad modules network traffic. In this evaluation, proposed method accurately detected 96% of candidate ad graphs and under 10% false positive rate of standard graphs. Additionally, the examination of an effect of one vertex graphs in training set for detection rate. In the result, training set without one vertex graphs shows improved detection rate 6.7% of known ad graphs and 3.5% of candidate ad graphs.

The main contributions is considered to be:

- A method for generating graphs of HTTP sessions that can be used to show relationships between sessions.
- A process, using this method, for detecting ad module traffic even when it is interleaved with normal application traffic.

4.2 Background

4.2.1 Permission Framework

Android provides a permission framework for an application privileges management. Permissions are linked to specific resources and an application needs the appropriate permission

to access a specific resource. At the time of writing, there are 151 permissions defined by Android API level 21 [134].

4.2.2 Advertisement Modules

Figure 4.1 shows an ad module's network behavior. In general, an ad module is an SDK provided by the ad service provider, and is executed as part of the application. During execution, an ad module connects to its provider's server.

Ad modules usually provide the following three features. First, an ad delivery service that periodically gets ad images and displays them on the user's device. Second, a mediation service that combines multiple ad delivery services. Mediation services optimize ad processing by accessing device information. Third, ad modules collect user statistics, such as application run-time information history and device location, and periodically send this information to outside servers. Typical ad modules are Google's admob [137], Mobclix [138], and Flurry [139]. Admob uses the device identification number for targeted ads. Mobclix connects multiple ad services, in different applications, and adjusts its ad images accordingly. Flurry collects a run-time log of application behavior to generate user statistics.

4.3 Problem Description

4.3.1 Advertisement Modules Behavior

Typical ad module behavior is shown in Figure 4.2. If an application uses an ad module, the application must acquire the permissions required by the ad module. This process can build a sense of distrust for two reasons. In some cases, the ad module takes advantage of the essential permissions required by the application but not required for ad module operation. In other cases, the application does not require certain permissions, but requests these permissions on behalf of the ad module.

Many applications' permission combinations require both **INTERNET** and sensitive information (e.g., **LOCATION**, **PHONE_STATE** and **CONTACTS**) permissions in the section 3.3. The ad modules attached to these applications can access sensitive information without additional user acknowledgment, and send it to outside servers. The Android permission framework does not support a privilege separation granularity on the level of SDK, so the user

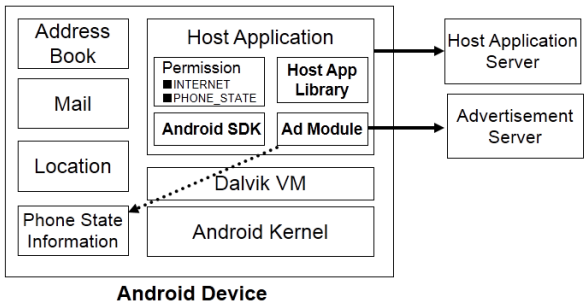

Figure 4.2 An overview of the organization and permissions of an application that includes an ad module. The ad module can use the application's permissions the access sensitive information on the device and send it over the network.

cannot limit the part of an application's permissions that applies only to the ad module SDK. Therefore, if a user notices privacy violations carried out by the ad module, she cannot restrict the application's network behavior without rewriting the application.

4.3.2 Advertisement Modules Traffic Analysis

HTTP Sessions

In order to understand ad modules' network behavior in detail, The HTTP GET requests in the network traffic data is collected and analyzed from 1,188 free applications. Table 4.1 shows the number of HTTP GET requests sent for each content type. Table 4.2 details the image sizes generally downloaded by the various content types in HTTP responses. In the HTTP GET requests are investigated, the most common request was for image files, and the second most common was for text. In HTTP GET responses, the most common response image size was 320 x 50. The images of this size, and they primarily represented ads for various services are manually reviewed (e.g., games, goods). Therefore, almost all 320 x 50 images are downloaded by ad modules.

Advertisement image downloading

The analysis of HTTP GET requests shows that applications downloading images are likely to be doing so for an ad module. Examination of how many ad modules were included in analyzed applications, and inspected each ad module's sequence of network traffic during ad image

Table 4.1 The number of HTTP GET requests for each type of with content. The most commonly requested content types (in decreasing order) are image files, script files, and document files.

Content Type	# HTTP GET Requests
jpg	13941
png	12539
gif	9197
php	5888
js	5114
css	2870
html	2093
json	841
txt	380
jpeg	282
JPG	128
Total	53273

downloads. A set of 120 already known ad modules is used for this investigation [31, 54, 55, 56]. To detect the suspicious ad modules' network traffic, these are filtered packets by the 'Host' field in the HTTP header contained a known ad module's name. This is rough selection method of ad modules' network traffic, it is manually comprehended the detail of these traffic whether is ad or not.

Table 4.3 shows the number of applications that include certain ad modules, and the methods that those ad modules used to download images. Analyzed applications' network traffic contained traffic from 45 known ad modules; and 797 applications included at least one ad module. Google's ad modules "doubleclick.net" and "admob.com" were widely used. Japanese ad modules such as "adlantis.jp" and "mediba.jp" are identified. In addition, some applications contained mediation service modules ("adwhirl.com", "mobclix.com" and "mobfox.com") and others had multiple ad modules. Analysis result shows that application ad module usage varies depending on the environment of device (e.g., country, network).

A set of rules that define the process of downloading an ad image is identified. The HTTP sessions consist of HTML, JavaScript, and HTTP redirects. An HTTP redirect that causes data to be fetched from a different URL, HTML and JavaScript is effective for ad delivery, because they

Table 4.2 Content types banner image sizes in HTTP GET responses. Banners of size 320x50 are most commonly downloaded by ad modules. Banners of size 300x48 are the next most common.

Content Type	Image Size			
	320x48	320x50	300x48	300x50
jpg	57	278	2	4
png	271	870	0	59
gif	248	1197	0	115
jpeg	25	249	0	20
Total	601	2594	2	198

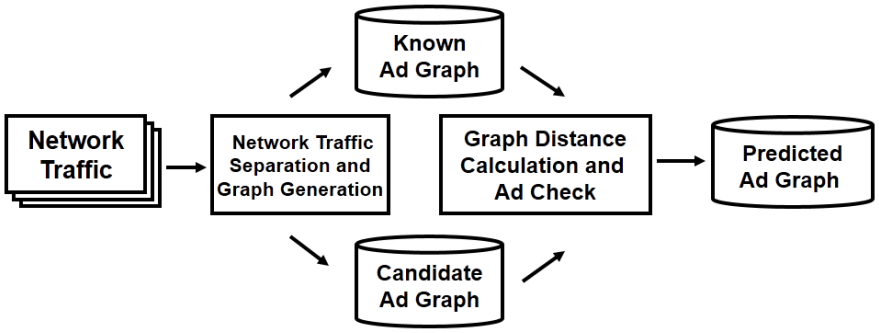

Figure 4.3 An overview of proposed approach. First, the known ad module network traffic is separated out. Next, ad graphs from the remaining network traffic is extracted by comparing the candidate graph distance from ad graph. Finally, new ad graphs is predicted.

allow dynamic control on the server side: an ad module can renew an ad image on the user's device at any given point in time, and different redirections allow different ad images to be displayed for each user without having to update the ad module. HTTP redirects and transitions to other domains used as load balancing method by the Content Delivery Networks (CDNs) that generate ad images, or provide mediation services for optimizing ad revenue, depending on the device location.

4.4 Approach

In this section, presented methods for: generating graphs from network traffic, and for calculating the graph distance between two such graphs. Discussion of how graph distance can be used to detect ad modules' network traffic as described in Section 4.3. The objective is to detect

Table 4.3 The number of applications that include ad modules, and the methods those ad modules used to download images. The total number of applications exceeds the number of analyzed applications, because some applications contain more than one ad module.

No.	Ad Module	# App	Ad Path			
			HTML	JavaScript	HTTP Redirect	Other Domains
1	doubleclick.net	397	x			
2	google-analytics.com	351				x
3	admob.com	337		x		
4	googlesyndication.com	244				x
5	flurry.com	110				
6	adlantis.jp	78		x		
7	adwhirl.com	57	x		x	x
8	medibaad.com	49				x
9-11	inmobi.com, inmobicdn.net, inmobi-jp.com	44		x		x
12	mediba.jp	30		x		
13	mydas.mobi	24	x			
14	airpush.com	19				
15	mobclix.com	17		x		x
16	wiyun.com	14	x			
17	mopub.com	10	x			x
18	mobfox.com	10	x			x
19-45	Under 10 applications modules	76	14	9	1	9
Total	45	1875	20	14	2	17

an ad module's network traffic, specifically the receipt of ad images by an ad module bundled in an application. The unique pattern of ad module HTTP sessions is focused, which start with an HTML or JavaScript request, then it downloads ad images, HTML or JavaScript files from a server, often after one or more redirects. Some HTTP sessions share same cookie is identified.

Figure 4.3 shows proposed approach, which consists of two phases. First, the network traffic is separated into two groups, and apply proposed graph transformation algorithm to them in order to generate graph groups representing 1) known ad modules and 2) graphs to be tested for ad content (candidate ad graphs). In the second phase, detection whether or not a network traffic graph was generated by an ad module. Specifically, the candidate ad graphs are compared with the known ad graphs using graph distance, and if the result of graph distance calculation is under a certain threshold, classified network traffic as ad modules generated.

4.4.1 Graph Definition

In this paper, graph has labeled vertices and labeled edges. Let L_V and L_E be the finite sets of possible vertex and edge labels. A graph is a 4-tuple $G = (V, E, \mu, \nu)$, where V is a set of finite vertices, $E \subseteq V \times V$ is the set of finite edges, $\mu : V \rightarrow L_V$ is a function assigning labels to the vertices, and $\nu : E \rightarrow L_E$ is a function assigning labels to the edges. If v_i is vertex in V, then $e_{ij} = (v_i, v_j)$ is edge between vertices v_i and v_j in E, and $E_{i*} \subseteq E$ is the set of finite edges of vertex v_i. If $V = \emptyset$ then G is an empty graph.

4.4.2 Graph of HTTP Sessions

The graph of HTTP sessions is constructed using an HTTP request and response pair derived from HTTP headers. Given an HTTP request and response, definition of an HTTP session graph vertex as the HTTP request and response pair, and the label of that vertex as the HTTP request line and content type from its HTTP header; that is, the vertex label of HTTP session v is $L_V(v) = (rline, ctype)$.

HTTP is a stateless protocol, which consists mainly of simple exchanges: an HTTP request from the client and an HTTP response from the server. Even though HTTP sessions do not usually relate to other HTTP sessions, content data between HTTP sessions can have direct relationships. Specifically, the ad module traffic discussed in section 4.3.2 involves multiple HTTP sessions following a standard set of patterns:

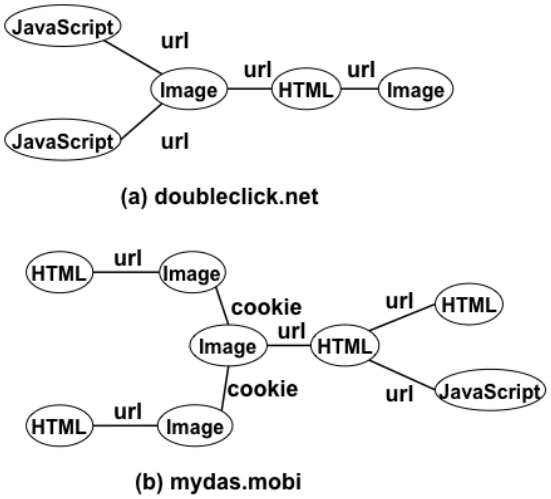

Figure 4.4 Ad module HTTP session graphs. (a) Graph of doubleclick, consisting of 5 vertices and 4 edges. One image vertex connects to both JavaScript and HTML vertices, while the other image vertex connects only to the HTML vertex. (b) Graph of mydas, consisting of 8 vertices and 7 edges. All 3 image vertices share the same cookie, but connect to different HTML vertices. One HTML vertex connects to another HTML vertex, as well as a JavaScript vertex.

- URL in HTTP Message-Body : Based on HTML or JavaScript, the original response includes URL in HTML or JavaScript, which causes new HTTP requests to start another HTTP sessions to access to included URL.
- HTTP Cookie : A relationship between multiple HTTP sessions, where the HTTP header field includes session information for the client and server.
- HTTP Persistent Connections : Multiple HTTP sessions related by their use of one TCP session.
- HTTP Redirect : HTTP sessions related by one URL forwarding to another URL.

These relationships to be assigned and labeled the edges. Given two HTTP sessions of vertices v_1 and v_2, definition of an edge as $e_{12} = (v_1, v_2)$ if a relation exists in between v_1 and v_2, and definition of the edge label $L_E(e_{12}) = (url, cookie, persistent, redirect)$ as the type of relation the edge represents. i.e., v_1 and v_2 have only same url, edge label shows $L_E(e_{12}) = (url)$ that has the type of relations in two vertices.

Figure 4.4 shows two sample graphs extracted from dataset. Both graphs represent relationships between ad modules' HTTP sessions, showing how many images and HTML or JavaScript files are downloaded and connected to each other.

4.4.3 Graph Distance

The similarity of two graph is computed by several graph matching algorithms in [140]. Proposed approach uses graph distance in [141], which computes the most common subgraph and uses that to determine the similarity between the graphs. Given two non-empty graphs G_1 and G_2, definition of the graph distance $d(G_1, G_2)$ as

$$d(G_1, G_2) = \frac{|mcs(G_1, G_2)|}{max(|G_1|, |G_2|)}$$

where $mcs(G_1, G_2)$ is a function which returns the most common subgraph of G_1 and G_2; $|G|$ is the number of vertices in graph G; and max is a function which returns the greater of its two input values.

4.4.4 HTTP Session Distance

Computing the distance between HTTP sessions requires two steps. First, graphs are generated from the network traffic using the graph translation algorithm described in section 4.4.2. Second, the similarity between the graphs is calculated using the graph distance algorithm from section 4.4.3. The graph distance requires the largest common subgraph between any two graphs, which is computed using Ullman's backtrack method [142]. The consistency of HTTP sessions is determined by matching vertices and edges to find the maximum common subgraph.

Algorithm 1 takes two HTTP session graphs, G_1 and G_2, as input, and outputs the permutation matrix P that represents a graph isomorphism from a subgraph of G_2 to a subgraph of G_1. Let M_1 and M_2 denote the adjacency matrices of G_1 and G_2 respectively; n and m, the number of vertices of G_1 and G_2 ; $P = (p_{i,j})$, the $n \times n$ permutation matrix whose initial state is $p_{ij} = 0$; $i, j = 1, \ldots, n$; and k, the number of matching vertices between G_1 and G_2, which is incremented before calling Backtrack.

The Backtrack function (pseudocode included in Algorithm 1) is designed to find a permutation matrix that represents a graph isomorphism from G_2 to G_1. First, Backtrack checks k, and if k

is bigger than m, Backtrack returns the permutation matrix P. Backtrack then checks whether the vertices $v_{1,i}$ and $v_{2,k}$ match, and then whether the edges $E_{1,i*}$ and $E_{2,k*}$ match (lines 13-14). If vertices and edges both match, Backtrack sets $p_{ki} \in P$ to 1 and $p_{kj} \in P$ to 0, where $j = 0 \ldots n, i \neq j$. Finally, Backtrack determines whether P is a permutation matrix of a subgraph isomorphism from G_2 to G_1 by comparing $S_{k,k}(M_2)$ to $S_{k,n}(P)M_1(S_{k,n}(P))^T$. $S_{k,n}(P)$ denotes the $i \times n$ submatrix of P, and P is valid subgraph isomorphism when $S_{k,k}(M_2) = S_{k,n}(P)M_1(S_{k,n}(P))^T$. $S_{k,n}(P)$ then represents the matching of k vertices between G_2 and G_1. If P is subgraph isomorphism, recursive on Backtrack after k is incremented.

4.4.5 Vertex and Edge Matching Rules

The vertex and edge matchings are computed using the graph labels. Given two graphs of HTTP sessions $G_1 = (V_1, E_1, \mu_1, \nu_1)$ and $G_2 = (V_2, E_2, \mu_2, \nu_2)$, definition of matchings as follows:

- Vertex matching : Two vertices $v_i \in V_1$ and $v_j \in V_2$ is considered to be matching if their all vertex labels $L_{V_1}(v_i) = (rline_i, ctype_i)$ and $L_{V_2}(v_j) = (rline_j, ctype_j)$ are the same, i.e. $rline_i = rline_j$ and $ctype_i = ctype_j$.

- Edge matching : Two edge groups $E_{i*} \subseteq E_1$ and $E_{j*} \subseteq E_2$ is considered to be matching if any of their contained edge pairs, $e_{ia} \in E_{i*}$ and $e_{jb} \in E_{j*}$, have matching labels, where if edge labels $L_{E_1}(e_{ia})$ and $L_{E_2}(e_{jb})$ are equal, then the two edges are matching.

4.5 Evaluation

4.5.1 Experimental Setup

Evaluation dataset contains 1,188 free applications from Google Play Japan's Top 100 and Recently Uploaded lists from January to April 2012, and collected the network traffic generated by running each application manually for 5 to 15 minutes on Android device.

Collected network traffic dataset contained 6,093,682 packets, and the total data size was 5.6 GBytes. There were 5,565,529 HTTP packets, and HTTP GET / POST requests made up 107,859 packets. Graphs are generated from this dataset, resulting in 4,696 generated ad graphs. Table 4.4 shows the number of graphs of each ad module. There were 16,205 other graphs which were not

generated by known ad modules. However, it is reasonable to assume that these graphs contain other ad module traffic and standard application traffic.

2,000 of these graphs (candidate ad graphs) are manually classified and 2,000 graphs from standard traffic (standard graphs), in order to evaluate the validity of proposed approach. All ad modules' name and traffic is not be able to collect, because ad modules are developed by a lot of company in worldwide. Assumption that is proposed approach detects general cases of ad modules' traffic with only major ad modules' graphs. Therefore, candidate ad graphs is classified and standard graphs to evaluate see if proposed approach has availability of detection or not. Evaluation process determined which graphs were likely to be ad graphs by carefully investigating each graph's topology. If graph contained an image label vertex that, when viewed was an ad image, it is defined as a candidate ad graph. This step does not depend on 'Host' field in the HTTP header. Assumption that accuracy ad modules' name are extracted from static analysis of application. Since ad graphs is only identified by their images, then may be text-based ad modules that were not considered. Table 4.5 shows the number of candidate ad graphs are able to associate with an ad module. Encrypted packets were not considered in this experiment.

The entirely of proposed approach is tested as follow. First, an ad module and N ad graphs at random is selected from each ad module's known graphs (see Table 4.4) except for one vertex graphs, where N was raised from 100 to 1,000 in intervals of 100. The N known ad graphs is used to test how accurately proposed approach could identify other ad graphs, and to see what could be detected about the 2,000 candidate ad graphs. Additionally, evaluation of false positive rate that shows how many graphs in 2,000 standard graphs could be detected by N known ad graphs. In detection rate, investigation shows that one vertex graphs affect to detection rate. the comparison of detection rate in between training set includes one vertex graphs and other set does not include it.

4.5.2 Experimental Results

Figure 4.5 shows the number of detected known ad graphs per graph distance, and Figure 4.6 shows the number of detected candidate ad graphs per graph distance. In both cases, the Y axis shows the number of detected ad graphs, and the X axis, the graph distance between graphs. these results is used to determine the threshold detecting ad graphs. If the distance between a candidate and a known ad graph is 1, the graphs are identical, so the candidate must be an ad graph. A

Table 4.4 The number of graphs associated with each known ad module, converted from the dataset traffic. Ad modules' traffic was selected using destination domain matching.

No.	Ad Module	# Graphs
1	google-analytics.com	1385
2	doubleclick.net	843
3	googlesyndication.com	408
4	admob.com	393
5	medibaad.com	363
6	mediba.jp	188
7	flurry.com	187
8	madserving.cn	131
9	adlantis.jp	114
10	mobclix.com	114
11	mydas.mobi	101
12	adwhirl.com	95
13	mopub.com	54
14	airpush.com	41
15	yicha.jp	35
16	inmobicdn.net	33
17	inmobi.com	31
18	mobfox.com	21
19	jumptap.com	15
20	adfonic.net	14
21	wiyun.com	14
22	mdotm.com	13
23	admarvel.com	12
24	domob.cn	12
25	inmobi-jp.com	10
26-45	others (under 10 graphs)	69
Total	45	4696

Table 4.5 The number of candidate ad graphs, associated with various ad modules. The traffic to convert to graphs was selected using domain matching in the HTTP headers and patterns of ad image downloading.

No.	Candidate Ad Module	# Graphs
1	ad-stir.com	229
2	ad-maker.info	140
3	nend.net	82
4	i-mobile.co.jp	63
5	microad.jp	58
6	amoad.com	51
7	adsmogo.com	49
8	droidhen.com	37
9	wallsmobile.com	33
10	menue.fm	32
11	pianoman-am.com	29
12	adimg.net	28
13	adlayout.net	28
14	shufoo.net	28
15	naver.jp	26
16	adresult-sp.jp	23
17	edgesuite.net	23
18	durasite.net	23
19	goo-net.com	20
20	carsensor.net	19
21	strikead.com	18
22	adsta.jp	16
23	umeng.com	16
24	madvertise.de	15
25-254	others (under 15 graphs)	914
Total	254	2000

threshold of 0.5 is selected, because in that case at least half of the graph is an isomorphism to an ad graph.

Figure 4.7 shows the detection rate of known ad graphs for varying values of N, and Figure 4.8 shows the detection rate of candidate ad graphs for varying values of N, and Figure 4.9 show the false positive rate of standard graphs for varying values of N. The Y axis shows the detection rate proposed approach achieves according to the values of N (shown on the X axes).

4.5.3 Training Set Screening

While investigating detection case in the experiments, if a small graph (only one vertex) was included in the training set, it matched many other small graphs, generating excessive false positives. Therefore, small graphs is avoided when determining the final training set.

In detail of this decision effect, comparison of detection rate of training set includes one vertex graphs whether or not. Figure 4.10 shows the improved detection rate of known ad graphs and candidate ad graphs for varying values of N except for one vertex graphs. used threshold is 0.5 in both cases. The detection rate are improved at 6.7% by N = 100 to 1.4% by N = 1,000, ad graphs. In candidate ad graphs, detection rate are decreased at 3.5% by N = 100 to 0.7% by N =1,000.

In proposed method, it derives to a distance is 1 that the case of one vertex graph is compared with one vertex graph. Then a distance is decreased from 0.5 in two vertices graph. If training set includes one vertex graphs, almost other one vertex graphs and two vertices graphs are detected as ad modules' traffic by proposed method. When training set has small number of graphs, this case directory affects to detection rate. Therefore, conclusion is one vertex graphs do not model correct ad modules' network traffic pattern for training set.

4.6 Discussion

4.6.1 Detection Rate Consideration

The experimental results show that this approach can detect ad modules' network traffic using given ad graphs. The accurate detection of ad modules and false positive of standard traffic rely on the value of the graph distance threshold.

When detecting other known ad graphs, proposed approach had a 50% detection rate at $N = 1,000$ with a graph distance threshold 1, but at only $N = 200$, the detection rate approached 70%

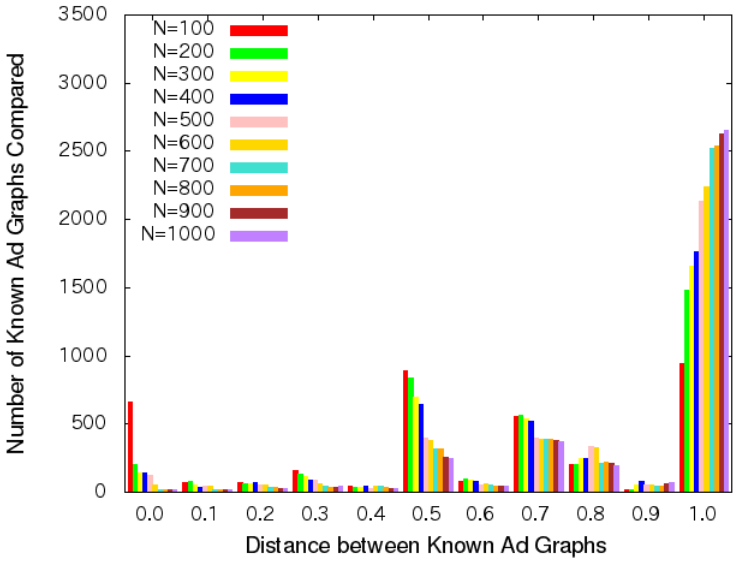

Figure 4.5 Graph distance statistics using N known ad graphs to classify other known ad graphs.

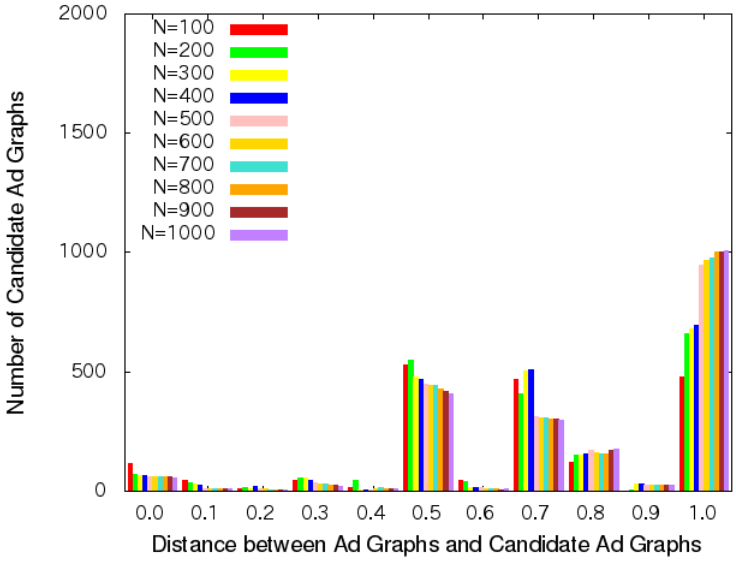

Figure 4.6 Graphs distance statistics for candidate ad graphs using N known ad graphs.

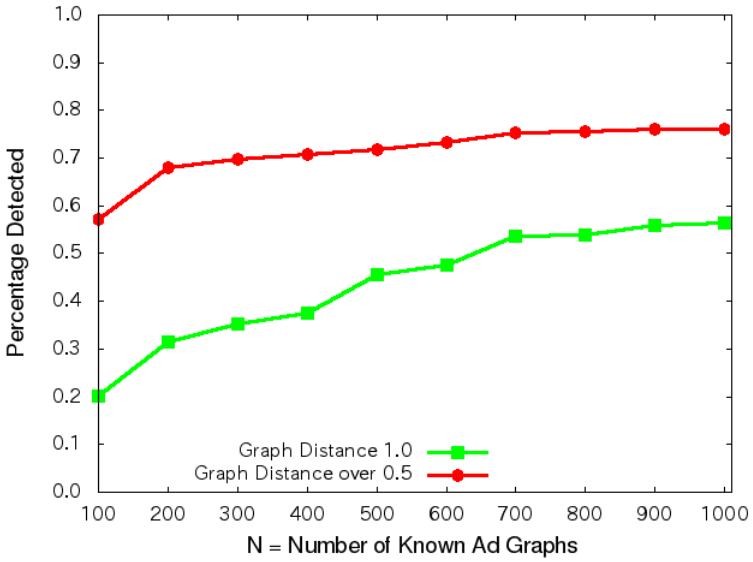

Figure 4.7 Detection rate of other known ad graphs using N known ad graphs.

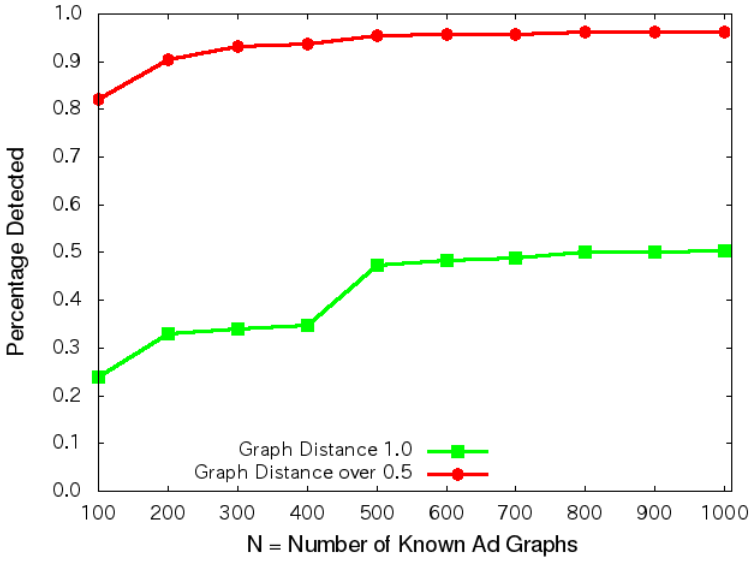

Figure 4.8 Detection rate of candidate ad graphs using N known ad graphs.

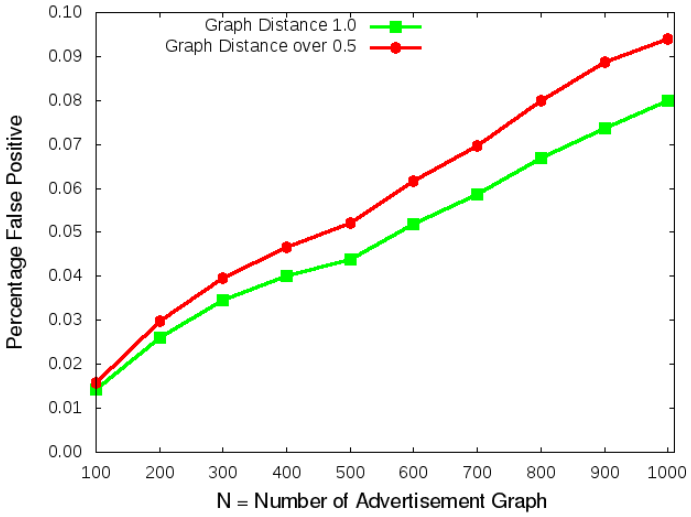

Figure 4.9 False positive rate of standard graphs using N known ad graphs.

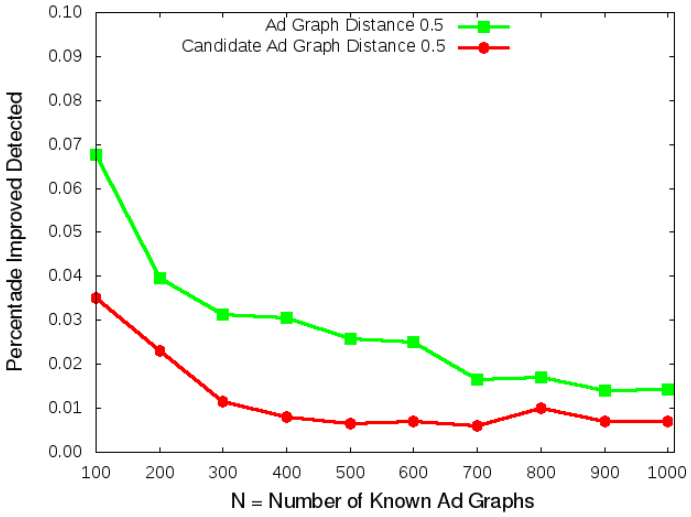

Figure 4.10 Improved Detection rate of know ad graphs and other known ad graphs using N known ad graphs except for only one vertex graphs.

at a graph distance threshold of 0.5. All known ad modules' groups (regardless of the size of N) included at least 1 graph from each known ad module. Therefore, the conclusion is a small number of N known ad graphs can be used to detect almost all ad modules' network traffic. Different ad modules produce similar network traffic, which shows that extracting a characteristic graph of an ad modules network traffic is a good indicator of other ad modules' network traffic.

When detecting candidate ad graphs, proposed approach only had a 50% rate of detection for all $N > 500$ with a graph distance of 1. However, using a graph distance threshold of 0.5 increased the detection rate to 90% by $N = 200$, indicating that the number of perfectly isomorphic graphs does not increase noticeably for $N \geq 500$, but that almost all candidate ad graphs are identifiable at only $N = 200$ with in a distance of 0.5. This result shows that a small number of known ad graphs can cover almost all candidate ad module graphs. the conclusion is candidate ad modules and known ad modules have similar network behavior.

4.6.2 False Detection Rate Consideration

When false detected standard graphs, proposed approach had 1.5% by $N = 100$ in a graph distance of 0.5 and 1.0. This rate is increased to under 10% by $N = 1,000$. This result shows a distance of 0.5 has more false positive than a distance of 1.0. It covers variety topology graphs and not strict matching. Therefore, these reason leads to false positive case at distance of 0.5. The conclusion is false positive rate is small effected from the number of know ad graphs. Proposed approach could smoothly distinguish between ad graphs and standard graphs.

4.6.3 Ad Module Network Traffic Consideration

Proposed approach shows only a 70% detection rate for other known ad graphs, but a 90% detection rate for candidate ad graphs. Ad module's graphs can exhibit different topologies, representing different network traffic patterns. Known ad graphs are randomly selected for the training set, so represents only a part of the other known ad graphs, leading to the relatively low detection rate. However, if the known ad graphs used for detection are carefully chosen, they can support a variety of network behavior, resulting a high detection rate. In the experiments, the candidate ad modules' behavior (both in the original traffic and in the converted graphs) is similar to that of known ad modules in the training set, particularly with regard to downloading images. It is this similarity that leads to proposed approach's high detection rate for candidate ad graphs.

The experimental results show that the number of detectable graphs remain fairly consistent for graph distance values over 0.5 and under 1. Ad modules' destinations change frequently because ad modules use cloud services or CDNs for load balancing. Additionally, an ad module update might influence the module's network behavior. However, proposed approach considers not only destination IP addresses but also network traffic patterns including the URLs in HTTP contents, the HTTP cookies, the HTTP persistent connections, and the HTTP redirects, so proposed approach can still identify ad modules network traffic with a high level of accuracy.

4.6.4 Complexity Analysis

If the graph matching input is taken to be two graphs of HTTP sessions G_1 and G_2, the time complexity is $O(|G_1||G_2|^{|G_1|})$, as there are $|G_1|$ steps and at each step, a permutation matrix of at worst size $|G_2|^{|G_1|}$ be must be computed. The space complexity is $O(|G_1|^2|G_2|)$, because the adjacency matrix of G_1 and G_2 and the permutation matrix of G_1 must be stored.

4.7 Conclusion

Ad services are supplied by ad modules, which can have access to personal information. It is pointed out that advertisement modules possibly which can violate the user's privacy. Ad module network traffic which are constructed by receiving ad contents over the network is characteristic in applications. Previous works analyzing ad modules aim to identify ad modules with in applications. These methods disassemble the application, or track information flow on the device. Proposed approach uses only the network traffic relationships to detect ad modules' activity and is thus applicable even if static analysis is can not be used (e.g., large quantities of data in corporate or WiFi networks). Additionally, while there are a small number of major ad service providers, many smaller providers are unlikely to be detected by static analysis. However, the experiments indicate that manually identified ad graphs are similar to known ad graphs, and thus previously unknown modules could be detected by proposed approach using only a few known ad graphs. The proposed advertisement traffic detection method is contributed as follows.

- In order to identify ad modules embedded in applications, new method is proposed for ad module traffic identification. Proposed method generates a graph of ad module network behavior focusing on the relationships between HTTP sessions, and computes the distance

between these graphs. In the evaluation, 1,188 Android applications' network behavior analyzed, which produced a dataset of 5,565,529 packets. 4,696 graphs of known ad modules is then generated and 16,205 graphs of other traffic from dataset.

- Proposed approach showed a 76% detection rate when using 1,000 known ad graphs to identify other known ad graphs, a detection rate of 96% for 2,000 manually selected candidate ad graphs and under 10% false positive rate for 2,000 manually selected standard graphs from the remaining 16,205 graphs. To further evaluation, a detection rate difference is measured when training set includes one vertex graphs. In the result, training set without one vertex graphs improved detection rates are 6.7% for known ad graphs and 3.5% for candidate ad graphs.

Algorithm 1 : Subgraph isomorphism using Ullman's method

Input: $G_1 = (V_1, E_1, \mu_1, \nu_1), G_2 = (V_2, E_2, \mu_2, \nu_2)$

Output: Matrix P, a subgraph isomorphism from G_2 to G_1

$n \Leftarrow |V_1|$
$m \Leftarrow |V_2|$
M_1, M_2 are adjacency matrices of G_1, G_2
$P = (p_{ij})$ is $n \times n$ initial permutation matrix
$k \Leftarrow 1$
Backtrack(M_1, M_2, P, k)

function Backtrack (M_1, M_2, P, k)
if $k > m$ **then**
    return P
end if
for all $i \Leftarrow 1$ to n **do**
    if vertex label matches $(v_{1,i}, v_{2,k})$ **then**
        if edge label matches $(E_{1,i*}, E_{2,k*})$ **then**
            $p_{ki} \Leftarrow 1$
            for all $j \Leftarrow 1$ to n **do**
                if $j \neq i$ **then**
                    $p_{kj} \Leftarrow 0$
                end if
            end for
            if $S_{k,k}(M_2) = S_{k,n}(P)M_1(S_{k,n}(P))^T$ **then**
                Backtrack($M_1, M_2, P, k + 1$)
            end if
        end if
    end if
end for
end function

Chapter 5

Conclusions

5.1 Concluding remarks

In this dissertation, several novel techniques that may be useful in a multi-layered defense against information leakage were introduced and evaluated.

In Chapter 1, it was argued that multiple layers of defense are an effective countermeasure for information leakage. The importance of the OS and application layers was stressed. Kernel vulnerability attacks on the OS being a significant security risk, novel countermeasures are needed to mitigate this threat. Given the popularity of devices with applications that store user information, the OS should prevent personal information leakage by novel network detection methods based on traffic modeling. Three research problems were presented. The first problem is that the adversary can defeat kernel countermeasures by executing an exploit code that leads to memory corruption in the kernel mode. Although existing protective methods cannot solve this problem, the proposed KMO does so by monitoring kernel virtual memory to identify illegal memory overwrites in the kernel mode. The second and third problems involve applications that leak personal information for advertising and tracking purposes. Previous security approaches are not able to deal with this satisfactorily. From a network security perspective, automatic signature generation and inspection of traffic at the network layer are effective at detecting personal information leaks without requiring modification of the Android framework. The problem of classification and identification of advertisement modules was introduced. The proposed network traffic modeling algorithm correctly converts advertisement module network behavior into a graph, to identify applications contains what kind of advertisement module is contained in the application.

In Chapter 2, it was noted that the OS must mitigate various attacks and reduce the attack surface to prevent hostile exploitation of kernel vulnerabilities. Although KASLR, CFI, KPTI, and SMAP / SMEP mitigate kernel vulnerability to memory corruption and thus to privilege escalation or the avoidance of security features, kernel layer attacks still have the potential to succeed. KMO, the proposed novel security mechanism, provides a secret observer to monitor the original kernel virtual memory. KMO has multiple inspection points to determine invalid kernel virtual memory overwriting, identify malicious system call arguments, and prevent attacks through the direct mapping region. In an evaluation of Linux, KMO was able to inspect system call arguments and identify the memory corruption of security features. The performance overhead achieved mitigation of this in terms of each system call invocation on the kernel. The web application overhead for KMO monitoring has a small cost at the running process. KMO supports PCID on TLB, which reduces the overhead penalty of KMO, to decrease the actual cost for monitoring and virtual memory switching.

In Chapter 3, advertisement services, which are widely accepted among application developers but pose security and privacy challenges, were considered. Many Android applications require permissions for sensitive information access and network features; among them are applications that connect to outside servers without the user's knowledge. Furthermore, observations of applications' network behavior have shown that a large amount of sensitive information, particularly immutable identifiers such as UDIDs, is routinely transmitted. A novel clustering method is proposed using HTTP packet distances, including both the distance between HTTP packet destinations and the distance between HTTP packet contents. This clustering method, in combination with signature generation and screening, was used on a dataset of actual Android applications and network packets, including sensitive information packets. The proposed method was able to achieve highly accurate detection of packets containing sensitive data, at the cost of relatively small false positives.

In Chapter 4, it was remarked that many "free" applications include ad services in order to obtain developer revenue. Ad services are supplied by ad modules, which can violate the user's privacy. Ad module network traffic is identifiable by its use of HTTP redirection and use of HTML or JavaScript requests to download ad images. A new method was proposed to identify ad modules embedded in applications. The proposed method generates a graph of ad module network behavior focusing on the relationships between HTTP sessions and computes the distance between these graphs. In the evaluation, the network behavior of actual Android applications was analyzed,

producing a dataset of network packets. From this dataset, known ad modules graphs and other traffic graphs were generated. The proposed approach showed a high detection rate when using the parts of known ad graphs to identify other known ad graphs, a high precision detection rate for manually selected candidate ad graphs, and a low false positive rate for manually selected standard graphs from the remaining graphs.

5.2 Future directions

This study has a number of limitations that need to be overcome in the future. In addition, it has implications for the direction of future research in this field.

(1) OS Monitoring Enhancement

KMO keeps the virtual memory switching functions in the kernel virtual memory space and then invokes them from the original kernel code. Although the adversary can potentially target the KMO's function, KASLR randomizes the virtual memory space, thus obscuring the KMO function's virtual addresses. The adversary identifies the valid monitoring data's virtual address of direct mapping to calculate manually the position from the physical page's virtual address. In response, KMO unmaps the secret pages of the direct mapping space in the kernel virtual memory to reduce the attack surface. Future research should support other security features that can run in this secret virtual memory space. This method could prevent kernel vulnerability attacks that evade the monitoring mechanism of the kernel security feature. The application overhead should be measured with other benchmark software, and the cost of increasing the monitoring data should be investigated. Additionally, the relationship between both monitoring timing and kernel vulnerability effects should be examined for attack-detection capability.

(2) Application Traffic Monitoring Capability

The using of HTTP packet destination distance allows the proposed system to generate useful signatures. A concern regarding the use of the destination distance is that two HTTP packets may have close IP addresses, but be owned by different organizations, thus generating an erroneously small distance. In practice, this situation is rare, so current implementation

does not specifically handle it. However, using a registration information process such as WHOIS could be helpful for the verification of IP addresses and domain names, which could be used to confirm the distances. The proposed approach also does not focus on encrypted or obfuscated traffic. It can be difficult to detect sensitive information in SSL traffic, but if an ad module uses one encryption key for all applications, or applies a deterministic hash function to sensitive information, the proposed approach can detect it. Some ad modules collect information supplied by the developers. If this traffic is small, the proposed approach may not cluster and thus not detect it. Detection would require deeply investigating small network traffic to determine the causes of sensitive information leakage from characteristic network behavior. This is a feature to add in the future. Also, the applications investigated in this study did not send email addresses, address lists, or location information; in future work, the number and type of applications evaluated should be increased.

(3) Deep Analysis of Suspicious Applications Libraries

The proposed graph modeling process focuses on the sequence of HTTP sessions. However, ad modules have other characteristic behaviors, such as refreshing images at predictable intervals and responding to user actions [143]. Automatically modeling and extracting protocol specifications from network traffic would improve the correctness of the ad module behavior models; this will be attempted in future work. Other goals include improving the exactness of the graphs and determining whether a combination of the approaches. Actually, a detection rate difference was measured when the training set included one vertex graph: it was found that the training set without the one vertex graph improved detection rates for known ad graphs and candidate ad graphs. Additionally, consideration should be given to the possibility of generating more general graphs from multiple known graphs, which would (ideally) widely cover a variety of ad module network behavior. If this is possible, it would decrease the total number of graphs without compromising the high detection rate. However, if such a graph were large, it could increase the time required for graph comparisons. The graph matching algorithm proposed has improved the time complexity of Ullman's algorithm [144], and so should be applied to the proposed approach in the future.

Acknowledgements

I am cordially grateful to my supervisor Associate Professor Toshihiro Yamauchi of the Graduate School of Natural Science and Technology at Okayama University for all the coordination and encouragement, and for his invaluable directions and constructive suggestions on this research activity and on writing this dissertation.

Many thanks also to Professors Akito Monden and Hideo Taniguchi of the Graduate School of Natural Science and Technology at Okayama University for their brilliant comments and numerous suggestions for revising this dissertation.

I greatly appreciate Mr. Satoshi Tonami, Mr. Kenichi Magata, and Mr. Yasushi Matsumoto of Intelligent Systems Laboratory, SECOM Co., Ltd. for their support and useful advice ever since I joined for SECOM Co., Ltd.

My sincere thanks also go to Professor Toyoo Takata of Iwate Prefectural University and Professor Hiroyuki Seki of Nagoya University, who support me with an opportunity to conduct this research.

Finally, I would also like to say a heartfelt thank you to my wife, sons, brother, and parents for helping me during this challenging research period.

References

[1] Japan Network Security Association (JNSA), 2018 Information security incident investigation report (Japanese), available from https://www.jnsa.org/result/incident/2018.html, (accessed 2019-10-09).

[2] National Center of Incident readiness and Strategy for Cybersecurity (NISC), Annual report of Cyber security 2019 (Japanese), available from https://www.nisc.go.jp/active/kihon/pdf/cs2019.pdf, (accessed 2019-10-09).

[3] Information-technology Promotion Agency, Japan (IPA), Information Security White Paper 2019 (Japanese), available from https://www.ipa.go.jp/security/publications/hakusyo/2019.html, (accessed 2019-10-09).

[4] The New York Times, Ecuador Investigates Data Breach of Up to 20 Million People, available from https://www.nytimes.com/2019/09/17/world/americas/ecuador-data-leak.html, (accessed 2019-11-01).

[5] The Japan Times, 1.25 million affected by Japan Pension Service hack, available from https://www.japantimes.co.jp/news/2015/06/01/national/crime-legal/japan-pension-system-hacked-1-25-million-cases-personal-data-leaked/, (accessed 2019-11-01).

[6] Department of Justice, U.S. Attorney's Office, Northern District of California, Former Yahoo Software Engineer Pleads Guilty To Using Work Access To Hack Into Yahoo Users' Personal Accounts, https://www.justice.gov/usao-ndca/pr/former-yahoo-software-engineer-pleads-guilty-using-work-access-hack-yahoo-users, (accessed 2019-11-01).

[7] ZDNet, Adobe left 7.5 million Creative Cloud user records exposed online, available from https://www.zdnet.com/article/adobe-left-7-5-million-creative-cloud-user-records-exposed-online/, (accessed 2019-11-01).

[8] Nikkei Asian Review, Customer data leak deals blow to Benesse, available from https://asia.nikkei.com/Business/Customer-data-leak-deals-blow-to-Benesse, (accessed 2019-11-01).

[9] The Shadow Brokers, Lost in Translation1, available from https://steemit.com/shadowbrokers/@theshadowbrokers/lost-in-translation, (accessed 2019-11-01).

[10] Reuters, Bangladesh Bank official's computer was hacked to carry out $81 million heist, available from https://www.reuters.com/article/us-cyber-heist-philippines/bangladesh-bank-officials-computer-was-hacked-to-carry-out-81-million-heist-diplomat-idUSKCN0YA0CH, (accessed 2019-11-01).

[11] The Japan Times, Cryptocurrency exchange Coincheck loses JPY 58 billion in hacking attack, available from https://www.japantimes.co.jp/news/2018/01/27/national/cryptocurrency-exchange-coincheck-loses-58-billion-hacking-attack/, (accessed 2019-11-01).

[12] Information Security Management System (ISMS), ISO 27001:2013, available from https://www.isms.online/iso-27001/, (accessed 2019-10-09).

[13] National Institute of Standards and Technology (NIST), SP 800 series, available from https://csrc.nist.gov/publications/sp800, (accessed 2019-10-09).

[14] The Center for Financial Industry Information Systems (FISC), Security Guidelines on Computer Systems for Financial Institutions (Ninth Edition), available from https://www.fisc.or.jp/publication/book/000020.php, (accessed 2019-10-09).

[15] The PCI Security Standards Council, PCI DSS, available from https://www.pcisecuritystandards.org/ (accessed 2019-10-09).

[16] U.S. Department of Health & Human Services (HHS), Health Insurance Portability and Accountability Act (HIPAA) and the Patient Protection and Affordable Care Act (ACA), available from https://www.cms.gov/Regulations-and-Guidance/Administrative-Simplification/HIPAA-ACA/index.html, (accessed 2019-10-09).

[17] Japan Institute for Promotion of Digital Economy and Community, The PrivacyMark system, available from https://privacymark.org/, (accessed 2019-10-09).

[18] The Cybersecurity and Infrastructure Security Agency (CISA), U.S. Computer Emergency Readiness Team (US-CERT), available from https://www.us-cert.gov/, (accessed 2019-10-09).

[19] Japan Computer Emergency Response Team Coordination Center (JPCERT/CC), available from https://www.jpcert.or.jp/, (accessed 2019-10-09).

[20] Lockeed Martin, The Cyber Kill Chain, available from https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html, (accessed 2019-10-09).

[21] Information-technology Promotion Agency, Japan (IPA), System Design Guide for Thwarting Targeted Email Attacks, available from https://www.ipa.go.jp/security/vuln/newattack.html, (accessed 2019-10-09).

[22] The MITRE Corporation, ATT&CK, available from https://attack.mitre.org/, (accessed 2019-10-09).

[23] National Center of Incident readiness and Strategy for Cybersecurity (NISC), Common Standards for Information SecurityMeasures for Government Agencies and Related Agencies (FY2018), available from https://www.nisc.go.jp/eng/pdf/kijyun30-en.pdf, (accessed 2019-10-09).

[24] Cloud Security Alliance (CSA), CSA Security Trust Assurance and Risk (STAR), available from https://cloudsecurityalliance.org/star/, (accessed 2019-10-09).

[25] The GSMA Foundation, IoT Security Guidelines and IoT Security Assessment, available from https://www.gsma.com/iot/iot-security/iot-security-guidelines/, (accessed 2019-10-09).

[26] International Organization for Standardization (ISO), ISO 28000:2007 (Specification for security management systems for the supply chain), available from https://www.iso.org/standard/44641.html, (accessed 2019-10-09).

[27] The Charter of Trust, available from https://www.charter-of-trust.com, (accessed 2019-10-09)

[28] Linux Vulnerability Statistics, available from https://www.cvedetails.com/vendor/33/Linux.html. (accessed 2019-07-05).

[29] Chen, H., Mao, Y., Wang, X., Zhow, D., Zeldovich, N. and Kaashoek, F, M.: Linux kernel vulnerabilities - state-of-the-art defenses and open problems, the 2nd Asia-Pacific Workshop on Systems (APSys), (2011).

[30] Linux Kernel Defence Map, available from https://github.com/a13xp0p0v/linux-kernel-defence-map. (accessed 2019-06-05).

[31] Enck, E., Octeau, D., McDaniel, P. and Chaudhuri, S.: A Study of Android Application Security, the 20th USENIX Conference on Security Symposium, (2011).

[32] Kemerlis, P, V., Polychronakis, M. and Keromytis, D, A.: ret2dir - Rethinking Kernel Isolation, the 23rd USENIX Conference on Security Symposium, pp. 957 - 972, (2014).

[33] Linden, A. T.: Operating System Structures to Support Security and Reliable Software, ACM Computing Surveys (CSUR), Vol. 8, No. 4, pp. 409 – 445, (1976).

[34] Security-enhanced Linux, available from http://www.nsa.gov/research/selinux/, (accessed 2018-08-10).

[35] Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N. and Boneh, D.: On the effectiveness of address-space randomization. the 11th ACM Conference on Computer and Communications Security (CCS), pp. 298 - 307, (2004).

[36] Abadi, M., Budiu, Mihai., Erlingsson, U. and Ligatti, J.: Control-Flow Integrity Principles, Implementations, the 12th ACM Conference on Computer and Communications Security (CCS), pp. 340 - 353, (2005).

[37] Kemerlis, P, V., Portokalidis, G. and Keromytis, D, A.: kGuard - Lightweight Kernel Protection against Return-to-User Attacks, the 21st USENIX Conference on Security symposium, (2012).

[38] Ingo Molnar, [announce] [patch] NX (No eXecute) support for x86, 2.6.7-rc2-bk2, available from http://lkml.iu.edu/hypermail/linux/kernel/0406.0/0497.html, (2004). (accessed 2018-08-10).

[39] Mulnix D.: Intel® Xeon® Processor D Product Family Technical Overview, available from https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview, (2015), (accessed 2018-08-10).

[40] Lipp, M., Schwarz, M., Fellner, R., Maurice, C. and Mangard, S.: KASLR is Dead - Long Live KASLR, 2017 International Symposium on Engineering Secure Software and Systems (ESSoS), Vol. 10379, No. 3, pp. 161 - 176, (2017).

[41] CVE-2016-8655, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655. (accessed 2019-05-12).

[42] CVE-2017-6074, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6074. (accessed 2019-05-12).

[43] CVE-2017-7308, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7308. (accessed 2019-05-12).

[44] CVE-2017-16995, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16995. (accessed 2019-05-12).

[45] CVE-2017-1000112, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000112. (accessed 2019-05-12).

[46] CVE-2017-7533, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7533. (accessed 2019-05-12).

[47] CVE-2016-9793, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9793. (accessed 2019-05-12).

[48] CVE-2016-4997, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4997. (accessed 2019-05-12).

[49] Exploit Database, Nexus 5 Android 5.0 - Privilege Escalation, available from https://www.exploit-db.com/exploits/35711/. (accessed 2019-06-15).

[50] grsecurity: super fun 2.6.30+/RHEL5 2.6.18 local kernel exploit, available from https://grsecurity.net/~spender/exploits/exploit2.txt. (accessed 2019-06-15).

[51] AppBrain, Number of available Android applications, available from http://www.appbrain.com/stats/, (accessed 2014-02-06).

[52] Manuscript, inc.: Karelog, available from http://karelog.jp/, (accessed 2014-12-12).

[53] ASIAJIN, Android App Karelog Lets You Spy On Your Boyfriend Remotely, available from http://asiajin.com/blog/2011/08/31/android-app-karelog/ , (accessed 2014-11-12).

[54] Hornyack, P., Han, S., Jung, J., Schechter, S. and Wetherall, D.: These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications, 18th ACM Conference on Computer and Communications Security (CCS), (2011).

[55] Grace, M., Zhow, W., Jian, X. and Sadeghi, A.: Unsafe Exposure Analysis of mobile In-App Advertisements, 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec), (2012).

[56] Stevens, R., Gibler, C. and Crussell, J.: Investigating User Privacy in Android Ad Libraries, Mobile Security Technologies (MoST), (2012).

[57] Leontiadis, I., Efstratiou, C., Picone, M. and Mascolo, C.: Don't kill my ads! Balancing Privacy in an Ad-Supported Mobile Application Market, 13th ACM Sigmobile Workshop on Mobile Computing Systems and Applications (HotMobile), (2012).

[58] Pearce, P., Felt, P, A., Nunez, G. and Wagner, D.: AdDroid: Privilege Separation for Applications and Advertisers in Android, 7th ACM Symposium on Information, Computer and Communications Security (AsiaCCS), (2012).

[59] Shekhar, S., Dietz, M. and Wallach, S, D.: AdSplit: Separating smartphone advertising from applications, the 21st USENIX Conference on Security Symposium, (2012).

[60] Shu, R., Wang, P., Gorski III, A, S. andow, B., Nadkarni, A, Deshotels, L, Gionta, J, Enck, W. and Gu, X.: A Study of Security Isolation Techniques, ACM Computing Surveys (CSUR), Vol. 49, No. 3, pp. 1 – 37, (2016).

[61] Zhang, F. and Zhang, H.: SoK A Study of Using Hardware-assisted Isolated Execution Environments for Security, the Hardware and Architectural Support for Security and Privacy 2016, pp. 1 - 8, (2016).

[62] Spencer, R., Smalley S., Loscocco, P., Hibler, M. andersen, D. and Lepreau, J.: The Flask Security Architecture: System Support for Diverse Security Policies, the 8th USENIX Conference on Security Symposium, (1999).

[63] Koning, K., Chen, H, B., Giuffrida, C. and Athanasopoulos, E.: No Need to Hide: Protecting Safe Regions on Commodity Hardware, the Twelfth European System Conference (EuroSys), pp. 437 - 452, (2017)

[64] Vahldiek-Oberwagner, A., Elnikety, E., Garg, D. and Druschel, P.: ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys, CoRR abs/1801.06822, (2018).

[65] Mogosanu, L., Rane, A. and Dautenhahn, N.: MicroStache - A Lightweight Execution Context for In-Process Safe Region Isolation, the 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID), pp. 359 - 379, (2018).

[66] Frassetto, T., Jauernig, P., Liebchen, C. and Sadeghi A.-R.: IMIX - In-Process Memory Isolation EXtension, the 28th USENIX Conference on Security Symposium, (2018).

[67] Kim, H, C., Kim, T., Choi, H., Gu, Z., Lee, B., Zhang, X. and Xu, D.: Securing Real-Time Microcontroller Systems through Customized Memory View Switching, the 25th Network and Distributed System Security Symposium (NDSS), (2018).

[68] K. Volodymyr., L. Szekeres., M. Payer., G. Candea., R. Sekar. and D. Song.: Code-Pointer Integrity, 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), (2014).

[69] Song, C., Lee, B., Lu, K., Harris, W., Kim, T. and Lee, W.: Enforcing Kernel Security Invariants with Data Flow Integrity, the 2016 Annual Network and Distributed System Security Symposium (NDSS), (2016).

[70] Ge, X., Cui, W. and Jaeger, T.: GRIFFIN: Guarding Control Flows Using Intel Processor Trace, the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (APLOS), pp. 585 - 598, (2017).

[71] Huang, W., Huang, Z., Miyani, D. and Lie, D.: LMP: Light-Weighted Memory Protection with Hardware Assistance, the 32nd Annual Conference on Computer Security Applications (ACSAC), pp. 460 - 470, (2016).

[72] Yamauchi, T., Akao, Y., Nakamura, Y., Hashimoto, M.: Additional Kernel Observer to Prevent Privilege Escalation Attacks by Focusing on System Call Privilege Changes, the 2018 IEEE Conference on Dependable and Secure Computing (DSC), (2018).

[73] Davi, L., D. Gens, C. Liebchen, and A.-R. Sadeghi.: PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables, the 23th Network and Distributed System Security Symposium (NDSS), (2016).

[74] Pomonis, M., and Petsios, T.: “kRˆX: Comprehensive Kernel Protection against Just-In-Time Code Reuse, the Twelfth European Conference on Computer Systems (EuroSys), pp. 420 - 436, (2017).

[75] Hua, Z., Du, D., Xia, Y., Chen, H. and Zang, B.: EPTI - Efficient Defence against Meltdown Attack for Unpatched VMs, 2018 USENIX Annual Technical Conference (ATC), 2018.

[76] Witchel, E, Rhee, J. and Asanovic, K.: Mondrix - memory isolation for linux using mondriaan memory protection, the 20th ACM Symposium on Operating Systems Principles (SOSP), pp. 31 - 49, (2005).

[77] Castro, M., Costa, M., Martin, J., Peinado, M., Akritidis, P., Donnelly, A., Barham, P. and Black, R.: Fast byte-granularity software fault isolation, the ACM 22nd Symposium on Operating Systems Principles (SOSP), pp. 45 - 58, (2009).

[78] Hsu. C, T., Hoffman, K., Eugster, P. and Payer M.: Enforcing Least Privilege Memory Views for Multithreaded Applications, the 2016 ACM Conference on Computer and Communications Security (CCS), pp. 393 - 405, (2016).

[79] Litton, J., Vahldiek-Oberwagner, A., E. Elnikety, D. G., Bhattacharjee, B. and Druschel, P.: Light-Weight Contexts - An OS Abstraction for Safety and Performance, 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), (2016).

[80] Boyd-Wickizer, S. and Zeldovich, N.: Tolerating Malicious Device Drivers in Linux, USENIX Annual Technical Conference (ATC), (2010).

[81] Tian, J, D., Hernandez, G., Choi, I, J., Frost, V., Johnson, C, P. and Butler, B, R, K.: LBM: A Security Framework for Peripherals within the Linux Kernel, 2019 IEEE Symposium on Security and Privacy, (2019).

[82] Hund, R., Willems, C. and Holz, T.: Practical Timing Side Channel Attacks against Kernel Space ASLR, 2013 IEEE Symposium on Security and Privacy, pp. 191-205, (2013)

[83] Jang, Y., Lee, S. and Kim, T.: Breaking Kernel Address Space Layout Randomization with Intel TSX, the 2016 ACM Conference on Computer and Communications Security (CCS), pp. 380 - 392, (2016).

[84] Carlini, N., Barresi, A., Payer, M., Wagner, D. and Gross, T. R.: Control-Flow Bending: On the Effectiveness of Control-Flow Integrity, the 24th USENIX Conference on Security Symposium, pp. 161 – 176, (2015).

[85] Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), the 14th ACM Conference on Computer and Communications Security (CCS), pp. 552 - 561, (2007).

[86] Song, D., Hetzelt, F., Das, D., Spensky, C. and Na, Y.: PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary, the 26th Annual Network and Distributed System Security Conference (NDSS), (2019).

[87] Seshadri, A., Luk, M., Qu, N. and Perrig, A.: SecVisor - a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes, the 21st ACM Symposium on Operating Systems Principles (SOSP), pp. 335 - 350, (2007).

[88] Azab, A., Swidowski, K., Bhutkar, R, Ma, J., Shen, W., Wang., R. and Ning, P.: SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM, the 2011 Network and Distributed System Security Symposium (NDSS), (2016).

[89] Cho, Y., Kwnon, D., Yi, H. and Paek, Y.: Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM, the 2017 Network and Distributed System Security Symposium (NDSS), (2017).

[90] McCune, M, J., et al.: TrustVisor - Efficient TCB Reduction and Attestation, 2010 IEEE Symposium on Security and Privacy, (2010).

[91] Koromilas, L., Vasiliadis, G., Athanasopoulos, E. and Ioannidis, S.: GRIM - Leveraging GPUs for Kernel Integrity Monitoring, the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID), pp. 3 - 23, (2016).

[92] Trusted computing group. tpm main specification, available from http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 2003, (accessed 2018-08-10).

[93] Sharif, I, M., Lee, W., Cui, W. and Lanzi, A.: Secure in-VM monitoring using hardware virtualization, the 16th ACM Conference on Computer and Communications Security (CCS), (2009).

[94] Deng, L., Liu, P., Xu, J., Chen, P. and Zeng, Q.: Dancing with Wolves: Towards Practical Event-driven VMM Monitoring, the 13th ACM SIGPLAN/SIGOPS International Conference, (2017).

[95] Zhang, Z., Cheng, Y., Nepal, S., Liu, D., Shen, Q. and Rabhi, F.: KASR: A Reliable and Practical Approach to Attack Surface Reduction of Commodity OS Kernels, the 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID), (2018)

[96] Bailey, M., Oberheide, J., Andersen, J. and Mao, M, Z.: Automated Classification and Analysis of Internet Malware, 10th Symposium on Recent Advances in Intrusion Detection (RAID), (2007).

[97] Ingham, L, K. and Inoue, H.: Comparing Anomaly Detection Techniques for HTTP, 10th Symposium on Recent Advances in Intrusion Detection (RAID), (2007).

[98] Wehner, S.: Analyzing Worms and Network Traffic using Compression, Journal of Computer Security, Vol. 15, No. 3, pp. 303 - 320, (2007).

[99] Gu, G., Zhang, J. and Lee, W.: BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic, 18th Network and Distributed System Security Symposium (NDSS), (2008).

[100] Bayer, U., Comparetti, M, P., Hlauschek, C., Krugel, C. and Kirda, E.: Scalable, Behavior-Based Malware Clustering, 18th Network and Distributed System Security Symposium (NDSS), (2009).

[101] Chung, Y, J., Park, B., Won, J, Y., Strassner, J. and Hong, W, J.: Traffic Classification Based on Flow Similarity, IP Operations and Management, 9th IEEE International Workshop (IPOM), (2009).

[102] Coull, E, S., Monrose, F. and Bailey, M.: On Measuring the Similarity of Network Hosts: Pitfalls, New Metrics, and Empirical Analyses, 18th Network and Distributed System Security Symposium (NDSS), (2009).

[103] Gu, G., Perdisci, R., Zhang, J. and Lee, W.: BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection, the 17th USENIX Conference on Security Symposium, (2008).

[104] Perdisci, R., Lee, W. and Feamster, N.: Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces, 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI), (2010).

[105] Newsome, J., Karp, B. and Song, D.: Polygraph: Automatically Generating Signatures for Polymorphic Worms, IEEE Security and Privacy (S&P), (2005).

[106] Li, Z., Sanghi, M., Chen, Y., Kao, Y, M. and Chavez, B.: Hamsa*: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience, IEEE Security and Privacy (S&P) (2006).

[107] Kong, D., Jhi, C, Y., Pan, Q., Zhu, S., Liu, P. and Xi, H.: SAS: Semantics Aware Signature Generation for Polymorphic Worm Detection, 6th International ICST Conference on Security and Privacy in Communication Networks (SecureComm), (2010).

[108] Ingols, K., Lippmann, R. and Piwowarski, K.: Practical Attack Graph Generation for Network Defense, the 22th Annual Computer Security Applications Conference (ACSAC), (2006).

[109] Wondracek, G., Comparetti, M, P., Krugel, C. and Kirda, Engin.: Automatic Network Protocol Analysis, 16th Annual Network and Distributed System Security Symposium (NDSS), (2008).

[110] Cui, W., Kannan, J. and Wang, J. H.: Discoverer: Automatic Protocol Reverse Engineering from Network Traces, the 16th USENIX Conference on Security Symposium, (2007).

[111] Lin, Z., Jiang, X., Xu, D. and Zhan, X.: Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution, 16th Annual Network and Distributed System Security Symposium (NDSS), (2008).

[112] Comparetti, M, P., Wondracek, G., Kruegel, C. and Kirda, E.: Prospex: Protocol Specification Extraction, IEEE Security and Privacy (S&P), (2009).

[113] Kolbitsch, C., Comparetti, M. P., Krugel, C., Kirda, E., Zhow, X. and Wang, X.: Effective and Efficient Malware Detection at the End Host, the 18th USENIX Conference on Security Symposium, (2009).

[114] Li, Z., Zhang, K., Xie, Y., Yu, F. and Wang, X.: Knowing Your Enemy: Understanding and Detecting Malicious Web Advertising, 19th ACM Conference on Computer and Communications Security (CSS), (2012).

[115] Enck, W., Gilbert, P., Chun, B., Cox, P, L., Jung, J., McDaniel., P. and Sheth, N, A.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones, 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), (2010).

[116] Zhemin, Y., Min, Y., Yuan, Z., Guofei, G., Peng, N. and Sean, X, W.: AppIntent: analyzing sensitive data transmission in android for privacy leakage detection, 20th ACM Conference on Computer and Communications Security (CCS), (2013).

[117] Enck, W., Ongtang, M. and McDaniel, P.: On Lightweight Mobile Phone Application Certification, 16th ACM Conference on Computer and Communications Security (CCS), (2009).

[118] Ongtang, M., McLaughlin, S., Enck, W. and McDaniel, P.: Semantically Rich Application-Centric Security in Android, The 25th Annual Computer Security Applications Conference (ACSAC), (2009).

[119] Nauman, M., Khan, S., Alam, M. and Zhang, X.: Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints, 4th ACM Symposium on Information, Computer and Communications Security (AsiaCCS), (2009).

[120] Jeon, J., Michinsk, K, K., Vaughan, A, J., Reddy, N., Zhu, Y., Foster, S, J. and Millstein, T.: Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android, University of Maryland, Department of Computer Science Technical Report (CS-TR-5006), (2012).

[121] Xu, R., Saidi, H. and Anderson, R.: Aurasium: Practical Policy Enforcement for Android Applications, the 21st USENIX Conference on Security Symposium, (2012).

[122] Hao, H., Singh, Vicky. and Du, W.: On the Effectiveness of API-Level Access Control Using Bytecode Rewriting in Android, 8th ACM Symposium on Information, Computer and Communications Security (AsiaCCS), (2013).

[123] Xiao, Z., Amit, A. and Wenliang. D.: AFrame: Isolating Advertisements from Mobile Applications in Android, the 29th Annual Computer Security Applications Conference (ACSAC), (2013).

[124] Rosen, S., Qian, Z. and Mao, M.: AppProfiler: A Flexible Method of Exposing Privacy-Related Behavior in Android Applications to End users, Third ACM Conference on Data and Application Security and Privacy (CODASPY), (2013).

[125] Toubiana, V., Narayanan, A., Boneh, D., Nissenbaum, H. and Barocas, S.: Adnostic: Privacy Preserving Targeted Advertising, 17th Network and Distributed System Security Symposium (NDSS), (2010).

[126] Bilenko, M., Richardson, M. and Tsai Y, J.: Targeted, Not Tracked: Client-side Solutions for Privacy-Friendly Behavioral Advertising, The 11th Privacy Enhancing Technologies Symposium (PETS), (2011).

[127] Barrera, D., Kayacik, G, H., Oorschot, P, C. and Somayaji, A.: A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android, 17th ACM Conference on Computer and Communications Security (CCS), (2010).

[128] Gilbert, P., Chun G, B., Cox, P, L. and Jung, J.: Automated Security Validation of Mobile Apps for App Markets, Mobile Cloud Computing and Services, (2011).

[129] Schlegel, R., Zhang, K., Zhow, X., Intwala, M., Kapadia, A. and Wang X.: Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones, 18th Network and Distributed System Security Symposium (NDSS), (2011).

[130] Book, T., Pridgen, A. and Wallach, S, D.: Longitudinal Analysis of Android Ad Library Permissions, Mobile Security Technologies (MoST), (2013).

[131] Felt, P, A., Wang, J, H., Moshchuk, A., Hanna, S. and Chin, E.: Permission Re-Delegation: Attacks and Defenses, the 20th USENIX Conference on Security Symposium, (2011).

[132] Lipp, M., et al.: Meltdown - Reading Kernel Memory from User Space, the 27th USENIX Conference on Security Symposium, (2018).

[133] CVE-2016-5195, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195. (accessed 2019-06-05).

[134] Google, inc. Android Developers Manifest.permission, available from http://developer.android.com/reference/android/Manifest.permission.html, (accessed 2014-11-12).

[135] Lookout, inc.: Lookout's take on the Apperhand SDK (aka Android.Counterclank), available from https://blog.lookout.com/blog/2012/01/27/lookout's-take-on-the-'apperhand'-sdk-aka-android-counterclank/, (accessed 2014-12-12).

[136] Cilibrasi, L, R.: Statistical Inference Through Data Compression, Ph.D Thesis, Amsterdam Universitity, (2007).

[137] Admob, available from http://www.google.com/ads/admob/, (accessed 2014-11-12).

[138] Mobclix, available from http://www.mobclix.com/, (accessed 2014-11-12).

[139] Flurry, available from http://www.flurry.com/, (accessed 2014-11-12).

[140] Conte, D., Foggia, P., Sansone, C. and Vento, M.: Thirty years of Graph Matching in Pattern Recognition, International Journal of Pattern Recognition and Artificial Intelligence, Vol. 18, No. 3, pp. 265 - 298, (2004).

[141] Bunke, H. and Shearer, K.: A graph distance metric based on the maximal common subgraph, Pattern Recognition, Vol. 19, Issue 3 - 4, pp.255 - 259, (1998).

[142] Ullman, R, J.: An algorithm for subgraph isomorphism, Journal of the Association for Computing Machinery, Vol. 23, No. 1, pp. 31 - 42, (1976).

[143] Narseo, R. V., Jay, S., Alessandro, F., Yan, G., Konstantina, P., Hamed, H. and Jon., C.: Breaking for commercials: characterizing mobile advertising, the 2012 ACM conference on Internet Measurement Conference (IMC), (2012).

[144] Messmer, T, B. and Bunke, H.: Subgraph Isomorphism in Polynomial Time, Technical Report IAM-95-003, University of Bern, (1995)