2012

Novel Techniques of Using Diversity in Software Security and Information Hiding

Jin HAN
Singapore Management University, jin.han.2007@smu.edu.sg

Novel Techniques of Using Diversity in Software Security and Information Hiding

HAN Jin

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee

Debin GAO (Supervisor / Chair)
Assistant Professor of Information Systems
Singapore Management University

Robert DENG Huijie (Co-Supervisor)
Professor of Information Systems
Singapore Management University

Yingjiu LI
Associate Professor of Information Systems
Singapore Management University

Jianying ZHOU
Senior Scientist
Institute for Infocomm Research

Singapore Management University

2012

Abstract

Diversity is an important and valuable concept that has been adopted in many fields to reduce correlated risks and to increase survivability. In information security, diversity also helps to increase both defense capability and fault tolerance for information systems and communication networks, where diversity can be adopted from many different perspectives. This dissertation, in particular, focuses mainly on two aspects of diversity – the application software diversity and the diversity in data interpretation.

Software diversity has many advantages over mono-culture in improving system security. A number of previous researches focused on utilizing existing off-the-shelf diverse software for network protection and intrusion detection, many of which depend on an important assumption – the diverse software utilized in the system is vulnerable only to different exploits. In the first work of this dissertation, we perform a systematic analysis on more than 6,000 vulnerabilities published in 2007 to evaluate the extent to which this assumption is valid. Our results show that the majority of the vulnerable application software products either do not have the same vulnerability, or cannot be compromised with the same exploit code.

Following this work, we then propose an intrusion detection scheme which builds on two diverse programs to detect sophisticated attacks on security-critical data. Our model learns the underlying semantic correlation of the argument values in these programs, and consequently gains more accurate context information compared to existing schemes. Through experiments, we show that such context information is effective in detecting attacks which manipulate erratic arguments with

comparable false-positive rates.

Software diversity does not only exist on desktop and mainframe computers, it also exists on mobile platforms like smartphone operating systems. In our third work in this dissertation, we propose to investigate applications that run on diverse mobile platforms (e.g., Android and iOS) and to use them as the baseline for comparing their security architectures. Assuming that such applications need the same types of privileges to provide the same functionality on different mobile platforms, our analysis of more than 2,000 applications shows that those executing on iOS consistently ask for more permissions than their counterparts running on Android. We additionally analyze the underlying reasons and find out that part of the permission usage differences is caused by third-party libraries used in these applications.

Different from software diversity, the fourth work in this dissertation focuses on the diversity in data interpretation, which helps to defend against coercion attacks. We propose Dummy-Relocatable Steganographic file system (DRSteg) to provide deniability in multi-user environments where the adversary may have multiple snapshots of the disk content. The diverse ways of interpreting data in the storage allows a data owner to surrender only some data and attribute the unexplained changes across snapshots to the dummy data which are random bits. The level of deniability offered by our file system is configurable by the users, to balance against the resulting performance overhead. Additionally, our design guarantees the integrity of the protected data, except where users voluntarily overwrite data under duress.

This dissertation makes valuable contributions on utilizing diversity in software security and information hiding. The systematic evaluation results obtained for mobile and desktop diverse software are important and useful to both research literature and industrial organizations. The proposed intrusion detection system and steganographic file system have been implemented as prototypes, which are effective in protecting valuable user data against adversaries in various threat scenarios.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank Professor Steven MILLER, Professor Robert DENG, Assistant Professor Debin GAO, Associate Professor Yingjiu LI and Senior Scientist Doctor Jianying ZHOU for their guidance in completing my dissertation.

I also thank my friends YAN Qiang and KOH Noi Sian for the research collaboration, their friendship, and their encouragement.

Finally, I would like to thank my parents, who are always supporting me and encouraging me with their best wishes.

Dedication

I dedicate my dissertation work to my loving parents, HAN Yandong and LI Aiqin.

Thanks, Mum and Daddy.

Chapter 1

Introduction

Diversity, in general, is an important and valuable concept that has been adopted in many fields to reduce correlated risks and to increase survivability. For example, individuals and financial institutions spread their portfolio among multiple investment vehicles so that the fluctuations of a single investment will have less impact on a diverse portfolio [90]. In agriculture, crop diversity is necessary to prevent the attack of diseases on the dominant crop type. The Irish potato blight of 1846 that caused the deaths of over one million people was the result of planting only two potato varieties, both of which were vulnerable to the Phytophthora infestans mold [70].

In information security, diversity also helps to increase both defense capability and fault tolerance for information systems and communication networks, where diversity can be adopted from many different perspectives. For example, operating systems, application software, routing protocols, service models, and encryption mechanisms used in an information system can all be diversified to increase the security and survivability of the system. This dissertation, in particular, focuses mainly on two aspects of diversity – the application software diversity and the diversity in data interpretation, which will be introduced in the following two sections, correspondingly.

1.1 Diversity in Software Security

Software diversity has many advantages over mono-culture in improving system security [55, 99]. A number of previous researches focused on utilizing existing off-the-shelf diverse software for network protection [82, 112] and intrusion detection [67, 87, 101, 51], many of which depend on an important assumption – the diverse software utilized in the system is vulnerable only to different exploits. With this assumption, replicas constructed using diverse software will not be compromised by the same attack. However, to the best of our knowledge, there has not been a systematic analysis in evaluating the extent to which this assumption is correct.

Thus, in this study, our first work is to perform a systematic analysis on the vulnerabilities published in 2007 to evaluate the extent to which this assumption is valid. We focus on vulnerabilities in application software, and show that the majority of these software products either do not have the same vulnerability or cannot be compromised with the same exploit. We also find evidence that indicates the use of diversity in increasing attack tolerance for other software such as hardware-specific or OS-specific software. These results show that systems utilizing off-the-shelf software products to introduce diversity are effective in detecting intrusions.

Following this work, we then propose an intrusion detection scheme which builds on two diverse programs to detect sophisticated attacks on security-critical data. In many programs, the value of a function argument in one normal program execution could become illegal in another normal execution context. Attacks utilizing such erratic arguments are able to evade detections as fine-grained context information is unavailable in many existing detection schemes [23, 72, 100]. In order to obtain such fine-grained context information, a precise model on the internal program states has to be built, which is impractical especially when monitoring a closed source program alone. Thus, we propose an intrusion detection scheme utilizing two diverse programs providing semantically-close functionalities. Our model learns the underlying semantic correlation of the argument values in these programs, and consequently gains more accurate context information compared to existing schemes. Through experiments, we show that such context information is effective in detecting attacks which manipulate erratic arguments with comparable false-positive rates.

Software diversity does not only exist on desktop and mainframe computers. It also exists in mobile platforms like smartphone operating systems. The fast growing mobile device market has brought intense competition among smartphone manufacturers, which has led to a variety of smartphone platforms upon which mobile applications are developed. Security analysis of these platforms and their influence on the corresponding applications as well as application development processes has started to gain momentum in recent years. However, the lack of baseline for comparison makes such analysis difficult.

Thus, in our third work in this dissertation, we propose to investigate applications that run on diverse mobile platforms (e.g., Android and iOS) and to use them as the baseline for comparing their security architectures. Assuming that such applications need the same set of privileges to provide the same functionalities on different mobile platforms, our analysis of more than 2,000 applications shows that those executing on iOS consistently ask for more permissions than their counterparts running on Android. The additional permissions required are mainly on accessing private data such as device ID, user contacts and calendar. We further investigate the underlying reasons and find out that part of the permission usage differences are caused by third-party libraries used in these applications. Finally, the application permissions that are not supported on iOS but are requested by third-party Android applications are also studied and the underlying rationale is revealed.

1.2 Diversity in Information Hiding

Different from software diversity, the fourth work in this dissertation focuses on the diversity in data interpretation, which helps to defend against coercion attacks. By utilizing existing steganographic file systems (stegfs), the same set of data on disk can be interpreted in different ways given different decryption passwords. Such a system hides encrypted user data among dummy data that contain only random bits. Without the correct password, it is not possible to differentiate user data from dummy (based on the assumption that the output of a block cipher is indistinguishable from random bits [18, 19]), even for an adversary who understands the mechanisms of the file system and is able to gain access to the storage devices. This feature allows a data owner to selectively reveal some directories/files, but disclaim the existence of his sensitive data.

To be believable, the disclaimer of the data owner must be consistent with the information that the adversary is able to gather about the file system. This is much more challenging to achieve in modern computing environments when the user data are encrypted and stored in shared network storage, where the adversary is no longer limited to a single snapshot of the disk content at the point of attack. Instead, the adversary could now locate the physical server machines being used [89] and quietly amass multiple snapshots of the file system over a period of time before launching his attack. In earlier stegfs designs [19, 79, 64, 85], dummy data are created when the disk is formatted and remain static thereafter. Thus, the differences across multiple snapshots will disclose the locations of user data, and could even reveal the user passwords.

In order to address such threats, we introduce a Dummy-Relocatable Steganographic (DRSteg) file system to provide deniability in multi-user environments where the adversary may have multiple snapshots of the disk content. With its novel techniques for sharing and relocating dummy data during runtime, DRSteg allows a data owner to surrender only some data and attribute the unexplained changes across snapshots to the dummy operations. The level of deniability offered by DRSteg is configurable by the users, to balance against the resulting performance overhead. Additionally, DRSteg guarantees the integrity of the protected data, except where users voluntarily overwrite data under duress.

1.3 Contributions and Organization

To summarize, the following contributions have been made in this dissertation:

- We systematically analyzed more than $6,000$ vulnerabilities published in the year of 2007, to validate the assumption that diverse software which provides similar functionalities is vulnerable only to different exploits. Our results show that the majority of the vulnerable application software products either do not have the same vulnerability, or cannot be compromised with the same exploit code.

- We proposed an intrusion detection scheme which builds on two diverse programs providing semantically-close functionalities to detect sophisticated attacks. Our model learns the underlying semantic correlation of the argument values in these programs, and consequently gains more accurate context information, which is effective in detecting attacks that manipulate erratic arguments.

- We investigated the detailed iOS application permissions, and compared them to Android permissions. We also performed static analysis on over 1,000 pairs of applications that run on Android and iOS, the results of which reveal the detailed permission usage differences for Android and iOS third-party applications.

- Finally, we introduced a Dummy-Relocatable Steganographic file system to provide deniability in multi-user environments where the adversary may have multiple snapshots of the disk content. The diverse ways of interpreting data in the storage allows a data owner to surrender only some data and attribute the unexplained changes across snapshots to the dummy data which are random bits.

The rest of this dissertation is organized as follows: Chapter 2 reviews the existing studies from three perspectives – software diversity, mobile security and steganographic file systems. Chapter 3 introduces our first work, which is the systematic study on vulnerabilities in diverse software. The technique of utilizing diverse application programs to construct intrusion detection systems is then proposed in Chapter 4. Our third piece of work is presented in Chapter 5, which is another empirical study. This study focuses on analyzing diverse mobile applications to compare the security architectures of diverse smartphone platforms. Our fourth work which utilizes the data interpretation diversity in information hiding is presented in Chapter 6. Finally, Chapter 7 concludes the contribution of this dissertation and describes the future direction of the current research.

Chapter 2

Literature Review

2.1 Software Diversity for Fault Tolerance and Intrusion Detection

The potential security risk brought by software monoculture has arisen great attention from computer security researchers. Although using homogeneous software could result in improved interoperability and reduced costs, security researchers still claim that the current lack of software diversity is troublesome and believe that a reliable system and network security can only be achieved if a multitude of application software and platforms is utilized [55, 99].

Research work has been done to introduce diversity at the system level through a variety of techniques. Early works on software diversity construct intrusion-tolerance and fault-tolerance systems [29, 88] with software providing semantically-close functionalities. Methods that systematically generate stochastic diversification within source code to increase system resistance and survivability were introduced by [75]. Instruction-set randomization [21, 69] has also been implemented to safeguard systems against code-injection attacks. The basic heterogeneous networking philosophy and models were introduced by [112] to achieve network survivability. Distributed algorithms have also been proposed to improve network security by reducing the ability of an attacker to move from system to system [82].

In the past ten years, diversity-based intrusion detection techniques [34, 51, 52, 68, 102] were also proposed, which use Commercial Off-The-Shelf (COTS) software to build the detection models. Among those schemes, the techniques proposed by Just et al. [68] and Totel et al. [102] are output voting schemes, which only compare the final outputs (HTTP status codes and files) of the diverse software to detect intrusions. However, as many of the intrusions may not result in observable deviation in the responses of those server software, such intrusions can evade detections of these techniques.

Behavioral Distance model by Gao et al. [51, 52, 53] was later proposed to defend against stealthy attacks which are not addressed by both the output voting schemes and traditional intrusion detection techniques which only monitor single application. However, since the hidden Markov model used in their scheme (to train the normal-behavior profiles of the system call sequences) is only able to handle finite states, their model cannot be simply extended to detect attacks utilizing erratic arguments, which is addressed in this dissertation.

2.2 Mobile Application Security

Previous studies on mobile security focus on either Android or iOS platform alone. From programming perspective, Burns [27] provides a useful background for developers of Android applications, which includes discussions of common developer errors, such as using Intent Filters instead of permissions. Enck et al. [43] examine Android security policies and some of the developer pitfalls. They further investigate on the source code of a thousand Android applications [42], which reveals a number of interesting findings for Android application security, some of which are also confirmed by our results on Android. A large-scale evaluation for Android applications is performed by Zhou et al. [114], which reveals hundreds of malicious Android applications.

From the attacker's perspective, Felt et al. [45] demonstrate that a less privileged Android application is able to perform a privileged task through another third-party application with higher privilege, which undermines the requirement for users to approve each application's access to privileged devices and data. A similar privilege escalation attack which is implemented with return-oriented programming is introduced by Davi et al. [35]. The defense mechanism for such privilege escalation attack is proposed by Bugiel et al. [26].

There are also a number of research work focusing on analyzing and improving the Android permission model. Barrera et al. [22] present a methodology for the empirical analysis of permission-based security models, and apply it to Android permission model. In addition, Ongtang et al. [83] present a fine-grained access control policy infrastructure for protecting applications on Android. Hornyack et al. [65] also present two privacy controls to protect user data from exfiltration by permission-hungry applications. A recent work by Grace et al. [59] systematically analyzed several Android smartphone images and found they do not properly enforce the Android permission model.

The most related work to this work on Android application analysis is from Felt et al. [44]. They study Android applications to determine whether Android developers follow least privilege with their permission requests. They develop a static analysis tool Stowaway that detects overprivilege in Android applications. They also create an Android API-to-permission mapping using dynamic testing, and this mapping is used as one of the inputs in our Android static analysis tool. This mapping could also be used in the permission check tool introduced by Vidas et al. [105], which aids developers in specifying a minimum set of permissions required for a given Android application with source code.

From the dynamic analysis perspective, TaintDroid developed by Enck et al. [41] provides a system-wide dynamic taint tracking for Android. TaintDroid is implemented as a post-production tool for real-time analysis and it found 68 potential information misuse examples in 20 applications. However, this tool focuses only on data flow, and does not consider action-based vulnerabilities, which is then improved by ComDroid [31]. Another variant of such dynamic analysis tool on Android is ScanDroid [48].

In comparison to the literature on Android, there are relatively less studies on iOS platform. Seriot [96] demonstrated that any applications downloaded from the iTunes Store to a standard iPhone device can access a significant quantity of personal data. He explains how malicious applications could pass Apple's vetting process unnoticed and harvest data through officially sanctioned iOS APIs. Egele et al. [40] study the privacy threats that third-party iOS applications pose to users. They present a static analysis tool that analyzes programs for possible leaks of sensitive information from a mobile device to third parties. Our iOS static analysis tool adopts a similar mechanism in resolving _objc_msgSend as their tool, but with a different emphasis on resolving the API calls and parameters that are related to application permissions.

To our best knowledge, there is no literature to date that systematically compares the application security of Android and iOS platforms. Our research is the first work that establishes the baseline of examining the massive cross-platform applications to compare the effect of security mechanism utilized by Android and iOS, which reveals interesting behavioral differences for the third-party applications on these two platforms.

2.3 Cryptographic and Steganographic File Systems

Cryptographic file systems (e.g., [25, 28, 60, 111]) and their implementations (e.g., [11, 12]) have been studied extensively in the last two decades . A cryptographic file system complements the access control mechanism of the operating system (OS). Even if the OS is compromised or the data storage is removed from the OS, data in the file system remain protected by the user's password. A weakness of cryptographic file systems is that they leave evidence of the existence of encrypted data, such that a determined attacker may compel the users to reveal their decryption passwords.

Utilizing diversity in the data interpretation to provide plausible deniability of the secret data was first proposed by Anderson et al. [19], where two steganographic file system (stegfs) schemes were introduced. In the first scheme, the disk is initialized with several cover files that have equal length and contain random data. A secret object is stored through an exclusive-or operation on a subset of the cover files, identified by the corresponding bits in the access key. To protect against brute force attacks, the number of cover files must be sufficiently large; this imposes heavy I/O overheads as each read/write request for an object translates into operations on multiple cover files. The scheme is effective against single-snapshot attacks but not multiple-snapshot attacks. In particular, the differences between just two snapshots of the storage can expose the access key used.

In Anderson's second scheme [19], the disk is first filled with random bits. Subsequently, secret data blocks are written to pseudorandom addresses. An implementation of this scheme on Linux is reported by McDonald et al. [79], a peer-to-peer version by Hand et al. [64] and a distributed version by Giefer et al. [57]. The disadvantage of the scheme is that the probability of collision in the locations where data are stored increases as more data are added to the disk. Although replicating each data block in different locations reduces the likelihood of data loss, the risk cannot be eliminated; hence data integrity is not guaranteed.

Pang et al. [85] utilized a bitmap to track block allocation to avoid overwriting data and to improve system performance. To defend against single-snapshot attacks, dummy data are added when the disk is initialized. The dummy data cannot be changed or relocated at runtime, so the scheme is susceptible to multiple-snapshot attacks. Zhou et al. [113] provided for the relocation of dummy blocks. Their solution requires a trusted agent to manage all the user passwords and dummy data, which effectively transfers the risk of password disclosure to the agent.

Diaz et al. [37] proposed to defend against traffic analysis [103] through a mix-based stegfs that employs a local mix to relocate files in the remote storage. They show that the security of the scheme depends on the file-size patterns in the system. Another work by Domingo-Ferrer et al. [38] addressed the problem of data loss in a stegfs with multiple users. It is not designed to defend against multiple-snapshot attacks though. Furthermore, neither of the two schemes guarantees data integrity under legitimate data operations.

TrueCrypt[1], an open-source disk-encryption software package, enables a user to create a deniable file system within a regular encrypted file or partition. The file system is deniable if the adversary only sees the final content of the disk. However, it cannot defend against an adversary who possesses multiple snapshots of the encrypted partition. The same weakness exists in similar products that provide deniability for secret files, e.g., Phonebook[2] and Rubberhose[3].

Different from existing schemes, our proposed stegfs is designed to defend against multiple-snapshot attacks. It employs novel techniques to share and relocate dummy data at runtime, which enables users to surrender only some of their data, and attribute any unexplained changes across snapshots to dummy operations. The deniability provided by our stegfs is configurable individually, and our proposed stegfs also guarantees the integrity of the protected data, except where users voluntarily overwrite data under duress.

[1]TrueCrypt, http://www.truecrypt.org/
[2]Phonebook, http://www.freenet.org.nz/phonebook
[3]Rubberhose, http://iq.org/~proff/rubberhose.org

Chapter 3

A Systematic Study on Vulnerabilities in Diverse Software

3.1 Introduction

Software diversity has many advantages over mono-culture in improving system security [55, 99]. Linger [75] proposed methods that systematically generate stochastic diversification in program source to increase system resistance and survivability. Obfuscation techniques (e.g., instruction-set randomization [21, 69] and address space randomization [24]) were proposed to safeguard systems against code-injection attacks and other memory error exploits. N-variant systems [33] execute a set of automatically diversified variants on the same inputs, and monitor their behavior to detect divergence that signals anticipated types of exploits, against which the variants are diversified.

Instead of artificially introducing diversity, some recent work focused on utilizing existing diverse software for network protection [82] and intrusion detection [51]. Some of these systems (e.g., the HACQIT system [67, 87] and its successor [101]) employed output voting to monitor outputs from diverse replicas, while others (e.g., Behavioral Distance [51, 52, 53]) monitor the low-level behavior of the diverse replicas.

An interesting and important assumption made by many of these systems utilizing off-the-shelf diverse software is that the diverse software is vulnerable only to different exploits. With this assumption, replicas constructed using diverse off-the-shelf software will not be compromised by the same attack. This is a reasonable assumption because most of the off-the-shelf diverse software is developed independently by different groups of developers, and so the same mistake/vulnerability is unlikely to be introduced. However, to the best of our knowledge, there has not been a systematic analysis to evaluate the extent to which this assumption is correct. Such analysis also guides users in choosing between artificially introducing diversity (e.g., instruction-set randomization, address space randomization, and N-variant systems) and utilizing off-the-shelf software products to introduce diversity.

In this chapter, we present a systematic analysis on the effectiveness of utilizing off-the-shelf diverse software for improving system security. In particular, we evaluate the extent to which different off-the-shelf software suffers from the same vulnerability and exploit. This is achieved by carefully analyzing over 6,000 vulnerabilities published in the year of 2007.

To get a better idea of what is to be analyzed and how this analysis benefits systems that utilize off-the-shelf diverse software, consider an example in which a system uses behavioral distance [51, 52, 53] for intrusion detection (see Figure 3.1). In this example, a web service is provided by two diverse web servers running on two diverse operating systems. The same input, which may potentially be an attack input, is processed by both servers. Similar architectures, e.g., diverse servers on the same operating system, have also been introduced [67, 87, 101].

This system detects an intrusion when deviations are found in the two replicas when they are processing the same input. Such deviations may be detected in server outputs [67, 87, 101] or in the low-level behavior, e.g., system calls [51, 52, 53]. A very important observation is that such deviations occur only if the two replicas behave differently when processing the same malicious input. The system assumes that either the two replicas do not have the same vulnerability, or they cannot be

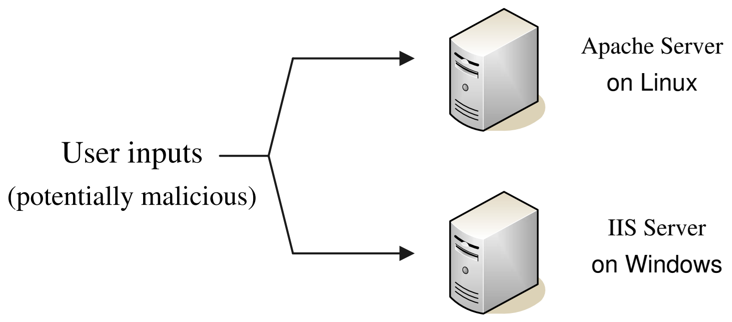


Figure 3.1: An example (Behavioral Distance) of utilizing off-the-shelf diverse software

exploited simultaneously with a single attack.

In order to evaluate the extent to which this assumption is valid, several questions need to be answered:

- Among the large number of vulnerable software products, how many of them have potential substitutes that provide similar functionalities? For those that are software substitutes of one another, do they have the same vulnerability? If they do have the same vulnerability, can they be exploited with the same attack?

- Among the large number of vulnerable software products, how many of them can run on multiple operating systems? For those that run on multiple operating systems, do vulnerabilities of the software on one operating system propagate to the same software on a different operating system? If so, can they be exploited by the same attack when running on different operating systems?

To the best of our knowledge, there is no closely related work which could answer these questions. We systematically analyzed more than $6,000$ vulnerabilities published in the year of 2007. In summary, our results show that more than 98.5% of the vulnerable application software products have software substitutes (and therefore can be used in a replicated system to detect intrusion), and the majority of them either do not have the same vulnerability, or cannot be compromised with the same exploit code. In addition, among the application software products, nearly half are officially supported to run on multiple operating systems. Although the different operating system distributions of the same product are likely (more than 80%) to suffer from the same vulnerability, the attack code is different in most cases. We also found evidence that indicates the use of diversity in increasing attack tolerance in other categories of vulnerable software.

It is not the objective of this work to build systems utilizing software diversity or to evaluate how difficult it is to manage such systems. Instead, we measure the extent to which software diversity could be utilized to increase system security in using off-the-shelf software products.

In the rest of this chapter, we first present the data source we utilized and some preliminary analysis (see Section 3.2). We then focus our analysis on the application software vulnerabilities in which we analyzed whether diverse software products providing the same services could suffer from the same vulnerability (see Section 3.3), and whether the same software product running on different operating systems will suffer from the same vulnerability and exploit (see Section 3.4). In Section 3.5, we present analysis on other vulnerable software products. Finally, we conclude in Section 3.6.

3.2 Source of Information and Preliminary Analysis

The main source of information we used for our analysis was the NVD/CVE (National Vulnerability Database/Common Vulnerabilities and Exposures) vulnerability database. We analyzed all the vulnerabilities recorded in CVE in the year of 2007, which consist of 6,427 vulnerability entries[1]. To obtain detailed information on the vulnerabilities and the corresponding software products, we also consulted other sources including SecurityFocus, FrSIRT, CERT, Milw0rm, Secunia, OS-VDB, IBM X-Force, as well as vulnerability advisories, security announcements,

[1]The CVE 2007 database published on April 25, 2008 was used (http://nvd.nist.gov/download/nvdcve-2007.xml).

and bug lists from software vendors. After removing 87 entries that were rejected by CVE, the total number of vulnerabilities that we focused on was 6,340.

Note that the limited information introduced errors in our analysis. First, not all vulnerabilities are published. We only analyzed vulnerabilities found and published in 2007. Second, we may not have found all information on some published vulnerabilities. This is due to the limited resources we have, although we did our best in searching various public resources; it might also be the fact that some information about the vulnerabilities is not publicly available.

Our first step in the analysis was to find whether the vulnerable software has any substitutes (software products that offer similar functionalities). We also categorized the vulnerabilities into five different types for further analysis.

3.2.1 Software without substitutes

To implement a replicated system with diverse replicas (e.g., the one shown in Figure 3.1), we need to find (at least) two software products that provide the same service (*software substitutes*) and/or software products that run on multiple operating systems. If the software product does not have any substitutes and runs only on a single operating system, then diversity using off-the-shelf software cannot work and one has to introduce diversity via other artificial means (e.g., address space randomization). Therefore, we first analyze all the vulnerable software products in the CVE database to see if they have any substitutes.

We find that most software products do have substitutes and those that do not have mostly fall into one of the following three categories:

- **Hardware specific software**: This includes hardware drivers and firmware only provided by corresponding hardware vendors.

- **OS specific software**: This includes utilities that are specific to an operating system, e.g., Mac Installer, Windows Login window. They are only provided by the OS vendor.

- **Domain specific and customized software**: This includes software that is used in medical, biological, nuclear and other specific domains. The customized software refers to that developed for a specific company, e.g., management software that is used in a specific company; ActiveX controls developed and used for online transactions on a specific web site.

Vendor	Product	CVE entry
ATI	Display driver	CVE-2007-4315
NVIDIA	Video driver	CVE-2007-3532
Intel	2200BG Wireless driver	CVE-2007-0686
HP	Help and Support Center	CVE-2007-3180
HP	Quick Launch Button	CVE-2007-6331
Alibaba	Alipay ActiveX control	CVE-2007-0827
Microgaming	Download Helper ActiveX	CVE-2007-2177

Table 3.1: Examples of software products without substitutes

Table 3.1 shows some examples of software products that do not have substitutes. An interesting observation is that we did not find many vulnerable software products from the CVE database that are domain specific or customized. This does not necessarily mean that these software products do not have vulnerabilities. Domain specific and customized software products are used in a more controlled environment and it is less likely that they are reported in public vulnerability resources.

3.2.2 Vulnerable software categorization

Some vulnerabilities exist in application software that runs as user-space programs on an operating system. Others may exist in scripts that run on top of another software program. The analysis we performed varies according to the type of vulnerable software products. Therefore, we first put the vulnerable software into different categories.

- **Application software**: Application software is the most interesting because it is relatively easy to find the software substitutes. It is usually compiled into binary format and run as a process of its own in the user space. Word processors, web browsers, web servers and computer games are some examples of application software. It also includes plug-ins, extensions, and add-ons to application software, except those for a web server (see the next category).

- **Web script modules**[2]: These are light-weighted software modules which only run on web servers. We put them into a separate category instead of a sub-category of application software because of the large number of vulnerabilities in them. Examples include Content Management Systems (CMS), forums, bulletin boards, and other script modules.

- **Operating systems**: This category includes the operating system kernel and utilities that are closely related to the operating system, e.g., Apple Installer and the login window of Microsoft Windows.

- **Languages and libraries**: These include programming languages and libraries for general programming use, e.g., PNGlib (for decoding the PNG image) and SMTPlib (for implementing the SMTP protocol).

- **Others**: For example, firmware (including Routers, IP phones, hardware firewalls, etc.), software that runs on mobile phone, video game consoles (e.g., XBox) and so on.

Figure 3.2 shows the number of vulnerabilities in each software category and the corresponding percentage.

3.2.3 Vulnerabilities in application software

As shown in Figure 3.2, 41.4% of the vulnerabilities found in 2007 are in application software. We focus our analysis on this category because it contains most of the commonly used and critical software, and it is usually what an intrusion detection

[2]They may be called web applications (e.g., in SANS [36]). We call this category web script modules, instead, to avoid the misunderstanding that it also contains web servers and browsers.

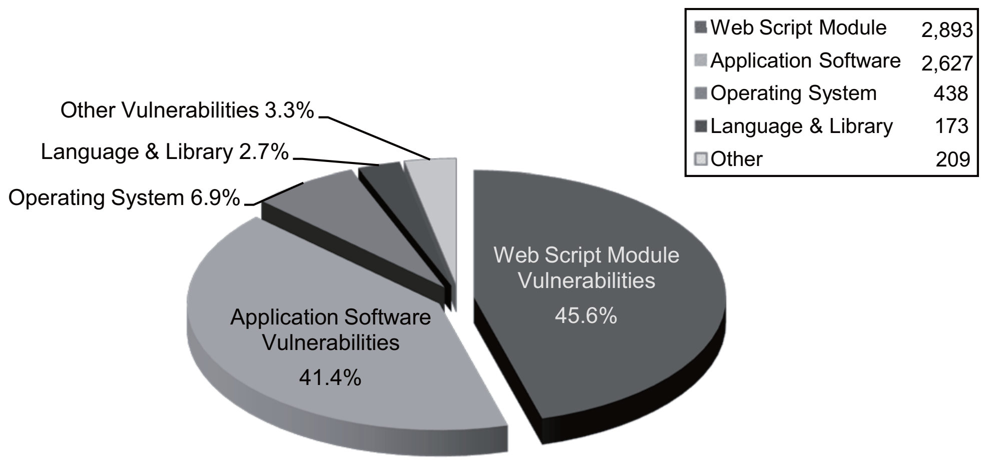


Figure 3.2: Vulnerabilities in different software categories

system tries to protect. Not only that, it is also easy to find substitutes for an application software product, which makes it a natural candidate for introducing diversity. This is also the category for which information is best available and therefore the results of our analysis are most accurate.

The first analysis we did was to find the number of vulnerable application software products that do not have substitutes. As discussed in Section 3.2.1, this is important because one of the two ways of utilizing off-the-shelf software products to introduce diversity is to use software substitutes (the other is to run the same software on multiple operating systems). If many vulnerable application software products do not have any substitutes, then we will have to rely on the other way of introducing diversity.

We found 1,825 distinct application software products in all the 2,627 application software vulnerabilities[3], out of which only 25 (1.4%) do not have software substitutes. Some of the examples were shown in Table 3.1. This result coincides with our expectation in view of the highly competitive software industry market.

We have found that most software products in this category have software substitutes. The next question is whether these software products and their corresponding substitutes have the same vulnerability or not. In order to do this analysis, we fur-

[3]A total number of 4,120 different names of software products were found in the descriptions of these vulnerabilities. Many of them were duplicates with different naming conventions or different product versions. After eliminating these duplicates, we found 1,825 distinct software products.

ther classify the application software vulnerabilities (Box 1 in Figure 3.3) into two sub-categories: vulnerabilities that exist in multiple software products (Box 2) and vulnerabilities that exist in a single software product (Box 3).

The results of the classification are obtained by examining the vulnerable product information and the description of each vulnerability in the CVE database. Figure 3.3 shows that majority of the vulnerabilities (2037 out of 2627) exist in only a single software product, which is an evidence in favor of introducing diversity since the replicas constructed in a replicated system are unlikely to suffer from the same vulnerability. We look into each of the two categories for further analysis.

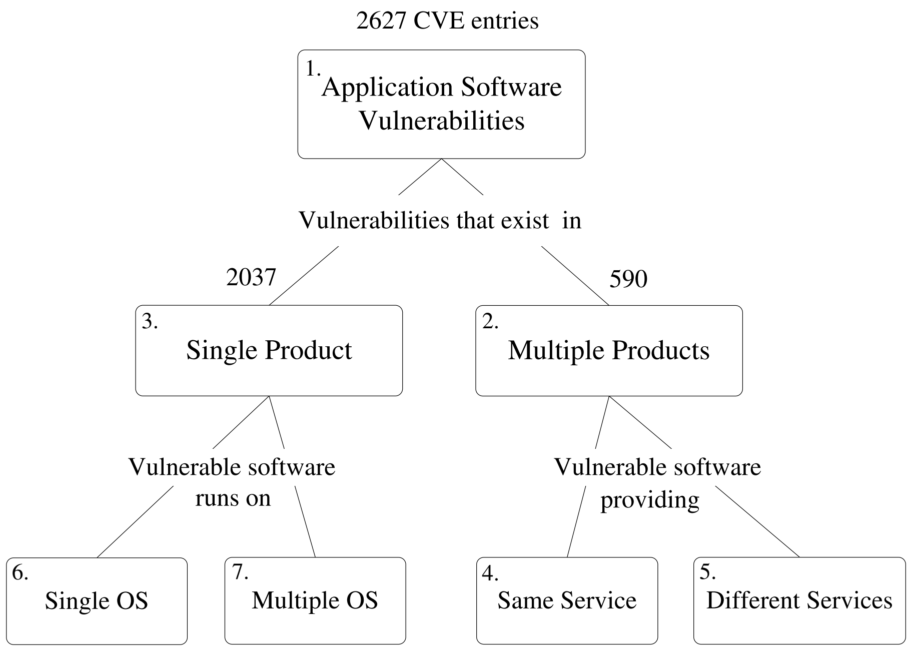

2627 CVE entries
1. Application Software Vulnerabilities
Vulnerabilities that exist in
2037
590
3. Single Product
2. Multiple Products
Vulnerable software runs on
Vulnerable software providing
6. Single OS
7. Multiple OS
4. Same Service
5. Different Services

Figure 3.3: Analysis on application software vulnerabilities

Among the vulnerabilities that exist in multiple software products (Box 2 in Figure 3.3), we want to find out whether software products suffering from the same vulnerability are substitutes of one another (i.e. whether they provide the same service). This analysis is important because only software products providing the same service can be used in an intrusion detection system using software diversity (such as the behavioral distance system shown in Figure 3.1). If software programs and their substitutes suffer from the same vulnerability (Box 4), then such intrusion detection systems will not be effective in detecting intrusions. We present our detailed analysis for this in Section 3.3. If multiple software products – which suffer from the same vulnerability – are not providing the same service (Box 5), then they are not used simultaneously for constructing the intrusion detection system and therefore will not affect the effectiveness of diversity using off-the-shelf software products.

Among those vulnerabilities that exist in a single product (Box 3 in Figure 3.3), we want to find out how many of these software products can execute on multiple operating systems. For those that run on multiple operating systems (Box 7), it is also important to find out whether their vulnerabilities can be exploited in the same way when they are running on multiple operating systems. We present our analysis of these problems in Section 3.4. If a software product can only run on a single operating system (Box 6), then it cannot be used in a replicated system in which replicas are constructed using the different distributions of a single software product on multiple operating systems.

3.3 Vulnerabilities in Software Substitutes

As shown in Figure 3.3, there are 590 entries of vulnerabilities in multiple software products. Each of these vulnerabilities exists in more than one software product, which may or may not provide the same service. In this section, we first briefly show our method for finding vulnerabilities in software substitutes and our findings using this method (Section 3.3.1), and then discuss the attack code for exploiting the same vulnerability in these software substitutes (Section 3.3.2).

3.3.1 Finding vulnerabilities in software substitutes

An interesting observation is that the same vulnerability may be represented in multiple entries in the CVE database. For example, entries CVE-2007-2761 and CVE-2007-2888 correspond to the same vulnerability (see Table 3.2). For this reason, we cannot simply rely on different CVE entries to distinguish different vulnerabilities.

Different CVE entries that refer to the same vulnerability usually have similar descriptions. We use Vector Space Model [91], one of the classical models in information retrieval, to compare the descriptions for all CVE entries. The similarity between two vulnerability descriptions is calculated using

$$\mathsf{sim}(d_1, d_2) = \frac{\overrightarrow{d_1} \cdot \overrightarrow{d_2}}{|\overrightarrow{d_1}| \times |\overrightarrow{d_2}|} = \frac{\sum_{i=1}^{t} w_{i,1} \times w_{i,2}}{\sqrt{\sum_{i=1}^{t} w_{i,1}^2} \times \sqrt{\sum_{i=1}^{t} w_{i,2}^2}}$$

where $\overrightarrow{d_1}$ and $\overrightarrow{d_2}$ are the descriptions of two vulnerability entries, $w_{i,j}$ is the weight for the i^{th} term in description d_j which is assigned with the frequency of the term. The threshold for the similarity score is set to 0.65 by manual tuning to obtain a good trade-off between the number of false positives and false negatives[4].

After the automatic comparison process using Vector Space Model and additional manual verification and correction, 410 distinct vulnerabilities are obtained from the 590 vulnerability entries that exist in multiple software products. We then performed a detailed analysis for each vulnerability and found that 29 of them (which involve 69 CVE entries) fall into the category in which the same vulnerability exists in multiple software products providing the same services (software substitutes). Some examples are shown in Table 3.2.

The result shows that although many vulnerabilities (410) exist in multiple software products, only a small portion of them (29) exist in multiple software products that provide the same service. Note that although the Vector Space Model helped a lot in finding similar descriptions in different vulnerability entries, some manual analysis was needed to obtain the results shown above.

[4]This process is mainly manual inspection, while the automatic Vector Space Model analysis is to assist the manual inspection. A simple false-positive rate does not apply here actually. Several CVE entries may map into the same vulnerability after the automatic analysis, and then I manually verify each group of CVE entries which are regarded as similar by the auto analysis. Sometimes, in the same group, maybe only part of the entries is really the same vulnerability while other entries are not (e.g., 4 entries are categorized as the same vulnerability by auto-analysis while only 2 of them are actually the same vulnerability). Thus, it is very difficult to give a simple false positive number. An approximate false-positive rate is 0.4; while the false-negative rate was not measured, but is believed to be very small (<0.01).

CVE Entry	Description
CVE-2007-2761	Stack-based buffer overflow in MagicISO 5.4 build 239 and earlier allows remote attackers to execute arbitrary code via a long filename in a .cue file.
CVE-2007-2888	Stack-based buffer overflow in UltraISO 8.6.2.2011 and earlier allows user-assisted remote attackers to execute arbitrary code via a long FILE string (filename) in a .cue file.
CVE-2007-0548	KarjaSoft Sami HTTP Server 2.0.1 allows remote attackers to cause a denial of service (daemon hang) via a large number of requests for nonexistent objects.
CVE-2007-3340	BugHunter HTTP SERVER (httpsv.exe) 1.6.2 allows remote attackers to cause a denial of service (application crash) via a large number of requests for nonexistent pages.
CVE-2007-3398	LiteWEB 2.7 allows remote attackers to cause a denial of service (hang) via a large number of requests for nonexistent pages.

Table 3.2: Two examples of the same vulnerability in software substitutes

3.3.2 Exploit Code

In this step of the analysis, we further examine the 29 vulnerabilities that exist in software products providing the same services. If it happens that these software products are used to construct replicas in a replicated system (e.g., a behavioral distance system in Figure 3.1), then both replicas suffer from the same vulnerability. We want to find out whether the exploit codes on them are the same. If they are the same, then both replicas will be compromised by a single attack, and the intrusion detection system will fail to detect the intrusion.

We manage to find all the exploit codes (on multiple products) for 20 out of the 29 vulnerabilities. Exploit codes for the rest do not seem to be readily available to the public. By comparing the exploit codes for each of the 20 vulnerabilities for all the corresponding software substitutes, we found that the exploit code is the same across multiple software products for 14 of the 20 vulnerabilities.

It is not surprising that the same vulnerability will be exploited in the same way, even on different software products. A couple of notes are worth mentioning though. First, some of these vulnerabilities are about denial of service (DoS) attacks, which are usually not the type of intrusions a replicated system utilizing software diversity tries to detect [51, 52]. For example, the same exploit code for sending a large number of requests for non-existent pages will cause a denial of service in the three software products in the second group in Table 3.2. Therefore, this result is not necessarily a strong evidence against the effectiveness of using off-the-shelf software to introduce diversity. Second, we have not studied the effect of using multiple operating systems at this point. In some cases, the exploit codes may be dependent on the operating system, especially in code injection attacks (see the next section).

3.3.3 Summary

To summarize, our analysis of the application software products shows that 22.5% (590 out of 2627) of the vulnerability entries are vulnerabilities in multiple software products, among which 7.1% (29 out of 410) are vulnerabilities in multiple software products that provide the same service. For those vulnerabilities in multiple software products providing the same service, there are roughly 70% (14 out of 20) chances that the same exploit code can be used to compromise these software products. Although strictly speaking these three numbers cannot be multiplied together directly[5], they are very good indications that diverse off-the-shelf application software products can be utilized effectively in replicated systems to detect intrusion and increase system resilience against software attacks.

3.4 Software Products running on Multiple Operating Systems

Having analyzed the branch of vulnerabilities that exist in multiple software products in Figure 3.3 in Section 3.3, we now focus on the branch of vulnerabilities that

[5]This is due to the lack of knowledge about the number of vulnerabilities each software has, the commonality of each software product in terms of the number of requests per unit time, the consequence of a compromise, and etc.

exist in a single software product. As shown in Figure 3.3, this category consists of the majority of vulnerabilities in application software. Therefore, understanding how software products in this category can be utilized to introduce diversity is important. Here we focus on diversity via running software on multiple operating systems, since the vulnerability exists only on a single product and diversity via running software substitutes will definitely work. Running the same software on multiple operating systems is also a cheaper way of introducing diversity due to its lower cost in managing the replicated system.

In this section, we first briefly show the different operating systems we considered (Section 3.4.1), and then examine whether the software products in this category run on multiple operating systems (Section 3.4.2). Finally, similar to our analysis in Section 3.3.2, we analyze the corresponding exploit code in Section 3.4.3.

3.4.1 Different operating systems

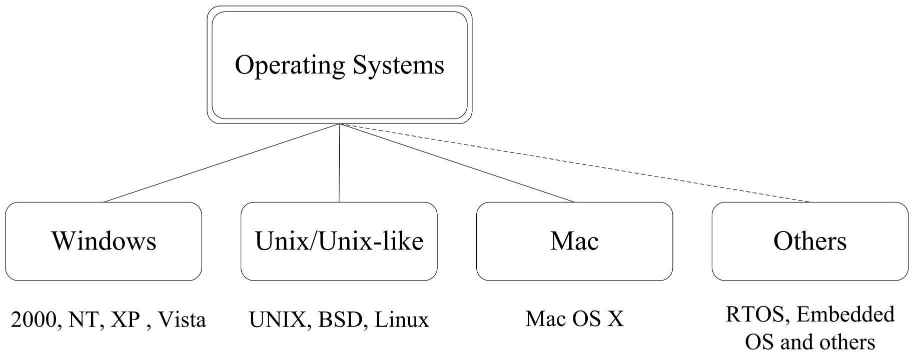


Figure 3.4: Different operating systems

Figure 3.4 shows the different operating systems that we consider in our analysis. We classify operating systems into four families: Microsoft Windows, Unix/Unix-like, Mac and others (see Figure 3.4). This is mainly due to their different kernels and binary executable formats (Portable Executable for Windows systems, ELF for Unix and Unix-like systems, and Mach-O for Mac). Note that it is an important requirement that these operating systems are diverse so that the same exploit is unlikely to compromise the same program running on different operating systems. Although Mac OS X shares part of the kernel code with BSD operating systems, we show in Section 3.5.2 that they rarely share common vulnerabilities.

3.4.2 Software products running on multiple operating systems

Next, we want to find out whether software products in this category (in which vulnerabilities exist only in one software product) can run on multiple operating systems. Since a lot of manual work is required in this analysis, we randomly picked 300 out of the $2,037$ vulnerability entries for analysis. Results are shown in Figure 3.5.

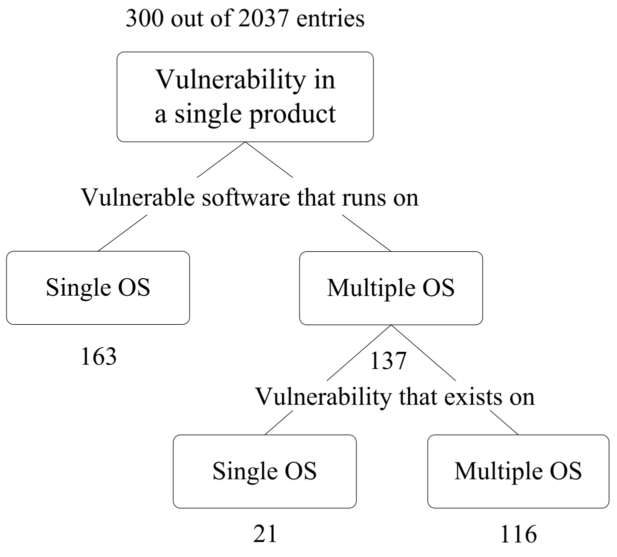

300 out of 2037 entries
Vulnerability in a single product
Vulnerable software that runs on
Single OS
Multiple OS
163
137
Vulnerability that exists on
Single OS
Multiple OS
21
116

Figure 3.5: Vulnerable software on multiple operating systems

Figure 3.5 shows that more than 54% (163 out of 300) of the software products we analyzed officially supports only one operating system. However, note that it is still possible to construct diverse replicas using software substitutes that provide the same service for them.

Among the rest of the 45.7% software products that are supported to run on multiple operating systems, 15.3% (21 out of 137) do not share the same vulnerability among different operating system versions (e.g., the first entry in Table 3.3, in which the vulnerability exists only on the Windows version of Mozilla Firefox, but not on the Unix and Mac versions). From our analysis, this is mainly due to the fact that many of these vulnerabilities are design errors, which easily propagate across versions that run on multiple operating systems. One typical example is the vulnerability entry CVE-2007-5264, in which the client's information is sent unencrypted to the game server (second entry in Table 3.3).

CVE Entry	**Description**
CVE-2007-3285	Mozilla Firefox before 2.0.0.5, when run on Windows, allows remote attackers to bypass file type checks and possibly execute programs via a (1) `file:///` or (2) `resource:URI` with a dangerous extension, followed by a NULL byte (%00) and a safer extension. (Vulnerability in only one of the OS versions of the software product)
CVE-2007-5264	Battlefront Dropteam 1.3.3 and earlier sends the client's online account name and password unencrypted to the game server. A remote attacker with administrative privileges could exploit this vulnerability to obtain user account, product key and other sensitive information. (Vulnerability in multiple OS versions of the software product)

Table 3.3: Vulnerabilities in software products that run on multiple OSes

3.4.3 Exploit Code

Similar to Section 3.3.2, in this subsection we look into the 116 vulnerabilities (each of which exists on multiple OS versions of the single software product), to see whether the same exploit code can be used to compromise the corresponding software program that executes on multiple operating systems.

We first consider a naive attacker, who is not aware that a replicated system where the vulnerable software is being executed on multiple operating systems. We assume that the attacker is trying to exploit a known vulnerability to execute some attack code, e.g., to overflow a buffer and overwrite a return address in order to execute a shellcode. There are at least two reasons why such an exploit is unlikely to succeed.

First, the source of the same software product on different OSes may be different. This could cause many differences in, e.g., memory layout which is critical for a successful buffer overflow. For example, calculating time intervals on Windows usually requires two variables (`SYSTEMTIME` and `FILETIME`) and a conversion between the two, whereas it usually takes only one variable (`timeval`) on Linux.

Second, even when the source is exactly the same for different OS distributions of the same product, the attack code to be executed may be different due to the different APIs and system calls across different operating systems. It is highly unlikely that the same machine code can be used on different operating systems, e.g., to open a shell. The system interface could be different even across OSes in the same family, e.g., different versions of Microsoft Windows. Table 3.4 shows some of the typical system calls and their corresponding system call numbers on different versions of the Windows operating system.

System Call	NT	2000	XP	2003 Server	Vista
`NtClose()`	0x000f	0x0018	0x0019	0x001b	0x002f
`NtOpenFile()`	0x004f	0x0064	0x0074	0x007a	0x00b8
`NtReadVirtualMemory()`	0x0089	0x00a4	0x00ba	0x00c2	0x0102
`NtTerminateProcess()`	0x00bb	0x00e0	0x0101	0x010a	0x014f

Table 3.4: System calls on Windows

Next, we consider a more sophisticated attack in which the attacker is aware that a replicated system running the vulnerable software on multiple operating systems is in use. If the attacker wants to evade the intrusion detection system, he/she will most likely have to design and implement an exploit code that first figures out which operating system is running and subsequently execute the corresponding exploit code (see Algorithm 1).

Algorithm 1 Exploiting the same software running on multiple OSes

```
os_ret ← os_test();
if is_win(os_ret) then
    win_attack_code();
else if is_unix(os_ret) then
    unix_attack_code();
else if is_mac(os_ret) then
    mac_attack_code();
end if
```

Note that Algorithm 1 is very different from one in which the attacker knows the operating system (and its version) to be exploited before sending the attack code. Many attack tools first interact with the vulnerable server to find out which operating system is running by using operating system fingerprinting techniques [49, 104]. After that, the attack packets specifically designed for the corresponding operating system are sent to the vulnerable server. This type of attacks will not work here because 1) the replicated system (e.g., Figure 3.1) usually removes any non-determinism in the system, which makes operating system fingerprinting impossible or inaccurate; 2) the same operating-system-specific attack will be duplicated and sent to all replicas, and the attack only compromises the vulnerable replica (the difference of the behaviors of the compromised and uncompromised replicas makes such operating-system-specific attacks easily detectable).

There are at least two difficulties in implementing Algorithm 1. One is to implement `os_test()` which not only executes on all different operating systems but returns different outputs when executing on different operating systems. The other is that such an exploit code, which is at least several times that of the exploit code for any specific operating system, is usually too long to fit in the limited buffer available in the vulnerable program. We have not found a real attack that employs the technique shown in Algorithm 1.

Another observation is that only three cross-OS viruses have been reported in Kaspersky Lab's viruslist according to the statement issued by Kaspersky Lab[6]. According to Kaspersky Lab, all the three viruses are proof-of-concept malicious programs written purely with the intention of demonstrating that such viruses are possible. None of these viruses actually had any practical applications so far.

3.4.4 Summary

In this section, we analyze the vulnerabilities that exist in a single application software product. Our analysis shows that:

[6] http://www.kaspersky.com/news?id=184875287

- 45.7% (137 out of 300) of the vulnerable software products involved in this category are officially supported on multiple operating systems;
- Among those that are officially supported on multiple operating systems, 84.7% have the vulnerability propagated across multiple OS versions;
- At least two factors (different memory layout and different machine instructions) make it difficult to construct an exploit that can compromise software running on multiple operating systems simultaneously. No such practical attacks have been reported.

These findings show that roughly 50% of the software products are candidates for a replicated system running the same software on multiple operating systems. Even if the same vulnerability exists on multiple replicas, compromising them simultaneously remains difficult. However, due to the fact that most of these vulnerabilities are shared among the different OS versions of the same software, utilizing diverse operating systems is not as effective as utilizing software substitutes.

3.5 Vulnerabilities in Other Software Products

In this section, we present our analysis on the other three categories, namely web script modules, operating systems, language and libraries.

3.5.1 Web script modules

Software in this category consists of light-weighted products that run on web servers to provide web-based applications. Examples include forums, bulletin boards, shopping carts and other script modules. We analyzed the CVE vulnerability database and found close to $3,000$ entries that fall into this category. Some common and well-known types are shown in Table 3.5.

An interesting finding is that most of the vulnerable software in this category is operating system independent. For example, most PHP modules are deployed

Vulnerability Types	Number of entries	Percentage
Cross-site scripting	714	24.7%
SQL injection	669	23.1%
PHP remote file inclusion	634	21.9%
Directory/Path traversal	267	9.2%
Cross-site request forgery	50	1.7%
Others	559	19.3%
Total	2893	100%

Table 3.5: Vulnerabilities in web script modules

on Apache web servers, which can run on all common operating systems. This means that we could use diverse operating systems to introduce software diversity. However, it is different from the application software we analyzed in Section 3.4, since many of the web script modules operate on top of a web server, and seldom interact with the operating system. If the vulnerable software does not interact with the operating system, then constructing replicas using diverse operating systems is not an effective way of introducing diversity because the exploit code is likely to be the same on different replicas. Therefore, we shift our focus of analysis to using software substitutes for introducing diversity.

Cross-site scripting (XSS) vulnerabilities

Cross-site scripting (XSS) is one of the most common web script module vulnerabilities in the CVE database. Attackers exploit this vulnerability by injecting malicious scripts into the output of an application (usually a web page) which is sent to the client's web browser. This script is then executed on the client's web browser and used to transfer sensitive data to a third party (i.e., the attacker) [106]. Unlike other types of web vulnerabilities, XSS vulnerabilities exist and are exploited on the server side but take effects on the client side. Thus, the protection and prevention mechanisms are carried out both on the server side [109] and the client side [106].

In most cases, the server-side scripts are vulnerable no matter what operating systems or web servers on which the scripts run (see an example in Figure 3.6, the attack payload is usually some malicious `HTML/JavaScript`, which is first posted to the server and then downloaded and run at the client side), thus introducing diversity on the server side is not effective. However, introducing diversity on the client side by utilizing diverse browsers is possible. Figure 3.7 shows two examples of XSS attack payload in the exploit code as shown in Figure 3.6.

Description	Cross-site scripting vulnerability in `picture.php` in Advanced Guestbook 2.4.2 allows remote attackers to inject arbitrary web script or HTML via the picture parameter.
Exploit code	`http://www.site.com/picture.php?picture=[attack_payload]`

Figure 3.6: CVE entry CVE-2007-0605 and the corresponding exploit code

[Payload 1] Works for Internet Explorer 6.0 but not Opera 9.0 or Firefox 2.0

```
<IMG SRC=javascript:location.replace('http://evil.
com/steal/index.asp?cookies='+encodeURI(document.
cookie))>
```

[Payload 2] Works for Opera 9.0 but not Internet Explorer 6.0 or Firefox 2.0

```
<IMG SRC=javascript:document.createElement('IMG').
setAttribute('src','http://evil.com/steal/index.
asp?cookies='+encodeURI(document.cookie))>
```

Figure 3.7: XSS attacks that have different impact on browsers

Both XSS attack payloads shown in Figure 3.7 utilize the `HTML` tag `<IMG>` and are used for stealing cookies from client machines that access the vulnerable web site. The exploit codes do not have the same effect on the contemporary browsers because of the implementation difference. The evidences that XSS attack codes have different effects on different browsers can also be found from other resources. For example, 68 out of the 110 XSS attack vectors on the XSS Cheat Sheet (http://ha.ckers.org/xss.html) have different impacts on diverse web browsers. Note that the application scenario here is slightly different from the example shown in Figure 3.1: utilizing diverse browsers to construct the replicated system is a client-side solution instead of the server-side example shown in Figure 3.1. Our results show that by comparing the different impacts on different browsers when given the same input, many XSS attacks could be detected. Analyzing the detection rate of such a system is out of the scope of this chapter.

SQL injection

SQL injection arises when a user input is not correctly or sufficiently filtered. SQL injection attacks are usually launched through specially crafted user inputs on web applications that use strings to construct SQL queries [20]. Although simple SQL statements are constructed exactly the same for different databases, they are different in constructing sophisticated SQL Injection exploits. Consider Blind SQL Injection in CVE-2007-1166, CVE-2007-3051, and many other vulnerable products. The exploit code utilizes the following SQL statements (simplified version).

$$IF\ ((SELECT\ user)\ =\ 'Alice')\ SELECT\ 1\ ELSE\ SELECT\ 1/0$$

After receiving this request, the SQL Server will throw a divide-by-zero error if the current user is not Alice, while the MySQL server will report a *parsing error*.[7] There has also been research on utilizing diverse off-the-shelf databases to obtain fault tolerance [54].

Directory traversal

Directory traversal (or path traversal) vulnerabilities appear when web applications do not sufficiently validate or sanitize the user-supplied file names. It may allow attackers to gain access to directories and files that reside outside of the directory of web documents.

A notable difference in traversing directories on diverse operating systems is that Unix and Unix-like systems use "`../`", while Windows systems use "`..\`". Not only that, the root directory on Windows uses the "`<drive letter>:\`"

[7]Example statement here was tested on SQL Server 2005 and MySQL 5.0. More resources on different syntax for constructing SQL Injection attacks to different databases can be found on SQL Injection Cheat Sheet at http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/

format, which limits directory traversal to a single partition (e.g., C:\). There are other differences, e.g., the file organization also varies a lot on different operating systems.

Remote File Inclusion (RFI)

RFI vulnerabilities allow an attacker to include his own malicious PHP code on a vulnerable web application. RFI attacks are possible because of several PHP configuration flags that are not carefully set. This vulnerability could be avoided easily by disabling two global flags in PHP [39]. Thus, RFI vulnerabilities are not the focus of our study in this work.

Cross-site Request Forgery (CSRF)

By launching a successful CSRF attack to a user, an adversary is able to initiate arbitrary HTTP requests from that user to the vulnerable web application [66]. CSRF attacks are usually executed by causing the victim's web browsers to create hidden HTTP requests to restricted resources. Therefore, similar to XSS vulnerabilities, using diverse browsers is a possible way of detecting CSRF vulnerabilities.

3.5.2 Operating systems, languages and libraries

For operating system vulnerabilities, we try to find out if diverse operating systems have the same vulnerability. We find that Mac OS X has some common vulnerabilities with BSD (e.g., CVE-2007-0229), mainly because the implementation of Mac OS X kernel shares part of the code of BSD kernel [97]. However, these common vulnerabilities only constitute 2% (2 out of 98) of all the vulnerabilities on Mac OS, which indicates that utilizing Unix/Unix-like OS and Mac OS to construct replicas is effective.

Another observation we have is that different Linux operating systems have many common vulnerabilities, since they share the same kernel (e.g. CVE-2007-

3104, CVE-2007-6206 and others). These vulnerabilities contribute 64% (71 out of 111) of all the Linux OS vulnerabilities, which shows that different Linux operating systems are not diverse enough. Finally, by examining all the 438 OS vulnerabilities, no evidence has been found that the same OS vulnerability exists in both Windows and Unix/Unix-like or in both Windows and Mac operating systems.

Many programming languages and libraries (e.g., Java, PHP, Perl, and etc.) support multiple operating systems. However, our analysis in the CVE vulnerability database shows that many of the vulnerabilities in these products are platform dependent. For example, CVE-2007-5862 (a Java vulnerability that exists only in Mac OS X) and CVE-2007-1411 (a PHP buffer overflow vulnerability that allows local, and possibly remote, attackers to execute arbitrary code via several vulnerable PHP functions that exist only in Windows[8]).

3.5.3 Summary

Although in general, software diversity is not very effective in web applications, it is successful in detecting exploits of some web script module vulnerabilities by, for example, utilizing diverse browsers to defend against XSS and CSRF attacks and utilizing diverse databases to detect SQL Injection attacks.

Most OS vulnerabilities only exist in one OS family, which indicates that diversity is useful when utilizing diverse operating systems of different OS families. Although most language and library vulnerabilities are platform independent, there are cases in which they exist in only one particular OS version.

[8]This result is obtained by analyzing NVD/CVE, SecurityFocus and the PHP Buglists. SecurityFocus gives misleading information which indicates that this vulnerability exists on Unix/Unix-like systems (see http://www.securityfocus.com/bid/22893/info). However, the PHP Bug Info (Bug #40746) shows that it is a problem with the function dbopen() in the Microsoft ntdblib library, and does not exist when compiled with FreeTDS version of the dblib library that is used by Unix/Unix-like systems.

3.6 Discussion

In this work, we analyzed the vulnerabilities published in 2007 to evaluate the effectiveness of two ways of introducing software diversity utilizing off-the-shelf software: one is by utilizing different software products that provide the same service, and the other is by utilizing the same software product on different operating systems.

The results show that more than 98.5% of the vulnerable application software products have substitutes and the chance that these software substitutes be compromised by the same attack is very low. Nearly half of the application software products are officially supported to run on multiple operating systems. Although the different OS distributions of the same product have more than 80% of a chance to suffer from the same vulnerability, their attack code is quite different. For the web script modules and other types of software, although software diversity is less effective than that in the application software, some evidence has been found that there are possible ways to benefit from software diversity in these categories.

The limitation of our work mainly includes two parts. The first is that a large amount of manual work has been spent in order to get the accurate statistical results, which is too costly and time consuming. Other information retrieval and artificial intelligence techniques could be applied in our future work to speed up the analysis process. The other limitation is that we have not yet obtained the statistics for some categories due to the large information search space and the lack of closely related resources, which is a challenging task that remains to be done in the future.

Chapter 4

Software Diversity in Intrusion Detection

4.1 Introduction

Host-based anomaly detection techniques based on behaviors of programs in terms of system call sequences were first proposed by Forrest et al. [47], and improved and extended by a number of research work [46, 50, 56, 58, 80, 95]. The normal-behavior models of the applications are learnt from the behaviors observed during a training phase; while during detection, any deviations from the established models are interpreted as attacks to the programs monitored. Later research [23, 72, 86, 100] further enhanced the behavioral model by capturing the information of system call arguments.

Early schemes [72, 86, 100] model the argument behavior at the granularity of different system calls, i.e., each system call (e.g., `open`, `read`, `write`) is assigned with a profile. The granularity is then improved by differentiating the instances of the same system call when their call stacks are different [23]. For example, the legitimate arguments of `open@callstack1` and `open@callstack2` are assigned with different profiles so that they can be tested differently in the detection phase. However, since other context information is not captured during the training, an adversary is able to evade the detection of these existing schemes. Consider the following example code which assumes to contain a buffer overflow vulnerability:

```
int uid = geteuid();
char buf[128];
char* filename;
...
if (uid == 0)
    filename = "/www/admin/configure.ini";
else
    filename = "/www/user/configure.ini";
int fd = open(filename, O_RDWR);
write(fd, buf, sizeof(buf));
```

As illustrated in the example code, the system call `open` accepts two different parameter values in the training phase, both of which correspond to the same call stack. According to the existing schemes [23, 72, 100], both of these strings will be treated as legitimate values during detection. Thus, an attack which overflows `buf` and changes `uid` to 0 will be able to get the administrator privilege while evading detection. Such a situation is more common in modern software applications where code modules are extensively reused. Call stack is not able to tell a difference in the privilege in different executions.

The fundamental difficulty in detecting such attacks stems from the *erratic property* of function arguments. More formally, all legitimate values observed in different normal program executions are not necessarily legitimate at a particular execution. In a particular execution context, only a subset of the values (possibly one) is legitimate while others could potentially be malicious.

This problem seems deceptively simple. The fine-grained context information, which is required to differentiate the legitimate values at run-time, is difficult to gather when training merely one program [23, 72, 100], especially when the source code is not provided. Even for schemes which utilize two diverse applications, their model cannot be simply extended to detect such attacks. For example, hidden Markov models used in [51, 52, 53] (to train the normal-behavior profiles of the system call sequences) are only able to handle finite states, while the space of argument values is usually infinite.

In this work, we propose an intrusion detection scheme which builds on two diverse programs providing semantically-close functionalities. Our model learns the underlying semantic correlation of the argument values in these programs to detect attacks manipulating erratic arguments, which are recognized as normal inputs by existing schemes. Specifically, we make the following contributions:

- We provide a formal approach of detecting attacks utilizing erratic arguments, by learning relations of the function arguments between programs providing semantically-close functionalities.
- We utilize taint analysis to further refine the detection model, which eliminates the coincident relations to decrease the false-positive rates.
- We implement a prototype of our scheme and present a detailed experimental evaluation. The evaluation demonstrates that a number of real attacks which are hard to detect by existing schemes can be effectively detected using our technique. Specifically, it is shown that our detection model not only detects sophisticated attacks on security-critical data, but also detects some Denial-of-Service attacks which are not addressed by existing techniques, with comparable false alarm rates.

4.2 Diversity Detection Model

In this section, we first introduce the framework of our detection approach, which is followed by the definitions of the argument relations. Different algorithms are then provided to train the behavioral model for different types of arguments.

4.2.1 Overview

Figure 4.1 illustrates the basic idea of how our intrusion detection system (IDS) is constructed. We regard two diverse software having *semantically-close functionalities* if they provide same services. Examples of such diverse software could be web servers like Apache and Lighttpd, or office software like Adobe PDF Reader and Foxit PDF Reader. Similar to existing diversity-based intrusion detection techniques, the framework in Figure 4.1 utilizes two diverse software providing semantically-close functionalities to build the behavioral model, base on the observations that these software cannot be successfully exploited by the same attack [61].

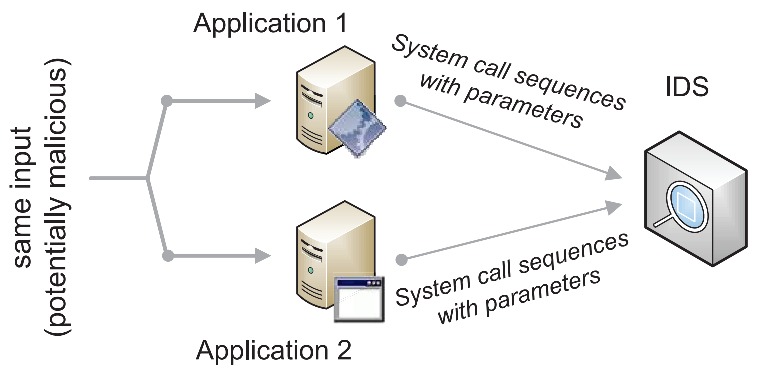

same input
(potentially malicious)
Application 1
Application 2
System call sequences with parameters
System call sequences with parameters
IDS

Figure 4.1: Our diversity IDS framework

In this work, we focus on building a normal-behavior model by extracting the function arguments of both applications. Since these applications provide semantically-close functionalities, there are semantic relations between the behaviors of these applications when they process the same input. **Such semantic relations will exhibit as the relations between the related function calls and their argument values.** For example, two web servers processing the same HTTP request need to access the same local file on the disk. Thus, consequently, there should be functions in both applications whose argument values contain the same file name. In the following, we will briefly introduce how our model captures the argument relations between the two diverse applications. Once the argument relations are trained, they will be utilized to detect attacks that attempt to fool traditional IDS with erratic function arguments.

In the model of Figure 4.1, the same inputs, which are assumed to be free of attacks in the training phase, are passed to both of these applications ($\mathsf{app_1}$, $\mathsf{app_2}$). In order to process the input, each of these applications will invoke a series of system calls (for each input):

$$S_1 = \langle s_{1,1}, s_{1,2}, ..., s_{1,l_1} \rangle \qquad S_2 = \langle s_{2,1}, s_{2,2}, ..., s_{2,l_2} \rangle \tag{4.1}$$

Each system call $s_{i,j}$ has a vector of arguments. In the training phase, all information for each $s_{i,j}$ will be recorded by corresponding monitor module of $\mathsf{app_i}$, and is used to extract the information of the arguments. Specifically, in our model, each argument is identified by:

$$\begin{aligned} \mathsf{arg_{i,x}} \quad & \text{where } i \in \{1, 2\}, \\ & x = \langle \mathsf{index,\ type,\ s\_name,\ callstack} \rangle. \end{aligned}$$

In the above representation, i in $\mathsf{arg_{i,x}}$ indicates this argument appears in the trace of $\mathsf{app_i}$. index is the position of this argument in the corresponding system call, whose name is $\mathsf{s\_name}$; type is the type of the argument (e.g. *string* or *integer*); $\mathsf{callstack}$ stores the call stack information of the corresponding invocation of this particular system call.

In the training phase, we first obtain a pair of system call traces (S_1, S_2) for each input. With all pairs of the system call traces, we then get a set of argument pairs. For each argument pair ($\mathsf{arg_{1,p}}$, $\mathsf{arg_{2,q}}$), $\mathsf{arg_{1,p}}$ is an argument in $\mathsf{app_1}$, which is identified by a unique set of $\langle \mathsf{index,\ type,\ s\_name,\ callstack} \rangle$ appearing in the training set, and $\mathsf{arg_{2,q}}$ is defined similarly. From the training data, we collect a set of value pairs $\mathsf{Value_{p,q}}$ for each argument pair, where $\mathsf{Value_{p,q}} = \{(\mathsf{v_1}, \mathsf{v_2}) | \mathsf{arg_{1,p}} = \mathsf{v_1}, \mathsf{arg_{2,q}} = \mathsf{v_2}\}$. According to $\mathsf{Value_{p,q}}$, we then produce a database of relations $\mathbb{R} = \{\langle \mathsf{arg_{1,p}}\ \mathsf{R}\ \mathsf{arg_{2,q}} \rangle\}$. This relation set $\mathbb{R}$ is finally utilized to detect whether there is any violation for each pair of parameter values. If the relation of a pair of parameter instances ($\langle \mathsf{arg}_{1,p} = \mathsf{v}_x \rangle$ and $\langle \mathsf{arg}_{2,q} = \mathsf{v}_y \rangle$) does not satisfy the corresponding $\langle \mathsf{arg}_{1,p}\ \mathsf{R}\ \mathsf{arg}_{2,q} \rangle$ in $\mathbb{R}$, the IDS will raise an alarm.

4.2.2 Relationships of the Arguments

In our model, we focus on two most common types of system call arguments – *string* and *integer*, the definitions of which follow the standard definition in programming language: a *string* is a sequence of zero or more characters followed by a `NULL` ("\0") character; while an *integer* is a numeric variable holding whole numbers.

We define binary relation R that captures the relationship between two system call arguments in the diverse applications. The relation between two arguments is expressed as $\langle \mathsf{arg}_1\ \mathsf{R}\ \mathsf{arg}_2 \rangle$, where arg_1 is a particular argument in the first application, and arg_2 is a particular argument in the second application. Different sets of candidate relations are given to *string* and *integer* since these two argument types have different characteristics.

We provide the following basic relations for *string* arguments:

- **equal** captures equality relation of the given two arguments, e.g., the file name passed to an `open` system call in app_1 could be the same as the file name passed to another `open` (or `stat64`) system call in app_2.
- **samePrefix(n)** indicates that the two string arguments have the same prefix, the length of which is at least n. For example, if arg_1 = `"/home/usr/xyz"` and arg_2 = `"/home/usr/abc"`, then $\langle \mathsf{arg}_1\ \mathsf{samePrefix(10)}\ \mathsf{arg}_2 \rangle$ holds.
- **sameSuffix(n)** indicates that the two string arguments have the same suffix substring with length at least n.
- **contain** means that the second argument is a substring of the first argument.
- **partOf** is the reverse of **contain** relation, in which the first argument is a substring of the second argument.

Note that for the same pair of arguments, more than one of the above relations may hold. For example, if arg_1 = `"/home/configure.ini"` and arg_2 = `"/home/conf.ini"`, then both $\langle \mathsf{arg}_1$ samePrefix(10) $\mathsf{arg}_2\rangle$ and $\langle \mathsf{arg}_1$ sameSuffix(4) $\mathsf{arg}_2\rangle$ hold. The above five relations defined are sufficient to cover the binary relations of string arguments proposed in existing approaches, which are defined for modeling the binary relations of arguments in a single program, such as isWithinDir, hasSameDirAs, hasSameExtensionAs [23].

For *integer* arguments, we use a polynomial equation to represent the relation of the two arguments. That is, let x = arg_1 and y = arg_2 (or x = arg_2 and y = arg_1), the following equation holds:

$$y = c_m x^m + c_{m-1} x^{m-1} + ... + c_1 x + c_0 \tag{4.2}$$

For example, for the two `malloc` calls which create a memory region to store the `uri` string parsed from the same request, the parameter values of these two `malloc` could have the form $y = 1 \cdot x + c_0$. The value of c_0 may not be 0 because the internal structures which store the `uri` are different in these two programs. Note that in Equation 4.2, when $c_1 = 1$ and $\forall i \neq 1, c_i = 0$, then arg_1 = arg_2. In our model, this equal relation between numeric arguments is able to capture most relations of flag arguments (such as `O_RDONLY` and `O_RDWR`), because they usually appear as the same in the diverse software providing semantically-close functionalities.

Polynomial relation does not cover all the binary relations between two integer arguments, e.g., exponential relation or bitwise relation may also exist under some circumstances. In our current model, we only preserve polynomial relation for integer parameters as it is the most common relation we observed in real applications.

4.2.3 Training Algorithms

The training procedure can be generally divided into three stages: argument pair extraction, relation acquisition and relation refinement.

Argument pair extraction

In this first stage, our purpose is to extract a set of $\mathsf{Value_{p,q}}$ for each pair of $\langle \mathsf{arg_{1,p}}, \mathsf{arg_{2,q}} \rangle$. Each $\mathsf{Value_{p,q}}$ set will contain all the value pairs occurred in the whole training procedure. All the sets of $\mathsf{Value_{p,q}}$ will then be used to train the relation R between $\langle \mathsf{arg_{1,p}}, \mathsf{arg_{2,q}} \rangle$. The algorithm of extracting each pair of arguments and its corresponding values are given in Algorithm 2, after which a set $\mathbb{PV}$= $\{(\mathsf{arg_{1,p}}, \mathsf{arg_{2,q}}, \mathsf{Value_{p,q}})\}$ will be collected. This $\mathbb{PV}$ set will then be used as input in Algorithm 3 and Algorithm 4.

Algorithm 2 Argument-pair extraction

```
 1: for each ($S_1$, $S_2$) pair in the training set do
 2:     for each $s_{1,j}$ in $S_1$ and each $s_{2,k}$ in $S_2$ do
 3:         if comparable($s_{1,j}$, $s_{2,k}$) then
 4:             for each $\mathsf{arg_{1,p}}$ belonging to $s_{1,j}$, and each $\mathsf{arg_{2,q}}$ belonging to $s_{2,k}$ do
 5:                 $\mathsf{v_1}$ = value of $\mathsf{arg_{1,p}}$
 6:                 $\mathsf{v_2}$ = value of $\mathsf{arg_{2,q}}$
 7:                 if ($\mathsf{arg_{1,p}.type} = \mathsf{arg_{2,q}.type}$) then
 8:                     if ($\mathsf{arg_{1,p}}, \mathsf{arg_{2,q}}, \mathsf{Value_{p,q}}$) already exists in $\mathbb{PV}$ then
 9:                         add ($\mathsf{v_1}$, $\mathsf{v_2}$) to $\mathsf{Value_{p,q}}$ if ($\mathsf{v_1}$, $\mathsf{v_2}$) $\notin \mathsf{Value_{p,q}}$
10:                     else
11:                         $\mathsf{Value_{p,q}} = \{(\mathsf{v_1}, \mathsf{v_2})\}$
12:                         add ($\mathsf{arg_{1,p}}, \mathsf{arg_{2,q}}, \mathsf{Value_{p,q}}$) to $\mathbb{PV}$
13:                     end if
14:                 end if
15:             end for
16:         end if
17:     end for
18: end for
```

This step is critical to the rest of the training procedure. The amount of all the combinations of $\langle \mathsf{arg_{1,p}}, \mathsf{arg_{2,q}} \rangle$ could be huge, however, we only consider argument pairs which appear in `comparable` function calls (as shown in line 3 of Algorithm 2). We define *comparable* function calls as those functions which have the same function names or whose functionalities are semantically related. For example, system calls `open` and `stat64` are comparable, and library calls `malloc`, `calloc` and `realloc` are comparable. System calls like `setuid` and `open` are not comparable since their functionalities are not semantically related. Our current implementation of Algorithm 2 reads in a configuration file that specifies which function calls are comparable. This configuration file is carefully constructed according to the platform on which the target applications are running. Our current implementation only considers the Linux operating systems with GNU C library.

Relation acquisition

The next step is to learn the relations between each pair of arguments gained by Algorithm 2. Here we introduce two algorithms for learning the relations: Algorithm 3 is used to learn the relations between two *string* arguments; while Algorithm 4 is for *integer* arguments. We use $\emptyset$ to denote that there is no relation between two arguments ($\mathsf{arg}_1\ \emptyset\ \mathsf{arg}_2$).

Algorithm 3 String-relation learning

```
Require: set $\mathbb{PV}$.
 1: for each ($\mathsf{arg}_{1,p}, \mathsf{arg}_{2,q}, \mathsf{Value}_{p,q}$) in $\mathbb{PV}$ do
 2:   if $\mathsf{arg}_{1,p}$.type = $\mathsf{arg}_{2,q}$.type = string then
 3:     for each ($\mathsf{v}_1$, $\mathsf{v}_2$) in $\mathsf{Value}_{p,q}$ do
 4:       calculate R $\in${equal, samePrefix(n), sameSuffix(n), contain, partOf, $\emptyset$},
          which satisfies $\mathsf{v}_1$ R $\mathsf{v}_2$.
 5:       if R $\neq \emptyset$ then
 6:         for each $\mathsf{R}_c$ that $\langle \mathsf{arg}_{1,p}, \mathsf{R}_c, \mathsf{arg}_{2,q} \rangle \in \mathbb{R}$ do
 7:           if R conflicts with $\mathsf{R}_c$ then
 8:             remove all $\langle \mathsf{arg}_{1,p}, \mathsf{R}_c, \mathsf{arg}_{2,q} \rangle$ in $\mathbb{R}$
 9:             add $\langle \mathsf{arg}_{1,p}, \emptyset, \mathsf{arg}_{2,q} \rangle$ to $\mathbb{R}$
10:           else
11:             add $\langle \mathsf{arg}_{1,p}, \mathsf{R}, \mathsf{arg}_{2,q} \rangle$ to $\mathbb{R}$
12:           end if
13:         end for
14:       else if $\langle \mathsf{arg}_{1,p}, \emptyset, \mathsf{arg}_{2,q} \rangle \notin \mathbb{R}$ then
15:         add $\langle \mathsf{arg}_{1,p}, \emptyset, \mathsf{arg}_{2,q} \rangle$ to $\mathbb{R}$
16:       end if
17:     end for
18:   end if
19: end for
```

Note that there is an update procedure in the learning process of Algorithm 3 for the relation of samePrefix(n) and sameSuffix(n), which is not shown in the algorithm. Take $\mathsf{samePrefix(n)}$ for example, suppose the existing relation for $\mathsf{arg_1, arg_2}$ in $\mathbb{R}$ is $\mathsf{samePrefix(n_{old})}$ and the new learnt relation is $\mathsf{samePrefix(n_{new})}$. The new relation of $\mathsf{arg_1, arg_2}$ in $\mathbb{R}$ will be updated as $\mathsf{samePrefix(min(n_{old}, n_{new}))}$.

Another important detail not shown in Algorithm 3 is that, a threshold N can be set for the relations $\mathsf{samePrefix(n)}$ and $\mathsf{sameSuffix(n)}$, to reduce the false positives caused by small n. During learning, if the calculated $n < N$, then set $\mathsf{R} = \emptyset$. And different N should be assigned for $\mathsf{samePrefix(n)}$ and $\mathsf{sameSuffix(n)}$. Also note that a set of confliction rules for the relations is needed in Algorithm 3 (at line 7). Generally, $\emptyset$ conflicts with other relations, and equal, $\mathsf{contain}$, partOf conflict with each other since the equal relation will always be verified first.

In Algorithm 4, the given order m should be at least 2, and should not be too large so as to avoid the overfitting problem. m can also be dynamically adjusted according to the size of each $\mathsf{Value_{p,q}}$. However, the value of m should be at most $\mathsf{Value_{p,q}.size} - 1$ in order to have enough value pairs for solving the equation set and leave at least one value pair to verify the results.

The whole learning process is optimized by utilizing the $\emptyset$ relations. The $\mathbb{PV}$ set does not need to be fully computed before running Algorithm 3 and Algorithm 4. If $\langle \mathsf{arg_{1,p}}\ \emptyset\ \mathsf{arg_{2,q}} \rangle$ already appears in $\mathbb{R}$, then the remaining instances of $\langle \mathsf{arg_{1,p}}, \mathsf{arg_{2,q}} \rangle$ do not need to be added into $\mathbb{PV}$. The $\emptyset$ relations will be dropped at the end of the training.

4.2.4 Model Refinement

In this subsection, we include an additional training phase to refine the relations we have obtained by the above algorithms. The relations $\mathbb{R}$ gained by using previous algorithms are patterns on the values we observed. However, certain trained relations may be due to the coincidence in the training data set, which could cause false alarms in detection. Thus, it will be better if we can remove those trained patterns in $\mathbb{R}$ which are not caused by the semantic relations between the two diverse

Algorithm 4 Integer-relation learning

```
Require: set $\mathbb{PV}$, order $m$.
1: for each ($\mathsf{arg}_{1,p}, \mathsf{arg}_{2,q}, \mathsf{Value}_{p,q}$) in $\mathbb{PV}$ do
2:     if $\mathsf{arg}_{1,p}$.type = $\mathsf{arg}_{2,q}$.type = integer then
3:         if $\mathsf{Value}_{p,q}$.size $< m$ then
4:             add $\langle \mathsf{arg}_{1,p}, \emptyset, \mathsf{arg}_{2,q} \rangle$ to $\mathbb{R}$
5:         else
6:             use the first $m$ pairs of ($\mathsf{v}_1$, $\mathsf{v}_2$) in $\mathsf{Value}_{p,q}$ to solve the equation set of
               Equation (4.2) to get $(c_m, ..., c_0)$, for both ($x = \mathsf{arg}_{1,p}, y = \mathsf{arg}_{2,q}$) and
               ($x = \mathsf{arg}_{2,q}, y = \mathsf{arg}_{1,p}$).
7:             if the equation set is solvable then
8:                 $\mathsf{R} = \{x, y, (c_m, ..., c_0)\}$
9:                 for each ($\mathsf{v}_1$, $\mathsf{v}_2$) left in $\mathsf{Value}_{p,q}$ do
10:                    if Equation (4.2) does not hold then
11:                        $\mathsf{R} = \emptyset$
12:                    end if
13:                end for
14:                add $\langle \mathsf{arg}_{1,p}, \mathsf{R}, \mathsf{arg}_{2,q} \rangle$ to $\mathbb{R}$
15:            else
16:                add $\langle \mathsf{arg}_{1,p}, \emptyset, \mathsf{arg}_{2,q} \rangle$ to $\mathbb{R}$
17:            end if
18:        end if
19:    end if
20: end for
```

applications.

However, it is not an easy task to validate the semantic relations of arguments and refine the trained model. Even with the source code, it is difficult for a human to capture the exact semantic meaning of a given function in a complex application. Thus, to automatically capture the semantic meanings of functions without the source code is an even harder problem. One way of learning the semantic relations between arguments is to use taint analysis [94]. Since the semantics of different set of function calls vary a lot, the detailed method of carrying out taint analysis needs to be customized accordingly. It is difficult to design a universal solution to perform the taint analysis for all the function calls.

In our current work, we develop a method of mapping memory management library calls (such as `malloc`, `free`, `realloc`, etc.) of two diverse web servers, according to the semantics gained by taint analysis. The basic idea is as follows: First of all, by tainting the request stream sent from client, we gain the knowledge that which portions of the request are mapped to which heap memory regions. Since these memory regions are created by the corresponding memory library calls, each library call can be correlated with a certain portion of the request. We mapped the two memory library calls (e.g., one `malloc` in Apache and one `calloc` in Lighttpd) whose memory regions store the same part of the request (e.g. the `uri`). We then preserve the argument relations that belong to the mapped library calls, and remove other unmapped relations from $\mathbb{R}$. The implementation detail is given in Section 4.3, the effect of such refinement will be further evaluated in Section 4.4.

4.2.5 Detection

After the relation set $\mathbb{R}$ is trained, the detection phase is quite straightforward. During detection, for each argument pair $(\mathsf{arg}_{1,p}, \mathsf{arg}_{2,q})$ that appears in $\mathbb{R}$, each instance of $(\mathsf{arg}_{1,p} = \mathsf{v}_x, \mathsf{arg}_{2,q} = \mathsf{v}_y)$ will be tested. If an instance does not satisfy the corresponding $\langle \mathsf{arg}_{1,p}\ \mathsf{R}\ \mathsf{arg}_{2,q} \rangle$ in $\mathbb{R}$, the IDS will raise an alarm. Although the complexity of the training is relatively high, the detection only involves simple and fast computation. The main cost of detection depends on the cost of monitoring and logging the function calls.

4.3 Implementation

We have implemented our approach on Ubuntu 8.04 (Linux kernel 2.6.24). The implementation consists of two online components and an offline component.

The two online components are both monitor modules (referred to as tracer), one of which is used to trace system calls, the other is used to trace library calls of the monitored programs. For the system call tracer, we utilize `ptrace` to intercept each system call made by the monitored program and log the following information: (a) the PC value from where the system call was invoked, (b) values of arguments, and (c) the call stack information which contains a set of absolute return addresses. For the library call tracer, we modify the GNU C library (glibc) under Ubuntu to output similar information for a selected set of library calls. Since the `backtrace` method cannot be used within the implementation of some library calls such as `malloc`, we implement our own `backtrace` method in the glibc to log the call stack information.

Each time when the monitored program starts, all the base addresses of its loaded shared libraries are also recorded, which is retrieved from corresponding `/proc/[pid]/maps`. These addresses will be used to convert the absolute addresses in the call stack recorded by the tracer to relative addresses, in the form of `[libname+offset]`. By having relative call stacks, we are able to identify the same instance of function call across different runs of the same program.

The offline component of our implementation includes the parsers of the logged traces and the training module that implements the algorithms in Section 4.2. As mentioned earlier, a configuration file is also provided to the training module, which specifies the function calls that are comparable. The implementation of the offline component is about 3.5K LOC.

For the model refinement part in the training, we utilize TEMU [15] to carry out the taint analysis. Web server programs running in TEMU are provided with tainted request stream and tainted local disk files, and the instructions of the monitored web server will be recorded when processing each request. The recorded instruction traces are then translated by the `trace_reader` tool in Vine [15] and used as inputs to the trace parsers we implemented. According to the taint information in the trace files, our trace parser will be able to extract the information that each memory library calls is related to which part of the request stream (or is related to which file on the disk). Then two library calls (in two diverse servers) which are related to the same part of the request (or the same local file) are recorded as the mapped library calls as mentioned in the previous section. This TEMU trace parser is around 1K LOC.

4.4 Evaluation

In this section, we first investigate the effectiveness of our approach in detecting real attacks and then analyze the false alarm rates. Performance overheads for intrusion detection are also discussed. All experiments are conducted under Ubuntu 8.04 and the training and testing are performed in offline mode.

4.4.1 Detection Effectiveness

Since the code injection attacks have been extensively addressed in prior research [46, 47, 50, 56, 58, 80, 95], we focus on evaluating the detection effectiveness of our model against attacks on security-critical data utilizing erratic arguments. Table 4.1 lists the set of attacks tested in our evaluation. The first two attacks in Table 4.1 are detectable by our approach since they both violate the *string* argument relations trained in our model, while the other two attacks in Table 4.1 violate the *integer* argument relations.

Reference	Vulnerable Program	Attack Description	Alternative Program	Type
S.Chen et al. [30]	Ghttpd	stack overflow to overwrite filename data	Null-httpd	String
S.Chen et al. [30]	Null-httpd	heap overflow to corrupt cgi-bin configuration string	Ghttpd	String
S.Chen et al. [30]	Wu-ftpd	format string attack to overwrite userid data	Pure-ftpd	Integer
CVE-2008-4298	Lighttpd	memory leak via duplicate request headers	Cherokee	Integer

Table 4.1: Selected non-control-flow attacks

Detection of anomalous string arguments

The first attack in Table 4.1 exploits a stack overflow vulnerability in Ghttpd's logging function [30], which occurs in the following code fragment in function `serverconnection()`:

```
1: if (strstr(ptr, "/.."))
2:   reject the request;
3: log(...);
4: if (strstr(ptr, "cgi-bin"))
5:   execve(ptr, ...)
```

In the above code, `ptr` is a char pointer to the string of URL requested by a remote client. The first two lines in the code are used to check the absence of "`/..`" in the URL, before the CGI request is parsed and handled in line 4–5. The stack buffer overflow vulnerability is in function `log()`, where a long user input string can overrun a 200-byte stack buffer. Chen et al. [30] managed to construct a stealthy attack which changes `ptr` to point to a string `cgi-bin/../../../../bin/sh` by exploiting the vulnerability in `log()`. Their attack neither injects code nor alters the return address, thus, it is difficult to be detected by most of the existing models.

Our approach is able to detect this attack. During training, our model learns the equal relation between the first parameter of `execve` in Ghttpd and the parameter of corresponding `execve` in Null-httpd (in function `cgi_main()`). Since this relation is later violated when this attack has successfully changed the value of `ptr` in Ghttpd, an alarm is raised by the IDS.

Although this attack is also detectable by the dataflow model [23], their mechanism is different. Their system first learns that all files executed at line 5 should be within the `"cgi-bin"` directory. The attack is detected when it accesses a file outside this directory. However, such `isWithinDir` [23] relation (trained by monitoring the program itself) may not be sufficient in practical scenarios. For example, in typical business applications, files under the same directory may have different access policies. A user x is only allowed to execute program A under the directory, but not program B. Due to the overflow attack, adversary with the privilege of user x is able to gain the access to program B. Under such a scenario, the `isWithinDir` relation will not be able to detect such attacks since all the programs are under the same directory, while our model is still able to detect attacks in cases like these.

The second attack in Table 4.1 targets on a heap overflow vulnerability exists in Null-httpd. This vulnerability is triggered when a special POST command is received by the server. This vulnerability can be used to corrupt the CGI-BIN configuration of Null-httpd and will result in root compromise without executing any external code. In the attack illustrated by Chen et al. [30], two POST commands are issued to precisely overwrite four characters in the CGI-BIN configuration so that it is changed from `"/usr/local/httpd/cgi-bin\0"` to `"/bin\0"`. After the corruption, `/bin/sh` can be started as a CGI program and any shell command can be sent as the standard input to `/bin/sh`.

This attack cannot be easily detected by control-flow schemes [46, 47, 50, 56, 58, 80, 95], and is not addressed by the dataflow scheme [23]. However, our diversity model is able to detect such an intrusion due to the same reason in the first attack – the equal relation (of the first parameter of the two `execve` calls in Null-httpd and in Ghttpd learnt during training) is violated when Null-httpd is exploited.

Note that although both of these two servers (Ghttpd and Null-httpd) have vulnerabilities, we can still use them together to build our diversity detection model because their vulnerabilities are not exploitable by the same attack code. In general, the probability that the same vulnerability exists in two diverse applications providing semantically-close functionalities is very low [61].

Detection of anomalous integer arguments

The third attack in Table 4.1 exploits a format string vulnerability in Wu-ftpd. The vulnerable code fragment is within the `getdatasock()` function:

```
1: seteuid(0);
2: setsockopt( ... );
   ...
3: seteuid(pw->pw_uid);
```

The above function is invoked when a user issues data transfer commands, such as downloading or uploading a file. It requires root privilege in order to perform the setsockopt() operation. Thus, the privilege is temporarily escalated using seteuid(0) and then changed back by the second seteuid(). The data structure pw->pw_uid is a cached copy of the user ID saved on the heap. The attack proposed in [30] exploits the format-string vulnerability to change pw->pw_uid to 0, which maintains the root privilege for the attacker so that arbitrary files can be uploaded and downloaded by the attacker as a root user.

Our model detects this attack when monitoring Wu-ftpd together with Pure-ftpd. Since the two servers have the same configurations, the parameter of seteuid()[1] function call on line 3 in Wu-ftpd always has the same value as the parameter of the seteuid() calls in function doport3() in Pure-ftpd. These *integer* parameter relations are violated when the adversary overflow the heap to change pw->pw_uid to 0.

The fourth attack in Table 4.1 exploits a memory leak vulnerability exists in Lighttpd. When a duplicated field appears in a request header (e.g., "User-Agent :Mozilla/4.0" and "User-Agent:MSIE/8.0" both appear in the header), the http_request_parse() method in Lighttpd will allocate a memory region to store the content of the second field (i.e., MSIE/8.0), but will not recycle this resource afterwards. An adversary can utilize this vulnerability to consume the memory of the server running Lighttpd by sending many requests with duplicate fields (with a maximum field length of 2KB).

Such Denial-of-Service attack cannot be directly detected by the existing approaches trained on a single server, especially when the total memory consumed is not large enough to cause any exception. The difficulty comes from the memory management behaviors of these web servers. For the most commonly used servers (such as Apache, Lighttpd, etc.), the allocated memory will be reused in processing the following requests and never be explicitly freed. Thus, for both normal request and attack request processing, only memory allocation methods (such as malloc, realloc ...) are observed, no deallocation method (such as free) will appear

[1]The underlying system call invoked is setresuid32().

in the library call sequences obtained. This makes it difficult for an IDS to precisely model the memory behaviors, as it requires simulating the complex internal memory management of these server applications.

Our diversity IDS is able to learn the integer argument relations of the corresponding memory allocation calls in the two servers monitored. To be specific, the IDS learns that 16 pairs of the parameter values to the `malloc` and `realloc` calls of Lighttpd and Cherokee servers are equal or have fixed difference (which is actually due to the size difference of the internal structures in these two servers). In the detection phase, the IDS detects the memory leak attack immediately when the attack request causes one of Lighttpd's `malloc` parameter to increase (in `buffer_copy_string_len()` invoked by `http_request_parse()`), which violates the integer relations that have been trained in the model.

4.4.2 False Alarm Analysis

There are three pairs of programs in Table 4.1. All of them are used to evaluate the false alarm rates of our approach, as shown in Table 4.2. Two pairs of them are http servers (Lighttpd and Cherokee, Ghttpd and Null-httpd), which are configured to hold the same content of the web site of our university. In the training phase, the two web servers in the same pair are provided with the same series of requests (10K requests) obtained from the real log of our university's web server. In the detection phase, another set of requests (50K requests) from the logs are sent to these servers to evaluate the false alarm rates. Applications in the third pair are FTP server programs (Wu-ftpd and Pure-ftpd). Since we do not have the access to the log of large amount of real FTP requests, we configure these two FTP servers to hold the files downloaded from GNU FTP[2], and simulate the requests by randomly issuing commands (such as `put`, `get`, `dir`, `passive`, `type`, etc.) for random files or directories on the servers.

[2]GNU Software FTP server, ftp.gnu.org/gnu .

Diverse Programs		Training Trace # of **Sys calls** ($\times 10^5$)	Detection Trace # of **Sys calls** ($\times 10^5$)	False alarms ($\times 10^{-5}$)
Pair 1	Lighttpd	2.29	10.90	0.826
	Cherokee httpd	5.19	24.35	
Pair 2	Ghttpd	7.24	39.51	1.948
	Null httpd	20.62	98.57	
Pair 3	Wu ftpd	10.78	54.15	0.617
	Pure ftpd	4.37	12.96	

Table 4.2: False alarm rate

We construct two different experiments to test our false alarm rates (as shown in Table 4.2 and Table 4.3). The first experiment only focuses on monitoring the system calls and their arguments so that it can be compared with existing approaches which also utilize system call arguments [23, 72] (e.g., the result of the dataflow model [23] shows the false-positive rate of the tested HTTP server is 64.12×10^{-5}, and the rate for SSH server is 0.02×10^{-5}). Note that the rates shown in Table 4.2 are "raw" false alarm rates, i.e., the fraction of system calls that caused violations, without combining the same type of violations. For example, the false alarm rate for Lighttpd in Table 4.2 is 0.826×10^{-5}, which means that one false alarm will be raised for every 100K system calls processed. This indicates that one out of 10K requests will cause false alarms, as on average 10.9 system calls are invoked to process one request for Lighttpd.

Programs	Training Trace # of **Lib calls** ($\times 10^5$)	Detection Trace # of **Lib calls** ($\times 10^5$)	False alarms ($\times 10^{-5}$)	
			Original	After Refine
Lighttpd	2.31	11.06	5.286	1.762
Cherokee	0.46	2.27		

Table 4.3: Model refinement by taint analysis

The results show that the second pair of applications have much higher false alarm rate than the other two pairs, as in Table 4.2. We investigated the reason for this higher false alarm rate, and found that this is due to the fact that during the training, there are several coincident contain relations for the string arguments between Ghttpd and Null-httpd, which are violated in the detection phase for benign requests. Our current implementation of the training algorithm regards two string arguments as contain as long as their values satisfy this relation, even if these pair of arguments only appear once in the training. However, some rules in the training phase could be added to further decrease the false alarm rate. For example, any string relations should have at least two instances of value pairs in the training phase so that one instance of values is used to set up the relation and other values can be used to validate the relation in the training (and any argument pairs which only have one instance should be regarded as $\emptyset$ relation in $\mathbb{R}$). Such modification could reduce the false positives of our model but should be carefully designed so that it would not decrease the detection capability as well. Investigation on this trade-off is left as future work.

In the second experiment (as shown in Table 4.3), we investigate the false-positive rate when our model monitors the memory management library calls of the diverse applications. Note that different from Table 4.2, only library calls are considered in Table 4.3. We further investigate the effectiveness on false positive reduction by refining our model using taint analysis. The result shows that after removing the library call argument patterns which are not mapped by the semantic relations, the false-positive rate decreases. It is possible to refine the relations of other arguments by using taint analysis. However, since the semantics of different set of library/system call arguments vary, taint analysis needs to be carefully customized accordingly.

4.4.3 Performance Overheads

Table 4.4 shows the size of the programs used in our evaluation, along with the model sizes in terms of the number of relations learnt. Note that the sizes of the programs in the first pair include some of their own shared libraries. This is because part of the functionalities of these servers are compiled as shared libraries in default (e.g., many of the commonly used functions in cherokee are compiled in `libcherokee-base.so` and `libcherokee-server.so`), which is different from standalone programs. It can be seen from the table that the size of our models are relatively smaller compared to the sizes of the programs.

Programs		Program Size (KB)	String Relations	Integer Relations
Pair 1	Lighttpd	767.9	143	367
	Cherokee httpd	1165.7		
Pair 2	Ghttpd	43.6	120	342
	Null httpd	34.3		
Pair 3	Wu ftpd	385.3	171	496
	Pure ftpd	87.8		

Table 4.4: Program size and model size

We also studied the time cost of our model for both learning and detection phases, which is illustrated in Table 4.5. The original size of the training traces were between 110MB and 526MB, consisting of 0.2 to 2 million system calls. As shown in Table 4.5, we measure the performance overheads of monitoring the system calls and library calls, which are the dominate overheads during detection. It shows that the overhead of monitoring system calls could be quite high for web servers (up to 83.4%). The overhead is mainly due to our system call tracer. As explained in Section 4.3, our monitor module utilizes `ptrace` for system call interception with our own implementation of the `backtrace` which records the call stack information of each system call. Similar overhead was also reported by existing approach [23] using `ptrace`. This cost can be reduced to less than 6% [50], by a kernel implementation of the interceptor.

Programs	Training time	Detection Overheads	
		Monitoring sys calls	Monitoring lib calls
Lighttpd & Cherokee	93.8 sec	29.10%	18.38%
Ghttpd & Null-httpd	1620.9 sec	83.39%	11.41%
Wu-ftpd & Pure-ftpd	2091.3 sec	17.56%	1.37%

Table 4.5: Training time and detection overhead

4.5 Discussion

Traditional intrusion detection techniques [32, 46, 47, 50, 56, 58, 80, 95, 107] mainly focus on utilizing only system call sequences to detect code injection attacks. Recent works [23, 72, 78, 86, 100] further incorporate system call argument information to defend against attacks which do not modify control flows. However, these approaches have difficulties in deciding which legitimate argument value is really benign, when multiple legitimate values appear in the training phase.

Early works on software diversity construct intrusion tolerance systems [29, 88] with software providing semantically-close functionalities. This architecture is then utilized for developing diversity-based intrusion detection techniques [34, 51, 52, 68, 102]. Most of these techniques use Commercial Off-The-Shelf (COTS) software to build the detection models. Among those schemes, the techniques proposed by Just et al. [68] and Totel et al. [102] are output voting schemes, which only compare the final outputs (HTTP status codes and files) of the diverse software to detect intrusions. However, as many of the intrusions may not result in observable deviation in the responses of those server software, such intrusions can evade detections of these techniques.

Behavioral Distance model by Gao et al. [51, 52] was later proposed to defend against stealthy attacks which are not addressed by both the output voting schemes and traditional intrusion detection techniques which only monitor single application. However, since hidden Markov model used in their scheme (to train the normal-behavior profiles of the system call sequences) is only able to handle finite states, their model cannot be simply extended to detect attacks utilizing erratic arguments.

Our approach proposed in this chapter is the first work that captures the underlying semantic correlation of the argument values in diverse programs. Our model gains more accurate context information compared to existing schemes. Such context information is critical in detecting sophisticated attacks on security-critical data utilizing erratic arguments. When deployed, our model can be combined with the existing system call sequence or control flow models to defend against a wider range of attacks. The main limitation of our scheme is the additional cost on the management of diverse software. However, such a cost could be negligible for some existing fault-tolerant system where diverse software have already been deployed to prevent simultaneous failure.

Chapter 5

Application Security Comparison of Diverse Mobile Platforms

5.1 Introduction

The current intensive competition among mobile platforms sparks heated debate over the question which platform has a better architecture for security and privacy protection. Among these platforms, Google's Android and Apple's iOS are the top players in terms of user base, which are often compared to each other [7, 8, 16, 17]. Some articles [8, 17] claimed Android is better due to 1) the complete permission list visible to the user and 2) the open-source nature of Android. In the meantime, some other media [7, 16] argued that iOS is actually better for the following reasons: 1) Apple screens applications before they let them appear in the iTunes Store (also known as Apple's *vetting* process); 2) Apple controls their hardware completely so that OS patches and security fixes are issued immediately on all devices; 3) The open source nature of Android also brings advantages for malicious software developers. There are also other reports [9, 14] that conclude these two platforms achieve comparable security but in different ways. These different voices raise a question on how we can provide a reliable baseline for security comparison among the security architecture of different mobile platforms. Obviously, the prior efforts [7, 8, 16, 17, 9, 14] in only comparing abstract and general practices are not sufficient. We need to go deeper to investigate how the platform difference influences the root of their security, the applications.

In this work, we make the first attempt to establish a reliable baseline for comparing security of mobile platforms by investigating permission usage of cross-platform applications. A *cross-platform* application is an application that runs on multiple mobile platforms (particularly, on Android and iOS). For example, the official Facebook application releases both Android and iOS versions with almost same functionalities. We start with searching these cross-platform applications by crawling both Android Market and iTunes App Store. Our web crawler collects the information for more than 200,000 Android applications and 300,000 iOS applications. Several data mining techniques are then adopted to match the applications released for different platforms. We find that 10.9% of the applications on Android Market have a replica on iTunes Store. Among them, we select 2,100 applications (1,050 pairs) for further analysis.

In order to analyze the similarity and differences of permission usage, the first challenge is to develop a mapping from Android permissions to iOS permissions, as there is currently no official permission list for iOS. Based on the permission descriptions of Android, we create a permission list for iOS and map these permissions to corresponding Android permissions. Our analysis produces a list of permissions supported on both Android and iOS, and also the list of permissions which are not supported on iOS for various reasons. After that, the second challenge that needs to be solved is to construct an API-to-permission mapping for iOS. This mapping will be used by our static analysis tool to automatically extract the permission usage from corresponding API invocations. We reuse the API-to-permission mapping on Android provided by a recent research [44]. With these mappings available, we build our static analysis tools to perform massive static analysis for cross-platform applications, which use Android Dalvik binaries and iOS Objective-C executables as inputs.

By analyzing 1,050 pairs of cross-platform applications stratified sampled from most popular applications, our results show that 81% of the applications on iOS turn to request additional permissions (2.7 on average), compared to their replicas on Android. The additional permission required are mostly for accessing sensitive resources such as device ID, camera device, user contacts and calendar, which may cause serious privacy breaches or security risks without being noticed. We then further investigate the underlying reasons by separately analyzing third-party libraries and applications' own code. Our results show that the commonly used third-party libraries on iOS, especially advertisement and analytics libraries turn to require more permissions compared to the libraries on Android. Similar phenomenon is also observed from analyzing the applications' own code. The reason of such differences is because sensitive resources can be accessed more stealthily on iOS, compared to Android where all the permissions required by an application has to be shown to the user during installation. We also detect and confirm with the original Android application developers that sensitive permissions may be intentionally avoided as they will appear in the permission list. These results imply that Apple's vetting process is not as effective as what most users might think, particularly in the aspect of protecting users' private data from third-party applications.

The contributions of this work are as follows:

- We establish the first reliable baseline for comparing the security architecture of different mobile platforms by examining permission usage of cross-platform applications. This baseline provides a more comprehensive understanding on characterizing how the platform difference influences applications in security and privacy.
- We investigate the permissions of iOS platform and their relations to Android permissions, which complement the knowledge on the security architecture of contemporary mobile platforms. We also construct a mapping from iOS APIs to iOS permissions, which is essential for automatic static analysis of

permission usage.

- We implement static analysis tools for both Android and iOS applications, which is required to perform massive static analysis for cross-platform applications. Our results show the significant differences in permission usage for Android and iOS third-party applications. The detailed analysis implies that Apple's vetting process is not as effective as Android's explicit permission list mechanism in restricting permission usage by application developers.

5.2 Background and Overview

5.2.1 Security Model: Android vs. iOS

Mobile security is very different from desktop PC – the security goal of mobile operating systems is to make the platforms inherently secure rather than to force users to rely upon third-party security software. Thus, various security mechanisms are adopted and enabled as default on current mobile operating systems. The security features used by Android and iOS are listed in Table 5.1, where the general security models of these two platforms are compared.

Security Feature	**Android**	**iOS**
Permission Notification	Yes	Little*
Approval/Vetting Process	Partial	Yes
Digital Signing	Yes	Yes
Binary Encryption	No	Yes
Sandboxing	Yes	Yes
Data Encryption	Yes	Yes
Damage Control	Yes	Yes
Address Space Layout Randomization	After v4.0	After v4.3

* Only two: using location information and allowing push notifications.

Table 5.1: Security model comparison: Android vs.iOS

Permission Notification: On Android, an application has to explicitly declare what permissions it requires, which indicates the services/data it intends to access. The user is informed with the list of permissions before installing each application, so that he can choose not to install this application if he is unwilling to grant certain permissions to it. On iOS, however, all third-party applications are "equal", in the sense that they are all given the same set of data and resource access as default, without the users' awareness. The only iOS resources which require user to explicitly allow are location information and push notifications. Actually, iOS applications do not really have the permission concepts [1].

Approval/Vetting Process: Apple must approve an application before it is distributed via the iTunes Store, which is the only way to get applications on iOS unless it is jail broken. Apple screens each uploaded application to check whether it contains malicious code or violates Apple's privacy policy before releasing it to the iTunes Store. This vetting process is not well-documented, and there have been cases where malicious applications passed the vetting process but had to be removed later from the iTunes Store [98]. On the Android platform, Bouncer [77] is recently revealed by Android team, which provides automated scanning of Android Market for potentially malicious software. However, different from Apple's vetting process, Bouncer does not require developers to go through an application approval process, it performs a set of analyses on new applications, applications which already exist in Android Market, and also developer accounts.

Signing and Encryption: On both platforms, every application is digitally signed with a certificate. This signature authenticates the identity of the distributor of the application and ensures that the application has not been modified or corrupted since it was signed. The difference is, Android applications are signed by developers but iOS applications are signed by Apple. In addition to signing, iOS application binaries are also partially-encrypted to mitigate unauthorized distribution. Each application downloaded from the iTunes Store has to be decrypted first in memory,

[1]Security entitlements are used in OS X applications, which are semantically similar to permissions. However, entitlements are not fine grained. And most importantly, entitlements are used mainly in OS X applications, not in iOS applications.

each time it is launched.

Other features: iOS uses a sandboxing policy and Android uses UNIX UIDs to separate each individual application. Both platforms provide the service of encrypting users' confidential data, which could also be remotely erased once the device is lost. In addition, both platforms have kill switches in the hands of Google/Apple, which can be used to remove malicious applications from the users' phones remotely. This feature limits the damage of a malicious application by preventing it from spreading widely. Finally, starting from Android v4.0 and iOS v4.3, both platforms provide address space layout randomization to help protect system and applications from exploitation due to memory management vulnerabilities.

From the general security model comparison, we can see that both platforms employ a number of common defense mechanisms, but also have their own distinct features. Android's permission notification has some security advantage, but it also pushes most of the security checking work to its end users who are not expertise in security and may not even read or understand those permissions listed during application installation. From the iOS's perspective, the approval process does provide certain degree of defense against malicious applications. However, its capability is limited and can be bypassed sometimes [98]. Thus, a systematic comparison of the applications on these two platforms is needed to fully understand the effect of these diverse security architectures.

5.2.2 Comparison Framework Overview

To perform a fair comparison, we choose the cross-platform applications on these two platforms as our main comparison objects. The overview of our comparison framework is given in Figure 5.1, and the rest of this chapter is organized according to the flow of this comparison framework. Section 5.3 provides the statistics of the cross-platform third-party applications running on both Android and iOS. Section 5.4 then compares the application-level permissions on Android and iOS in

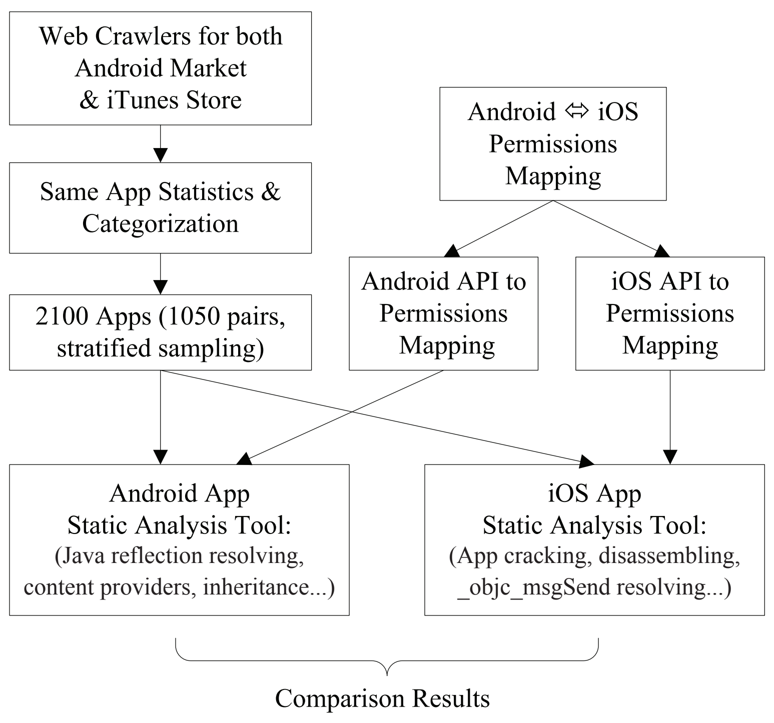

Web Crawlers for both Android Market & iTunes Store
Same App Statistics & Categorization
2100 Apps (1050 pairs, stratified sampling)
Android ⇔ iOS Permissions Mapping
Android API to Permissions Mapping
iOS API to Permissions Mapping
Android App Static Analysis Tool: (Java reflection resolving, content providers, inheritance...)
iOS App Static Analysis Tool: (App cracking, disassembling, _objc_msgSend resolving...)
Comparison Results

Figure 5.1: The overview of our comparison framework.

detail, where we obtain a list of permissions that are supported on both platforms. The design and implementation of our static analysis tools are presented in Section 5.5. Given the permission mapping and static analysis tools on both Android and iOS, we then perform our static analysis on the 2,100 applications selected, and the results are presented in Section 5.6. Finally, we summarize and discuss the contribution of this chapter in Section 5.7.

5.3 Cross-platform Applications

5.3.1 Preliminary Data Collection

In order to find out what are the applications that exist on both Android and iOS, we need to compare their detailed information such as application name, developer, application description, etc. The application product pages from Android Market or iTunes App Store do provide such information, but neither Google nor Apple provides public API for their online application stores. Thus, we build our own web crawlers for both Android Market and iTunes Store.

Our crawler tools first collect the general information for each application by browsing all the category pages. However, this direct method is not enough to thoroughly collect the application information, especially for Android Market, as it only shows at most 800 applications for each category. Thus, we construct our own search strings[2] and collect the information from each returned search result page. When the web crawling ends, the general information of each application (such as ID, name, URL...) is obtained. Following the URL of each application, the crawler then downloads the corresponding page to record detailed information such as developer, description, category, rating, etc. The results of the raw data finally collected from both online app stores are shown in Table 5.2. Unfortunately, we are not able to retrieve part of the application detail pages, which makes the coverage for the detailed information smaller.

	Android Apps	iOS Apps
# of apps for general information (appID, appName, URL...)	230,133	368,752
Total # of apps expected*	250,000 by July 2011**	425,000 by June 2011***
Coverage for general information	92.05%	86.77%
# of Apps for detailed information (description, developer, rating...)	183,070	352,129
Coverage for detailed information	73.23%	82.85%

* All the data are collected from July to August 2011. Thus, we choose the total # of apps announced in July and June as the total expected #.
** Android Wellness News, http://www.androidwellness.com/?p=537 .
*** Apple Event, http://www.apple.com/apple-events/wwdc-2011 .

Table 5.2: Data collected from Android Market and iTunes Store

[2]We utilize every possible permutation of 3-letter string and digit as the search keyword. A popular noun and adjective word dictionary is also utilized to construct the search queries. Together with the different combination of Free and Paid search options, the number of search queries finally produced for Android Market is around 90,000.

5.3.2 Identifying Cross-platform Applications

We regard two applications (one on Android, the other one on iOS) to be two versions of the same *cross-platform* application if they were designed to have the same set of functionalities. For example, the official Facebook application is supposed to provide the same set of functionalities on both platforms. However, given such large sets of applications, it is not practical to manually check each possible combination. Thus, five non-overlapping candidate sets of applications (CS_1 to CS_5) are automatically produced to help identify the cross-platform applications. The detailed descriptions of these five candidate sets are given in Table 5.3.2. Generally, we use Vector Space Model [92] to compare the application descriptions and we use Levenshtein Distance [74] to measure the similarity of application names.

Candidate Set	Conditions*	Unique App Pairs	Percentage	True Positive Rate
CS_1	same appName, same Company, high similarity Description	9,845	5.38%	$\approx 100\%$
CS_2	same appName, same Company, low similarity Description	2,098	1.15%	$> 98.3\%$
CS_3	same appName, different Company, high similarity Description	4,130	2.26%	83.3%
CS_4	similar appName, same Company, high similarity Description	5,408	2.95%	75%
CS_5	same appName, different Company, low similarity Description	8,430	4.60%	6.67%

* The high/low similarity threshold for Description is set to 0.45, and the edit distance threshold for similar appName is set to ≤ 5.

Table 5.3: Candidate sets for cross-platform applications: conditions and statistical results

As shown in Table 5.3.2, the conditions of selecting the candidate pair of applications depend mainly on three attributes: appName, Company and Description. We use Vector Space Model [92], one of the classical models in information retrieval, to compare the descriptions. The similarity between two application descriptions is calculated using:

$$\mathsf{sim}(d_1, d_2) = \frac{\overrightarrow{d_1} \cdot \overrightarrow{d_2}}{|\overrightarrow{d_1}| \times |\overrightarrow{d_2}|} = \frac{\sum_{i=1}^{t} w_{i,1} \times w_{i,2}}{\sqrt{\sum_{i=1}^{t} w_{i,1}^2} \times \sqrt{\sum_{i=1}^{t} w_{i,2}^2}}$$

where $\overrightarrow{d_1}$ and $\overrightarrow{d_2}$ denote the descriptions of two applications (one on Android, and the other one on iOS), after removing stop words, pure numbers and special HTML tags like `<br>`, `<p>` from the descriptions. $w_{i,j}$ is the weight for the i^{th} term in description d_j which is assigned with the frequency of the term. The threshold for the high/low similarity score is set to 0.45 by manual tuning to obtain a good trade-off between the number of false positives and false negatives.

The similarity of the application names (as used in the rule of candidate set CS_4), however, are measured with Levenshtein Distance [74], as names are usually short string which contains only a few characters. The well-known Wagner-Fischer algorithm [108] is used to calculate the distance and the threshold for similar names is set to ≤ 5. One example for the cross-platform applications with similar names is ActDroid on Android and ActPhone on iOS [1] – their name distance is 4 and their description similarity is 0.56.

Out of the three conditions, the condition "same company" is the easiest condition to determine, but it is not as straight forward as it may appear. On each application page of iTunes Store, there are three fields which indicate the ownership of this application – developer company, developer name and copyright company name. However, on Android Market, there is only one field – the developer's name. To make the analyses more tedious and difficult, all kinds of abbreviations are used in these fields (such as Ltd., Inc., LLC) and sometimes even part of the company's own name is abbreviated. Thus, instead of a strict string matching, we choose the "contain" relation – if the developer name on Android contains any of the three fields on iOS (or the other direction holds), then we regard the two given applications are from the same company.

We apply the corresponding rules to our whole data set, and the number of unique application pairs satisfying each candidate set is shown in Table 5.3.2. The percentage shown in the table is calculated base on the total number of Android applications collected with valid detailed information, which is 183,070. After obtaining those candidate sets, we then randomly choose 60 application pairs in each set to perform a manual validation – we manually read the descriptions of these two applications, examine their companies, icons and screenshots to judge whether they are actually cross-platform applications.

We did not find any false positives in the CS_1 set (first row in Table 5.3.2), but did find one false positive in the CS_2 set which is actually caused by parsing error of some non-unicode characters. There are quite a number of cross-platform applications which exist in candidate set CS_3. The main reason of this is that, for some applications, it indicates a general developer name (like "Android Software Programmer") on one of the application stores, which is very different from what is given on the other application store. A typical example is the HCPCS [2] application.

Finally, such analysis enables us to estimate the total number of cross-platform applications that exists on Android and iOS. Applying formula $\sum_i \mathsf{Percent}_{CSi} \times \mathsf{TPR}_{CSi}$ (where $i \in \{1, 2, ...5\}$, and TPR_{CSi} is the true positive rate of the corresponding candidate set), we get the result 10.9%. This indicates that, among those existing third-party applications on Android, approximately 1/10 of them have a replica application provided for iOS.

5.3.3 Stratified Sampling

We choose only application pairs from the first two candidate sets (CS_1 and CS_2) as our cross-platform application resource to perform further static analysis, as they rarely contain false positives. After removing invalid pairs (whose names contains double-question-mark characters) from CS_2, the two sets finally provide a total number of 11,879 unique application pairs. The distribution of these applications according to the 20 iOS categories are given in Figure 5.2, together shown with the distribution of the entire third-party applications on iOS.

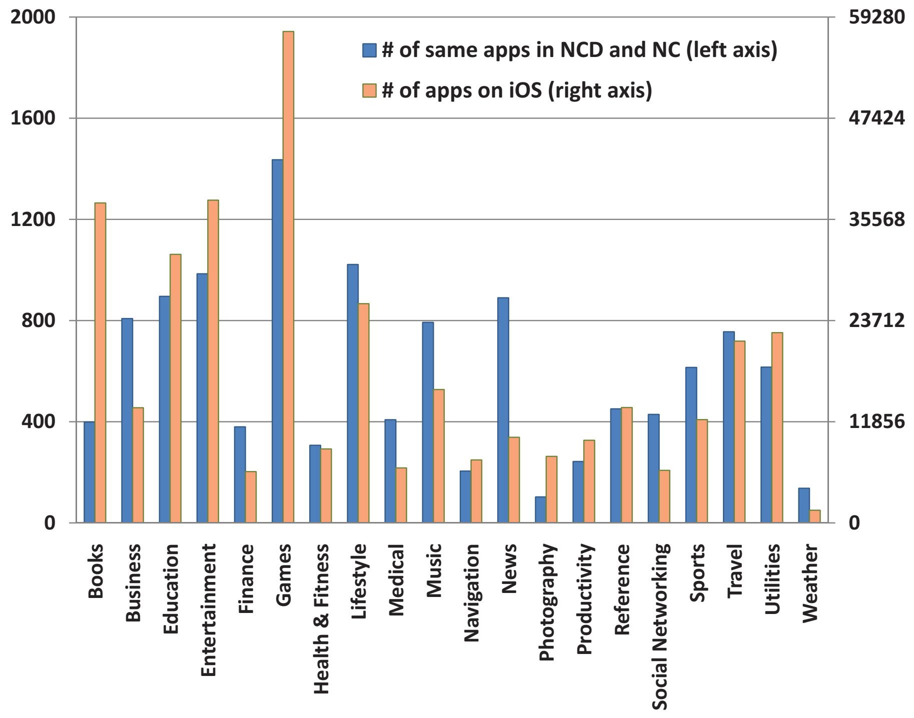


Figure 5.2: The distribution of the cross-platform apps in CS_1 and CS_2 vs. the distribution of the whole iOS app set.

In Figure 5.2, the ratio of the right axis compared with left axis is set to 29.64, which is the same ratio for the total number of iOS applications (352,129) compared to the total number of application pairs in CS_1 and CS_2 (11,879). This setting gives us a direct visual comparison of the distribution of these two data sets. We can see that cross-platform applications are more likely to appear in "Business" and "News" categories (as the blue bar is higher than the orange bar for these two categories in Figure 5.2); while cross-platform applications are less likely to appear in "Books" and "Games".

Among the cross-platform applications in CS_1 and CS_2, we then selected 1,050 pairs (2,100 applications) to perform detailed static analysis on the application executables. To improve the representativeness of this sample set, we perform a stratified sampling for the whole CS_1 and CS_2 application set according to the category distribution. We then pick the most *popular* free applications within each category. The results of the static analysis on these selected applications will be presented in Section 5.6.

5.4 Permission Comparisons

To compare the security architecture of two mobile platforms, one of the most important comparison perspectives is to find out the similarity and differences on restricting access permissions to the applications running on these platforms. However, to our best knowledge, such systematic analysis has not been conducted in the literature. On the Android platform, Google has provided a comprehensive list of application permissions, which is publicly available [10]. However, on the iOS platform, there are no official documentation specifying what permissions are allowed for third-party applications, and what are available only for firmware – this is one of the iOS mysteries we need to reveal in our work.

To make a fair comparison for Android and iOS platforms, we choose Android 2.2 (API level 8, which is released in May 2010), and iOS 4 (released in June 2010). These two OS versions are contemporary, and they have about one year of time to stabilize since they were released (until the time we download those applications), so that the majority of the applications available on these two platforms are compatible with these OS versions. Given the 112 application permissions supported on Android [10], we first find out what is the exact meaning of each permission by understanding the functionality of each API related to this permission, according to the existing Android permission to API mapping provided by [44]; we then carefully go through both online advisories and offline iOS documentations on Xcode[3] to find out whether this permission is supported, and how it is supported on iOS platform. The overview of the analysis result is given in Table 5.4.

[3]Xcode is a suite of tools by Apple, for developing software for Mac OS X and iOS. It provides iOS API documentations for registered developers. http://developer.apple.com/xcode/ .

Type of Permissions	# of Permissions
Not exist anywhere in Android Already deprecated in Android, or no Android API corresponds to this permission.	**3 + 4 = 7**
Android system permissions Only for OEMs, not granted to third-party application developers. i.e., these permissions are only used by applications signed with system keys.	**38**
Permissions not supported by iOS Either iOS does not have such device e.g., removable storage; or iOS does not allow third-party application to have such permission.	**46**
Permissions supported by iOS Third-party applications have these permissions on iOS as default.	**21**
Total # of Permissions	**112**

Table 5.4: Overview of permission analysis result

Among these 112 permissions, three of them (PERSISTENT_ACTIVITY, RESTART_PACKAGES and SET_PREFERRED_APPLICATIONS) have already been deprecated in Android 2.2. And according to the existing Android API-to-permission mapping provided by Felt et al. [44], four permissions (such as BRICK) do not really exist in Android, as there are no API calls, content providers or intents in Android related to these permissions. The rest of the permissions are then divided into three large groups according to our findings.

5.4.1 Android system permissions

The openness concept of Android and its online permission documentation [10] may have given a misleading understanding to both users and developers that a third-party Android application can have any permission. However, this is not true – some permissions are only provided for original equipment manufacturers (OEMs), and are not granted to third-party applications. Examples of these permissions include DELETE_CACHE_FILES, INSTALL_LOCATION_PROVIDER, FACTORY_TEST, etc.

As there are no official documentation specifying which permissions are reserved for OEMs, we obtain the list of system permissions by analyzing permission tags in the frameworks/base/core/res/AndroidManifest.xml file, as system permissions are labeled as android:protectionLevel="signatureOrSystem" or android:protectionLevel="signature" in this firmware configuration file. In order to validate this list, we also write a testing application which tries to request for all 112 permissions and then check those permissions that are denied to this testing application. Finally, 38 permissions are found to be reserved for the system applications, which will not be granted to third-party applications unless users explicitly give them the root privilege.

5.4.2 Permissions not supported by iOS

Among the rest of 67 permissions on Android which can be granted to third-party applications, we are interested in finding out how many of them are also supported by iOS. Surprisingly, our analysis result shows that about 2/3 of these permissions are not supported on iOS. The reasons are either because iOS does not have corresponding functionality/device, or iOS just does not allow third-party applications to have such permissions. Some permissions which are not supported on iOS are given in Table 5.5.

Among the permissions that are disallowed on iOS, it is interesting to notice that some of them are not due to security reasons. Although not officially documented, permissions for global settings which would involve modifying the user experience (UX) are usually disallowed by Apple, and that is one of the reasons why there are still many people who jailbreak their iPhones. Examples of such permissions include MODIFY_AUDIO_SETTINGS, SET_TIME_ZONE, SET_WALLPAPER, WRITE_SETTINGS, etc. Although this would limit the capability of third-party applications, it is still reasonable from the UX perspective. For example, it could be a disaster if you are waiting for an important call, but a third-party application mutes

Reason (1) iOS does not have corresponding functionality/device:		
Permission	Permission Description	iOS Explanation
EXPAND_STATUS_BAR	Allows an application to expand or collapse the status bar.	iOS 4.0 does not have expanded status bar, it is added in after iOS 5.0
MOUNT_FORMAT _FILESYSTEMS	Allows formatting file systems for removable storage.	There is no removable storage for iPhone/iPad/iPod.

Reason (2) iOS does not allow it to third-party applications:	
Permission	Permission Description
KILL_BACKGROUND _PROCESSES	Allows an application to kill background processes.
PROCESS_OUTGOING _CALLS	Allows an application to monitor, modify, or abort outgoing calls.
RECEIVE_SMS	Allows an application to monitor incoming SMS messages, to record or process on them.

Table 5.5: Examples of unsupported permissions on iOS

the sound globally without your awareness.

5.4.3 Permissions supported by iOS

Finally, we obtain 21 permissions which are supported both on Android and iOS. A comprehensive list of these permissions and corresponding explanations are given in Table 5.4.3.

Note that some permission may not be an exact match on Android and iOS. For example, the meaning of READ_PHONE_STATE is not exactly the same. This permission corresponds to at least 18 Android API calls, which can be used to read the device ID, phone number, SIM serial number and some other information. However, on iOS, only device ID is allowed to read since version 4.0 of iOS. Other information is forbidden to be accessed by third-party applications due to security reasons.

Another special case is the ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION permission. There are 20+ API calls related to these two permissions on Android, but all of them only require either one of the

Permission	Description & Explanation
ACCESS_COARSE _LOCATION	Allows an application to access coarse (e.g., Cell-ID, WiFi) location.
ACCESS_FINE _LOCATION	Allows an application to access fine (e.g., GPS) location.
ACCESS_NETWORK _STATE	Allows an application to access information about networks
ACCESS_WIFI_STATE	Allows access to information about Wi-Fi networks.
BATTERY_STATS	Allows an application to collect battery statistics.
BLUETOOTH	Allows to connect to paired bluetooth devices.
BLUETOOTH_ADMIN	Allows to discover and pair bluetooth devices.
CALL_PHONE	Allows an application to initiate a phone call.
CAMERA	Required to be able to access the camera device.
CHANGE_WIFI _MULTICAST_STATE	Allows applications to enter Wi-Fi Multicast mode
FLASHLIGHT	Allows access to the flashlight.
INTERNET	Allows an application to open network sockets.
READ_CALENDAR	Allows to read the user's calendar data.
READ_CONTACTS	Allows to read the user's contacts data.
READ_PHONE_STATE	Allows read only access to phone state. On iOS, only device ID and iOS version are allowed to read, others like SIM ID or phone # are all forbidden.
RECORD_AUDIO	Allows an application to record audio.
SEND_SMS	Allows an application to send SMS messages.
VIBRATE	Allows access to the vibration.
WAKE_LOCK	Allows to disable auto-lock or screen-dimming.
WRITE_CALENDAR	Allows to write the user's calendar data.
WRITE_CONTACTS	Allows to write the user's contacts data.

Table 5.6: Permissions supported on both Android and iOS

two permissions. But on iOS, the corresponding APIs (e.g., CLLocationManager.startUpdatingLocation) need both location permissions. Similar as Android, iOS devices employ a number of different techniques for obtaining information about the current geographical location, including GPS, cell tower triangulation and most inaccurate Wi-Fi connections. However, which mechanism is actually used by iOS to detect the location information is transparent to the application and the system will automatically use the most accurate solution that is available. Thus, for an iOS application which invokes the location-related API calls, it actually requires both ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION permissions.

Note that although there are only 21 permissions both supported on Android and iOS, these permissions cover the access rights to the most common resources/services, including user calendar, contacts, Bluetooth, Wi-Fi state, camera, vibration, etc.

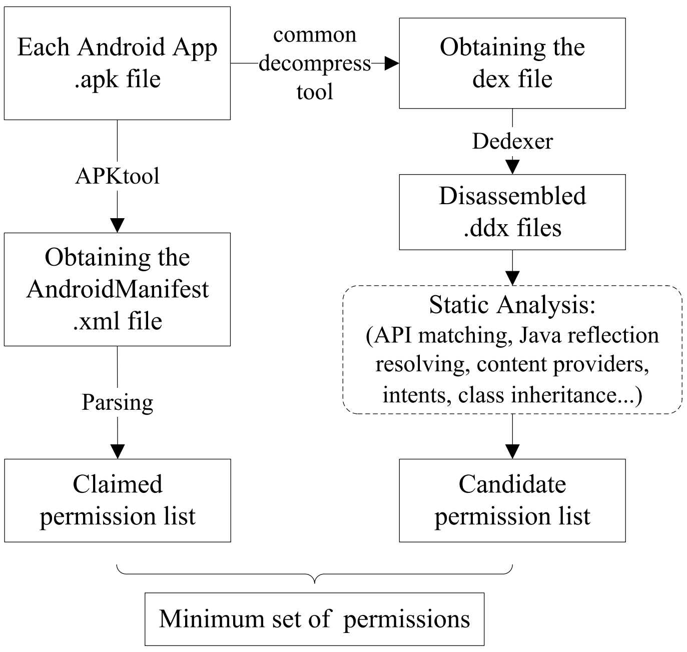
5.5 Static Analysis Tools

To compare the permission usage for third-party applications on Android and iOS, we build static analysis tools for both Android applications (Dalvik bytecode) and iOS applications (Objective-C executables). We will explain the work flow of both tools in this section. The API resolving rates of both analysis tools are reported in Section 5.6.1.

5.5.1 Android Static Analysis Tool

As introduced in Section 5.2, each Android application has a list of permissions that is shown to the user during installation, which is recorded in the AndroidManifest.xml in each application package file. However, this is not the exact list of permissions that this application actually uses – many third-party applications are overprivileged by requesting a superset of permissions [44]. Thus, the ultimate goal of our Android static analysis tool is to output a minimum set of permissions that a given third-party application actually uses. The work flow of our Android tool is given in Figure 5.3.

As shown in Figure 5.3, for each Android application, we first get the corresponding Dalvik executable (DEX), which is then disassembled into a set of .ddx files using the Dedexer tool [84]. With the existing Android API call to permission mapping provided by Felt et al. [44], our tool then perform multiple iterations on parsing and analyzing the disassembled files to produce a candidate list of permissions that this application needs. However, this candidate permission list is not the minimum permission set due to the ambiguity in the Android API-to-permission



Each Android App .apk file
common decompress tool
Obtaining the dex file
Dedexer
Disassembled .ddx files
APKtool
Obtaining the AndroidManifest .xml file
Static Analysis: (API matching, Java reflection resolving, content providers, intents, class inheritance...)
Parsing
Claimed permission list
Candidate permission list
Minimum set of permissions

Figure 5.3: The work flow of our Android static analysis tool.

mapping, which is caused by Android's permission validation mechanism. For example, android.app.ActivityManager.killBackgroundProcesses API call requires either RESTART_PACKAGES or KILL_BACKGROUND_PROCESSES permission – i.e., either permission is sufficient for the application to invoke this API call. Thus, the tool is not able to determine which permission is actually needed when such APIs are observed, which in turn adds unnecessary permissions into the produced candidate permission list. In order to remove those unnecessary permissions and output a minimum set of permissions, our tool then takes the intersection of the candidate permission list and the claimed permission list (parsed from AndroidManifest.xml). This minimum set of permissions will later be compared with the set of permissions used by the replica application on iOS.

There are several technical challenges in analyzing the disassembled application files and outputting the candidate permission list, including: resolving Java reflections; analyzing content providers and intents; tracking third-party libraries.

API calls and Java reflection

On Android, API calls which require permissions may be invoked with different class names due to inheritance. By analyzing class information in the disassembled files, our tool rebuild the class hierarchy so that it can recognize the API calls in the application's own classes, which are actually inherited from API classes. Recognizing these inherited methods are crucial, as they require the same set of permissions as original API calls.

API calls may also be invoked through Java reflection. Reflection gives developers the flexibility to inspect and determine API characteristics at runtime, which allows the leverage of new APIs where available while still supporting the older devices. Utilizing reflection, methods can be invoked with java.lang.reflect.Method.invoke, where the method instance can be obtained previously by calling java.lang.Class.getMethod, and the class instance can be obtained by invoking java.lang.Class.forName (or several other methods). Thus, from the point of each java.lang.reflect.Method.invoke, our tool performs backward slicing [110] to resolve the method name and class name actually invoked – it traverses the code backwards, resolving all instructions that influence the method variable and class variable used in corresponding reflection. We also apply specific heuristics to resolve inter-procedural or inter-classes reflections. However, Java reflection is still a challenging problem [76, 93] and it is not possible to completely resolve all reflections statically. Fortunately, Android applications rarely use reflections according to our results reported in Section 5.6.1.

Content providers and Intents

On Android, content providers store and retrieve data and make it accessible to all applications. Android ships with a number of system content providers for common data such as SMS, contacts, calendar, etc. These content providers are accessed by retrieving it through a URI that identifies the corresponding provider.

The URI can be obtained by either directly specifying a string value or using the public URI constant given in the corresponding provider class. For example, user contacts can be retried by either using “content://contacts” or using android.provider.Contacts.CONTENT_URI constant. Thus, similar as the Stowaway tool [44], our tool maintains a list of all known URI constants in order to recognize content providers. Another important component in Android architecture is Intent. An intent is an abstract description of an operation to be performed, which provides a facility for performing late runtime binding between the code in different applications. We also utilize the intent to permission mapping provided by [44] to discover the permissions needed by sending or receiving Android intents.

Library tracking

Additionally, our static analysis tool maintains a list of popular third-party libraries, and tracks the permission usage of these libraries. This would help us distinguish whether the required permissions are due to these libraries or the applications’ own code.

5.5.2 iOS Static Analysis Tool

Compared to the open Android platform, static analysis on iOS platform is even more challenging, as iOS is a closed-source architecture. Apple tries to control all software executed on iOS devices (iPhone, iPad, or iPod), which has several effects. First of all, the only way for an unchanged iOS device to install third-party applications is through iTunes App Store, which is typically accessed via iTunes. When an application is downloaded via iTunes Store, it will be encrypted and digitally signed by Apple. The decryption key for the application is added to the device’s secure key chain, so that each time this application is launched, it can be decrypted and then start to run on the iOS device.

It is not possible to directly perform static analysis on encrypted application binaries. Thus, before analyzing a third-party application downloaded from iTunes Store, the first step is to obtain the decrypted application binary, which requires to jailbreak the iOS device. *iOS jailbreaking* (which is legal according to the US Copyright Office [71]), allows users to gain root access to the operating system, and also allows iOS device to run even unsigned binary by modifying the system loader.

Jailbreaking the iOS device give us the capability to install the customized GNU Debugger, the Mach-O disassembler oTool and also the OpenSSH server on the device. With these development tools, we are then able to crack any installed application on the device. After obtaining the decrypted iOS application binary, we utilize IDA Pro. [13] to disassemble the binary, and then perform the static analysis on assembly ARM instructions. The work flow of our iOS static analysis tool is given in Figure 5.4. In the following, we will describe several technical problems in building our iOS static analysis tool including extracting Objective-C metadata, marking method boundaries, resolving API calls and constructing the permission to iOS API mapping.

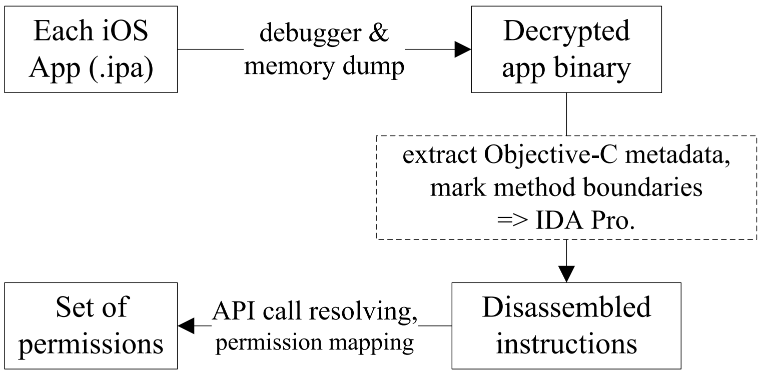

Each iOS App (.ipa)
debugger & memory dump
Decrypted app binary
extract Objective-C metadata, mark method boundaries => IDA Pro.
Disassembled instructions
API call resolving, permission mapping
Set of permissions

Figure 5.4: The work flow of our iOS static analysis tool.

Application Cracking

Development tools such as GNU Debugger, the Mach-O disassembler oTool and also the OpenSSH server are crucial in order to crack an installed application on the device. First of all, the disassembler is used to collect information in the given encrypted application binary, where the most important two fields are cryptoff and cryptsize. cryptoff indicates the offset in the binary file from where the file content begins with encrypted data; while cryptsize records the length of the encrypted data. We then launch the application and attach the debugger to this application's process. Now as the application has been started, it is sure that the system loader has performed the decryption. Thus, according to cryptoff and cryptsize, we can calculate the position of corresponding decrypted binary data in memory and dump it to a separate file. Finally, we replace the encrypted part with the decrypted data in the original binary and transmit the decrypted binary out of the device through the SSH server.

Extract Objective-C metadata

Objective-C runtime specification requires the metadata that describe all Objective-C classes, protocols, and categories that are implemented in a binary. These metadata are stored in a number of segments, where we can extract the class name, the instance method list, the class method list, the instance variable list, the property list, the protocols that a class conforms to, and the categories that add new methods to an existing class. For a method, we can get the method name, the method signature string, and the start address of the method body. For a class, we can also get its super class so that the whole class hierarchy implemented in the binary can be reconstructed. Protocol information does not directly tell the position of implemented methods (probably imported from external frameworks), but it shows the extra methods that an object may respond to. The method and class description data extracted will be used in the next stage to mark the method boundaries.

Mark method boundaries

This stage reconstructs the method boundaries. IDA is only able to mark a very small portion of methods, especially when the symbols are stripped in the binary. The underlying reason is that iOS binaries are allowed to interchangeably use two instruction sets, ARM and THUMB, which have different instruction sizes and alignments. Without knowing the starting point of a method, IDA may not correctly disassemble the binary data into code, and very likely to treat a code fragment as a data entry by mistake. Thus, we use the metadata previously collected to guide IDA, where the starting address of a method can be found in the class structure. And the disassembling mode (ARM/THUMB) can be decided by the last bit of that address. There are a number of implementation complexities in this stage, where mistakes made by IDA are corrected. For example, we need to detect and delete any unexpected data items or misaligned instructions generated in IDA's first round analysis; we also need to mark no-return functions and handle the conditional return instructions; otherwise IDA cannot correctly determine the method boundaries.

Resolve API calls

After disassembling all methods in IDA, the next step is to resolve the API calls in the disassembled Objective-C code. In order to resolve the API calls, the key step is to handle the objc_msgSend function. In the iOS executable, all accesses to a method or attribute of an Objective-C object at runtime utilize this objc_msgSend function, which is used to send messages to an instance of class in memory [81]. The first parameter to objc_msgSend is called the receiver, which is a pointer that points to the instance of the class that is to receive the message. The second parameter to objc_msgSend is the selector of the method that handles the message. The rest of the parameters form a variable argument list containing the arguments to the method. This objc_msgSend function is responsible for dynamically resolving and invoking the proper method that corresponds to the given selector. Thus, in order to observe what API call has been invoked in the application code, one has to locate the receiver and selector that are passed to each objc_msgSend function, and determine the corresponding class name and method name. To statically determine the API call related to each observed objc_msgSend, we adopt the backward slicing and forward constant propagation proposed by [40] in our iOS static analysis tool.

Permission to iOS API mapping

In the last step, in order to output the set of permissions required for an iOS application, our tool also needs the API-to-permission mapping on iOS to check whether a resolved API call requires certain permissions. However, unlike on Android platform, there is no existing API-to-permission mapping available for iOS platform. Thus, we manually create such mapping according to the 21 permissions both supported on Android and iOS. For each permission, we first thoroughly collects the functionalities related to this permission by checking each Android API call which requires this permission, and then we carefully go through iOS documentations to find corresponding API calls which perform these functionalities.

Most permissions can be directly recognized through corresponding API classes and methods, for example, user contacts are operated through ABPerson and ABAddressBook related APIs on iOS. However, some permissions like CALL_PHONE and SEND_SMS require further analysis of the parameter value. For example, given an API call as [[UIApplication sharedApplication] openURL:[NSURL URLWithString:[NSString stringWithFormat:@"tel:123-456-7890"]]], this will only launch the phone dialer when the string parameter starts with "tel:" prefix. SEND_SMS has both forms – the SMS sending view can be triggered by openURL with "sms:" prefix; an application can also call API such as MFMessageComposeViewController.setMessageComposeDelegate to send SMS. We carefully handle each of the cases for every resolved API call and corresponding parameter values in order to detect such permissions.

5.6 Comparison Analysis Results

We applied our static analysis tools to the 1,050 pairs of cross-platform applications, which is a stratified sample among the whole application set. The direct outputs of our analysis tools are the lists of permissions required for the cross-platform applications on iOS and Android. By obtaining such permission lists, we are then able to compare the permission usage of those applications. After finding out the permission usage differences of these two platforms, we further investigate the underlying reasons from two perspectives: third-party libraries and applications' own code. Finally, in order to have a comprehensive view of the permission usage of these two platforms, an additional analysis is carried out on the permissions that are not supported on iOS, but only on Android.

5.6.1 API Resolving Rate of Analysis Tools

On the Android platform, Java reflection is found to be commonly used [44], which is also confirmed by our observation. Among the 1,050 Android applications, we found that 629 (60%) use Java reflections to make API calls. However, the absolute number of reflections invoked is only 5,148, which means each application only makes 4.9 reflection calls on average. This is a small amount compared to 7,087 API calls [4] made by each application on average. Our Android static analysis tool is able to resolve 4,017 (78%) reflections, which indicates that out of thousands of API calls issued per application, only 1.1 API call is not resolved on average. Our tool failed to resolve the reflection call if the method name or class name is not generated statically. For example, some reflections invoke java.lang.Class.getDeclaredMethods with no parameters which simply return an array of methods, and then according to some dynamic rules, the code will pick one of the methods to invoke. Cases like this are very difficult to be resolved in a static manner, which is one of the limitations in our current implementation.

On the iOS platform, on average, 16.82% of the instructions in each application belong to C/C++ code and 82.85% instructions belong to Objective-C code; while the rest 0.33% are dummy instructions or the instructions that our tool is unable to interpret. Our tool is able to capture all the invocations for the API calls that

[4]This number does not contain the number of method calls that are defined within the application, which are not API calls by definition.

are invoked through C/C++ functions; while for the API calls in the disassembled Objective-C code, our tool is able to resolve 93% of the _objc_msgSend encountered. There are at least two cases where a given _objc_msgSend is usually not able to be resolved. 1) The corresponding class instance is passed from the runtime as an argument of a callback function. Callback functions are common for mobile applications, as they are the major mechanisms to handle user interaction events sent from the runtime. 2) The class instance is retrieved from a collection object such as an array that can hold any types of objects. Such a limitation also exists in other static analysis tools on iOS platform [40]. Although it has certain impact on the static analysis which ideally requires to resolve all objc_msgSend methods, it only has quite limited influence in our experiments. The reason is that actually a number of API call invocations for the same permissions are usually observed in an application, so that missing a small portion of API calls will not make our tool overlook the corresponding permission in most cases.

5.6.2 Comparisons on Both-supported Permissions

Our first comparison focuses on the 21 permissions that are both supported on Android and iOS. We are interested in finding out how differently these permissions are required on the two platforms for these cross-platform applications. The results show that 3,652 permissions are required by 1,050 applications on Android, which has on average 3.5 permissions per application. In comparison, 6,562 permissions are required by 1,050 iOS applications, which has on average 6.2 permissions per iOS application. 849 (81%) of the applications on iOS need additional permissions compared to its Android version.

Among those 21 permissions, some of them are required almost equally by the applications on both platforms. For example, INTERNET permission is required by 996 Android applications, and 1,011 iOS applications; BLUETOOTH is required by 10 Android applications and 13 iOS applications. However, some permissions are required much more often by iOS applications compared to Android applications. The top 12 permissions that are required more often on iOS compared to Android are listed in Table 5.6.2.

Permission	# of Android app	# of iOS app	**Main Reason**	**Only on iOS**[1]
ACCESS_WIFI_STATE	122	782	OS API	671
CAMERA	76	382	App & Lib	314
READ_PHONE_STATE	455	732	App & Lib	309
ACCESS_COARSE_LOCATION	274	543	OS API	301
ACCESS_FINE_LOCATION	317	543	OS API	241
ACCESS_NETWORK_STATE	632	782	OS API	237
VIBRATE	201	318	App & Lib	202
READ_CONTACTS	94	263	App & Lib	183
WRITE_CONTACTS	41	180	App & Lib	151
SEND_SMS	16	129	App & Lib	123
RECORD_AUDIO	43	109	App & Lib	78
READ_CALENDAR	6	82	App & Lib	77

Permission	Only on Android	On both platforms	App / Lib (Purely Lib) Ratio[2]
ACCESS_WIFI_STATE	11	111	95% / 35% (5%)
CAMERA	8	68	82% / 36% (18%)
READ_PHONE_STATE	32	423	74% / 61% (26%)
ACCESS_COARSE_LOCATION	32	242	80% / 44% (20%)
ACCESS_FINE_LOCATION	15	302	77% / 52% (23%)
ACCESS_NETWORK_STATE	87	545	96% / 18% (4%)
VIBRATE	85	116	60% / 63% (40%)
READ_CONTACTS	14	80	65% / 43% (35%)
WRITE_CONTACTS	12	29	67% / 43% (33%)
SEND_SMS	10	6	81% / 25% (19%)
RECORD_AUDIO	12	31	99% / 1% (1%)
READ_CALENDAR	1	5	52% / 51% (48%)

[1] The number of applications which have the corresponding permission only in the iOS version, but not in the Android version.

[2] This is a break-down for the source of the permission requirements: (a) from the app's own code; (b) from third-party libraries; (c) purely caused by the third-party libraries (i.e., the corresponding permission is not required by the app's own code). The base of the ratio is the number in column "Only on iOS".

Table 5.7: Permissions with greatest disparity that are required by the applications on Android and iOS.

Among those permissions that are required much more often on iOS, some of them are actually caused by the underlying difference in the OS API, as marked in Table 5.6.2. For example, Android provides APIs for checking the status (e.g., availability or connectivity) of different network types (e.g., WiFi or 3G). However, iOS APIs do not distinguish the different network types, but just check the reachability of a given host or IP address. Thus, a given iOS application needs both ACCESS_NETWORK_STATE and ACCESS_WIFI_STATE permissions when an API call like reachabilityWithHostName is observed in this application. Another typical case is the ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION, which has been explained in Section 5.4.3. If we add the number of Android applications which have ACCESS_COARSE_LOCATION and the number of Android applications which have ACCESS_FINE_LOCATION permission, the sum will be very close to the number of iOS applications which have ACCESS_FINE_LOCATION permission alone. Similar result applies to ACCESS_NETWORK_STATE and ACCESS_WIFI_STATE permissions.

However, the differences of the other 8 permissions listed in Table 5.6.2 are caused by the differences in the code of the cross-platform applications. For example, the famous chatting application – eBuddy Messenger [3] does not require READ_PHONE_STATE permission on Android, however, on its iOS version, we observe 7 locations in its code which read the device ID. Another typical instance is the famous free game "Words With Friends" [4] application. Compared to its Android version, the additional permissions required by its iOS version include (but are not limited to):

- BATTERY_STATS, as API call UIDevice.setBatteryMonitoringEnabled is observed;
- CALL_PHONE, as UIApplication.openURL with "tel:" parameter is observed in IMAdView.placeCallTo and two other locations;
- CAMERA, as UIImagePickerController.setSourceType with argument value 0x1 (which is UIImagePickerControllerSourceTypeCamera) is observed in MobclixRichMediaWebAdView.takePhotoAndReturnToWebview;

- FLASHLIGHT, as AVCaptureDevice.setTorchMode is observed in MobclixRichMediaWebAdView.turnFlashlightOnWithSuccess; etc.

Some of these API calls are listed above in order to provide a detailed view of the static analysis. As can be seen from such detailed example, an iOS application does turn to require more permissions compared to its replica version on Android.

More interestingly, we also check the most popular game application Angry Birds, although it does not belong to our sampling set as it is not free on iOS. We found out that, compared to its Android version, Angry Birds on iOS additionally reads the user contacts data, as API call ABAddressBookGetPersonWithRecordID and ABAddressBookCopyArrayOfAllPeople are observed in the code section of CCPrivateSession.getArrayOfAddressBookEmailAddressesNamesAndContactIDs and four other locations.

As shown in Table 5.6.2, our findings in the comparisons (on the 21 permissions both supported on Android and iOS) show that iOS third-party applications turn to require more permissions on some devices (such as camera and vibration) and are more likely to access the sensitive data such as device ID, user contacts and calendar. Thus, our next step of analysis is to find out the underlying reason why such phenomenon exists. As one may notice from the examples given above, some of these permissions are actually caused by the third-party libraries used in these applications (such as IMAdView and MobclixRichMediaWebAdView classes in the WordsWithFriends application). Thus, our next step is to analyze the permission usage of the third-party libraries on both platforms.

5.6.3 Reason Probing 1: Permission Usage of Third-party Libraries

In order to analyze the permission usage of the third-party libraries, first of all, we need to identify the third-party libraries within each application. As there are no clear boundaries that an included library or package in a given application is written by the application developer or is actually a third-party library, we first process the whole application set to count for the different package names (on Android) or class names (on iOS). Then the packages or classes that appear in more than 10 applications (and at least belong to two different companies) are automatically collected. We then manually check this list to identify the third-party libraries, which include advertisement libraries, analytic libraries or just third-party development libraries. Some of the packages or classes are combined because they actually belong to the same third-party library. Finally, we identified 79 third-party libraries on Android and 72 third-party libraries on iOS that are commonly used. The 8 most commonly used advertising and analytics libraries on Android are listed in Table 5.6.3, and the 8 most common libraries on iOS are listed in Table 5.6.3.

Library Name	App Ratio	Permissions*
com/google/ads	18.1%	ANS, INT
com/flurry/android	16.7%	LOC, INT
com/admob/android/ads	13.0%	LOC, INT
com/google/android/apps/analytics	11.1%	ANS
com/millennialmedia/android	8.7%	ANS, INT, RPS
com/adwhirl	5.1%	LOC, INT
com/mobclix/android/sdk	3.9%	ANS, LOC, INT, RPS
com/tapjoy	3.5%	INT, RPS

* ANS = ACCESS_NETWORK_STATE; INT = INTERNET; LOC = ACCESS_COARSE/FINE_LOCATION; RPS = READ_PHONE_STATE.

Table 5.8: Most common ads and analytics libraries on Android

By tracking the code regions of these libraries, our static analysis tools are able to determine the origin of the API call which causes the permissions in each application. We are then able to find out the permission usage of third-party libraries on both platforms, as shown in Table 5.6.3 and Table 5.6.3. The data from these two tables clearly indicate that libraries on iOS turn to require more permissions compared to Android third-party libraries. Thus, the permission usage difference for the cross-platform applications on Android and iOS is indeed partially caused

Library Name	App Ratio	Permissions[*]
Flurry	19.2%	LOC, INT, RPS
AdMob	17.2%	LOC, INT, CON, RPS
GoogleAds	16.4%	ANS, AWS, INT, RPS, SMS, VIB, WAK
Millennial Media	11.6%	ANS, AWS, LOC, INT, CON, RPS, VIB
Google Analytics	10.1%	INT
AdWhirl	7.6%	ANS, AWS, LOC, INT, RPS
TapJoy	7.5%	ANS, AWS, INT, RPS
Medialets	6.4%	LOC, INT, RPS, SMS, VIB

[*] ANS, INT, LOC, RPS refer to Table 5.6.3; AWS = ACCESS_WIFI_STATE; SMS = SEND_SMS; CON = READ/WRITE_CONTACTS; VIB = VIBRATE; WAK = WAKE_LOCK.

Table 5.9: Most common ads and analytics libraries on iOS

by the third-party libraries.

In order to quantify the influence of the third-party libraries on the permission usage difference, for each permission, we first identify those applications which have the corresponding permission only on its iOS version, but not on Android version (which is the “Only on iOS” column in Table 5.6.2). We then check the source which causes this permission for each application, and the results are summarized in the last column of Table 5.6.2. The first two ratios in this column represent the percentage of applications that: (a) the application’s own code causes the corresponding permission; (b) the third-party libraries used in the application cause the permission. As can be seen from the table that the sum of these two ratios is usually more than 100%. This is because in some applications, both the application’s own code and the third-party library used require this permission. Thus, the third ratio is given, which shows the application percentage that the corresponding permission is purely caused by the third-party libraries, not by the applications’ own code.

From the data given in the last column of Table 5.6.2, we can see that the third-party libraries do have certain impacts on the difference of application permission usage. For example, 40% applications which requires additional VIBRATE permission on iOS is purely because of the third-party libraries it uses. And from Table 5.6.3 we can find the exact source – libraries such as GoogleAds, Millennial

Media and Medialets all need VIBRATE permission. Thus, any application which includes these libraries will in turn need this permission. Similar links can be drawn from Table 5.6.2 and Table 5.6.3 for other permissions such as READ_CONTACTS and READ_PHONE_STATE.

Comparing the data in Table 5.6.3 and Table 5.6.3, the result shows that the most commonly used third-party libraries, especially advertisement and analytics libraries on iOS, require much more permissions compared to the libraries on Android. The most possible reason for this phenomenon could be because *on iOS, the user private data can be collected more stealthily compared to on Android, where applications need to list out the permissions they require during installation.* The permissions on iOS are granted to third-party applications as default without users' awareness, which gives certain freedom for these advertisement and analytics libraries to access user data and device resources.

In order to confirm our findings on the third-party iOS libraries, we further check each library listed in Table 5.6.3 to see whether it is an open-source library. For the open-source libraries, e.g., the AdWhirl [5], we manually look into its code and confirmed all five permissions it requires. For those closed-source libraries, such as Flurry, we also find evidences that this library collects the device ID in its official documentation [6], which mentioned "Because Apple allows the collection of UDID for the purpose of advertising, we continue to collect this data as the Flurry SDK includes AppCircle, Flurry's mobile advertising solution".

From the data given in the last column in Table 5.6.2, one can observe that third-party libraries only contribute a portion of the difference for application permission usage; the other part of the difference is caused by the applications' own code. By removing the permissions that are purely caused by third-party libraries, our static analysis tools manage to output the lists of permissions that are caused by the applications' own code on both platforms. The comparison result shows that 1,050 Android applications now require 3,045 permissions, while iOS applications require 5,609 permissions – there is still an obvious difference for the application permission usage on the two platforms. This difference leads us to investigate further into the applications' code logic so as to find out the underlying reasons.

5.6.4 Reason Probing 2: Microanalysis on Application Code Logic

In order to perform the microanalysis on the code logic of the cross-platform applications, it will be ideal to have the full access to the applications' source code. Applications which are open-source on both platforms are rare, given the fact that iOS platform only has very little open-source applications. We manage to find 8 applications that are open-source on both platforms, as shown in Table 5.10. We retrieve the source code of these applications and analyze the underlying reasons of their permission usage differences. The detailed API information collected from closed-source applications is also utilized to assist the analysis. From our manual inspection, there are at least two main reasons which cause the iOS applications turn to use more permissions compared to their corresponding Android applications.

	Permissions[*]on Android	Permissions[*]on iOS
WordPress	LOC, CAM, INT, VIB	LOC, ANS, AWS, CAM, INT, RPS
Mixare	LOC, CAM, INT, WAK	LOC, CAM, INT, WAK
MobileOrg	INT	ANS, AWS, INT
andRoc/iRoc	CWM, INT, RPS	CWM, INT, WAK
Mp3tunes	ANS, INT, RPS, WAK	INT, RPS
ZXing(Barcodes)	AWS, CAM, INT, CON, VIB, WAK	ANS, AWS, CAM, FLA, INT, CON
DiceShaker	INT	INT, VIB
MobileSynth	None	None

[*] ANS = ACCESS_NETWORK_STATE; AWS = ACCESS_WIFI_STATE; CAM = CAMERA; CWM = CHANGE_WIFI_MULTICAST_STATE; LOC = ACCESS_COARSE/FINE_LOCATION; INT = INTERNET; RPS = READ_PHONE_STATE; CON = READ/WRITE_CONTACTS; WAK = WAKE_LOCK; VIB = VIBRATE; FLA = FLASHLIGHT.

Table 5.10: The 8 applications which are open-source on both platforms and their permission usage.

Coding difference

The most natural reason which has been expected is the implementation difference for the cross-platform applications on Android and iOS. For example, ACCESS_NETWORK_STATE permission is only required by the iOS version of WordPress, but not on the Android version. In the iOS version, this permission is required because several methods in WPReachability class are invoked, which are used to test the reachability to the WordPress hosts. However, for the Android version of WordPress, there is no code for testing any reachability. When posting a blog to the server, for example, the code will simply check the return value to see whether the connection is successful or failed. However, on iOS, many classes of WordPress will actively check the reachability beforehand, and notify the users if the network is not reachable. Such implementation difference causes the result that WordPress on iOS requires the additional ACCESS_NETWORK_STATE permission. Similar evidence can be found in the code of MobileOrg application.

Such coding difference is also the main reason causing the difference in requiring the CAMERA permission. Take the popular applications eBuddyMessenger and SmackIt for examples. In their iOS versions, the photo of their user profile setting can be chosen either from the picture library, or directly from taking the photo with the device's camera. However, their Android versions do not provide such "taking photo" option. Note that such implementation difference does not only exist in the applications' own code, but also for the same third-party libraries on two platforms. For example, the CAMERA permission is required by OpenFeint library on iOS, but not by its Android version, which is caused by the same reason mentioned above.

Intentional avoidance

We start to explain this reason from the evidence found in the open-source application WordPress, which is one of the most popular applications on both platforms. Compared to its Android version, WordPress on iOS also requires the additional

READ_PHONE_STATE permission. In the WordPress iOS code, runStats method of WordPressAppDelegate reads the device_uuid, os_version, app_version, language, device_model, and then sends them to " http://api.wordpress.org/iphoneapp/update-check/1.0/ " to check whether this application needs to be updated. But actually, in order to check the update, the information of device_uuid is not really needed. On the Android platform, the code of WordPress does perform almost the same functionality – in the wpAndroid class, uploadStats method tries to retrieve the information of uuid, device_version, device_language, mobile_country_code, mobile_network_type, etc., and sends these data back to WordPress server to check for update.

However, there is a major difference of the WordPress code on Android compared to the WordPress code on iOS. In its iOS code, the uuid is retrieved by accessing UIDevice.uniqueIdentifier, which is really the device unique ID. However, on its Android version, the uuid read is a random ID which is unique, but not really the device ID. It is a unique ID that is randomly generated and stored as the first record in WordPress's own SQLite database on the Android device. Note that information such as os_version, device_language, mobile_country_code, mobile_network_type is not considered as privacy data, so retrieving such information does not require READ_PHONE_STATE permission on both Android and iOS. Thus, the difference of obtaining uuid is the reason causing the difference of requiring READ_PHONE_STATE permission.

The special way of obtaining the uuid on Android makes us believe that the programmers intentionally try to avoid triggering the READ_PHONE_STATE permission on Android. This is further confirmed by consulting one of the WordPress developers, who gives the explanation as: "a random id is better than the device id because it doesn't require that permission which reads quite poorly as 'read phone state and identity' ". Thus, why they do not try to avoid using the device ID on iOS is because of the same reason mentioned in Section 5.6.3 – on Android, an application needs to show the permissions it requires to the user during installation; while on iOS, no such notification is given to the user. We believe this is also the main reason which causes the difference in requiring permissions such as READ_CONTACTS and READ_CALENDAR. But unfortunately, due to the limited access to the applications' source code, we are not able to get the ground-truth evidence for these permissions, as what has been done for the READ_PHONE_STATE permission.

5.6.5 Comparisons on Full Permissions

Previous analyses focus on the 21 permissions that are both supported on Android and iOS, without taking into account of the additional 46 application permissions that are only supported on Android platform. Thus, the purpose of our final analysis is to find out how frequently the 1,050 Android applications apply for these 46 permissions that are not supported on iOS, and what are the characteristics of these permissions.

Taking into account of the 46 permissions, our result shows that the 1,050 Android applications require an additional number of 1,056 permissions in total. As shown in Table 5.11, the most frequently required permission that is not supported on iOS is WRITE_EXTERNAL_STORAGE, which are required by more than 80% of the applications. This is actually quite normal for Android applications. Different from iOS devices which have 4GBytes to 32GBytes of internal storage, Android devices usually have less than 200MBytes of internal storage. Thus, all Android devices support external storage such as microSD card. As a result, Android applications which need to write their application data usually need to require the WRITE_EXTERNAL_STORAGE permission in order to save the free space of the internal storage.

After removing the WRITE_EXTERNAL_STORAGE permission, the rest in those 46 permissions that are not supported on iOS are required very rarely (only 218 in total, and 0.2 per application on average). Such a result shows that the 21 both-supported permissions are the most common permissions used for third-party ap-

Permission not supported on iOS	# of Android Apps
WRITE_EXTERNAL_STORAGE	838
RECEIVE_BOOT_COMPLETED	40
GET_ACCOUNTS	35
GET_TASKS	27
CHANGE_WIFI_STATE	14
READ_LOGS	13
RECEIVE_SMS	11
DISABLE_KEYGUARD	10

Table 5.11: Usage of unsupported permissions on iOS for Android apps

plications. For those permissions listed in Table 5.11, they are simply not allowed on iOS. For example, the RECEIVE_SMS allows an application to "monitor incoming SMS messages, to record or perform processing on them", which is not allowed for iOS third-party application for security reason. iOS also does not allow any third-party application to "read the low-level system log files", which is the meaning of READ_LOGS permission on Android. And as mentioned in Section 5.4.2, permissions like CHANGE_WIFI_STATE which modifies the global settings and in turn changes the user experience are also forbade on iOS.

5.7 Discussion

Our work made the first attempt towards systematically comparing mobile application security for diverse mobile platforms. We established the first reliable baseline by comparing permission usage of cross-platform applications. Two most popular mobile platforms, Android and iOS, are chosen to investigate how the platform difference influences application in security and privacy. We investigated the permissions of iOS and their relations to Android permissions, which are previously unclear. We also constructed the mapping from iOS APIs to iOS permission required by automatic permission analysis. With these mapping available, we built our static analysis tools to perform massive static analysis for cross-platform applications.

Our findings have shown that third-party applications on iOS require more permissions compared to applications on Android, especially for the access to sensitive resources like device ID, camera and user contacts data. The immediate implication of such findings is that users on iOS have higher risks of leaking their private data compared to on Android. The underlying reason is because users' private data can be accessed more stealthily on iOS compared to on Android, where applications need to list out the permissions they require during installation. Although our findings do not necessarily imply that Android has better security compared to iOS, because iOS restricts certain permissions to third-party applications (as shown in Section 5.4) and also orthogonal research works [45, 26] have shown that third-party applications on Android could gain additional permissions by launching privilege-escalation attacks. Nevertheless, our findings indicate that it is insufficient to solely rely on Apple's approval process, and the security on iOS can be further improved if a permission notification mechanism is adopted on iOS platform, similarly as on Android.

We remark that our comparison analysis framework is still in its preliminary stage. Although we tried our best to search for all the related API calls that will lead to a corresponding permission, it is very difficult to test the completeness of this API-to-permission mapping. We may still miss some API calls that will require the corresponding permission. Permissions like INTERNET or BLUETOOTH relate to a huge number of Objective-C APIs provided by different frameworks and also some C libraries. For example, a third-party iOS program could just create its own socket and access the internet, which requires our tool to adopt a much more thorough way for the static analysis. The test for the completeness of the API mapping will be investigated in our future work.

Finally, an important assumption is used in our analysis when comparing iOS version and Android version of the cross-platform applications. We assume these two versions should ask for similar permissions and should have similar functionalities. There could be cases that the developers first implemented a full-functioned version on one platform and are just in the process of integrating new functionality into the other version on another platform. However, we believe that this assumption holds when it is considered in a large scale context. In order to validate this assumption, massive manual work has to be involved, which is left as future work.

Chapter 6

Data Interpretation Diversity in Information Hiding

6.1 Introduction

Diverse ways of interpreting the same set of data are used by steganographic file systems (stegfs), which are intended to provide plausible deniability to data owners in the event that they are forced to disclose their secret data [19]. A stegfs hides encrypted user data among dummy data that contain only pseudo-random bits. Without the correct password, it is not possible to differentiate user data from dummy (based on the assumption that the output of the block cipher is indistinguishable from random bits [18, 19]), even for an adversary who understands the mechanisms of the file system and is able to gain access to the storage devices. Given different password, the data in the storage can be interpreted in different ways, which allows a data owner to selectively reveal some directories/files by revealing some passwords, but disclaim the existence of his sensitive data.

To be believable, the disclaimer of the data owner must be consistent with the information that the adversary is able to gather about the file system. This is much more challenging to achieve in modern computing environments when the user data are encrypted and stored in shared network storage. Compared to portable and local storage, network storage dramatically increases the availability and accessibility of user data. However, it also brings new challenges in securing user data. With shared network storage, the adversary is no longer limited to a single snapshot of the disk content at the point of attack. Instead, the adversary could now locate the physical server machines being used [89] and quietly amass multiple snapshots of the file system over a period of time before launching his attack. The additional knowledge that the adversary gleams from the multiple snapshots must be factored into the stegfs design.

In earlier stegfs designs [19, 79, 64, 85], dummy data are created when the disk is formatted and remain static thereafter. These schemes are effective against adversaries who only see the final state of the storage, but cannot defend against adversaries who possess multiple snapshots of the storage. Indeed, changes among different snapshots not only reveal the location of secret data, but could even be utilized to recover the access keys (for example, when the first scheme by Anderson et al. [19] is utilized). Recent stegfs schemes, which are proposed to defend against multiple-snapshots attacks, either cannot guarantee the integrity of user data even under legitimate data operations [37, 38], or require a trusted agent to manage all the user passwords and dummy data [113], which effectively presents a single point of disclosure for user passwords.

In this chapter, we propose a multi-user stegfs for shared storage systems, which is named as DRSteg – Dummy Relocatable Steganographic file system. DRSteg is designed to meet the following requirements:

- Security: To provide plausible deniability of secret data in a multi-user environment in which the adversary could obtain multiple snapshots of the storage content. This protection should extend to any user even when the storage server and all the other users are completely compromised, i.e., they have surrendered all the information in their possession.

- Usability: To guarantee data integrity, and at the same time enable individual users to trade off between deniability and system performance.

To the best of our knowledge, DRSteg is the first stegfs that allows I/O operations observed on shared storage to be plausibly attributed to dummy data without requiring a trusted agent as used by Zhou et al. [113]. In addition, our work also manages to increase the deniability provided to individual users by sharing dummies among multiple users in the system. It is technically challenging to satisfy both the security and usability requirements, especially when dummies are shared. DRSteg incorporates a special dummy relocation mechanism that enables individual users to distinguish dummies from other users' data (in order to free dummies without destroying data), and to prevent adversaries from discerning the difference between dummy and user data even after obtaining multiple snapshots.

This is also the first work that formalizes the deniability achieved by a multi-user stegfs. The formalization enables us to develop a tunable mechanism for users to balance between deniability and system responsiveness. In DRSteg, the deniability enjoyed by individual users could be maintained beyond a specified threshold, whether or not all the other users are fully compromised. The amount of dummy operations is controlled individually; a user who specifies a more aggressive amount enjoys higher deniability at the expense of slower file operations.

To substantiate the usability of DRSteg, we present results of an empirical evaluation using file operation logs collected from 12 graduate students in our school. The results confirm that DRSteg is capable of achieving a wide range of user-specified deniability levels. We also implemented a prototype of DRSteg as a file system module in Linux kernel. Performance experiments on the prototype show that security and performance can be traded off against each other.

6.2 Problem Definition

6.2.1 Threat Model

Figure 6.1 depicts our model of a multi-user file system. In the model, user data are stored on a shared storage. The stegfs functionalities are implemented in the client module that runs on the user computers. This client module is secured so that sensitive data that are operated on as well as any passwords used for encrypting and decrypting the data are protected. The storage server manages the shared storage devices which provide block-level operations, including DAS (direct attached storage) and SAN (storage area network). Different from the model where the server manages all the user passwords [113], the storage server and shared storage in our model are not stegfs specific.

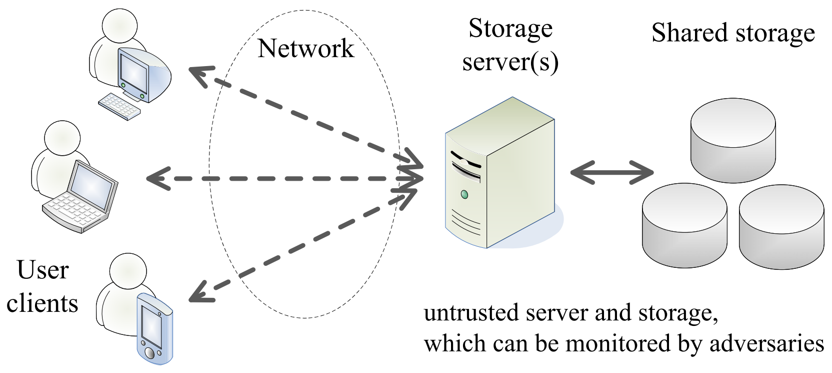

Network
Storage server(s)
Shared storage
User clients
untrusted server and storage, which can be monitored by adversaries

Figure 6.1: A multi-user stegfs with untrusted shared storage

The server and the storage devices are not trusted. This means that an adversary may infiltrate the server or the storage devices directly (or the backup of these devices) to copy and analyze the stored content. Although our scheme provides better protection when the communication between users and the server is anonymized, it is not a necessary condition for DRSteg to provide deniability to users. We will analyze the deniability of DRSteg under different scenarios in Section 6.4.

In this work, we focus on adversaries who are after the user data, and we explicitly rule out considerations of sabotage like overwriting/deleting data and denial of service. The threat posed by the adversary thus hinges on two factors: (a) his knowledge of the file system state, and (b) his access to the users of the system. These two factors together determine the adversary's ability to make deductions about the hidden data on the storage, and to verify any claims elicited from the users.

The first factor, knowledge of the storage state, is characterized by the number of observations of the storage content. An adversary who is able to access the storage only once (i.e., at the point of attack) only gains a *single snapshot* of the storage. An example is someone who is captured by criminals and forced to reveal all the contents in his portable drive. However, when the adversary has more than one chance to access the storage, he can record *multiple snapshots*. The information in those snapshots is then utilized to deduce the existence of secret data.

The second factor that defines the adversary's ability concerns his access to the users. Here, we make the following assumption:

Victim isolation assumption. In coercing information from the users, it would be effective for the adversary to interrogate them separately and cross-check the information elicited. Placed in isolation, a victim knows neither which other users have been compromised nor what information they have surrendered. Consequently, each victim has to assume the worst, i.e., that all the other users are compromised and all their secrets are revealed. He thus has to independently decide what data he can hide without being contradicted by other users' disclosure.

Multi-user encrypting file systems [12, 25, 28] are inadequate under the victim isolation assumption, as it is not safe for a user to claim his data to belong to someone else. A solution is to use dummy blocks, which should be operated on in similar ways as encrypted data blocks in order to defend against multiple snapshot attacks.

6.2.2 Definition of Deniability

To formalize the threat, an adversary has access to a sequence of snapshots $\mathsf{S} = \{\mathsf{s}_1, \mathsf{s}_2, \ldots, \mathsf{s}_\mathsf{T}\}$ of the stegfs partition on the disk, where s_T is the snapshot at the time of coercion. Following the victim isolation assumption, the adversary extracts all the passwords from other users (P') at the time of attack, and also coerces the victim to reveal his passwords $\mathsf{P_t} = \{\mathsf{p_1}, \mathsf{p_2}, \ldots, \mathsf{p_t}\}$. The adversary then utilizes the passwords obtained to decode the information in each snapshot.

Let $\mathsf{H_i^{dummy}}$ and $\mathsf{H_i^{data}}$ denote the hypotheses that an allocated block $\mathsf{blk_i}$ is a dummy block and a data block, respectively. Let $\mathsf{e_i}$ denote the evidence on $\mathsf{blk_i}$ observed from S, and $\mathsf{E} = \{\mathsf{e_i}\}$ the aggregate evidence across all the disk blocks. We define the *plausible deniability* of $\mathsf{blk_i}$ as follows.

Definition 1 *Given the evidence* $\mathsf{E} = \{\mathsf{e_i}\} = \mathsf{S} \cup \mathsf{P}' \cup \mathsf{P_t}$, *where* $\mathsf{S} = \{\mathsf{s_1}, \mathsf{s_2}, \ldots, \mathsf{s_T}\}$ *is a sequence of snapshots taken by the adversary and* $\mathsf{P}' \cup \mathsf{P_t}$ *is the set of passwords revealed to the adversary (along with the blocks decrypted with these passwords), the deniability of an allocated block* $\mathsf{blk_i}$ *is the posterior probability that* $\mathsf{e_i}$ *was generated by operations on dummy block* $\mathsf{blk_i}$*:*

$$\mathsf{deny_i} = Pr(\mathsf{H_i^{dummy}} | \mathsf{e_i}) \tag{6.1}$$

A steganographic file system is said to be ***α-deniable*** *if*

$$\mathsf{deny_i} \geq \alpha$$

for all $\mathsf{blk_i}$ *that cannot be decrypted with* $\mathsf{P}' \cup \mathsf{P_t}$, *for any* $t \geq 1$ *of the user's choice.*

An α-deniable stegfs guarantees that any evidence gathered by an adversary (e.g., disk images across multiple snapshots) is caused by dummy data operations with at least a probability of α. This means that a user of the system can attribute the evidence to dummy operations without revealing his secret data.

6.3 Design of DRSteg

DRSteg is designed to enable a user to selectively disclose some of his data, while enjoying α-deniability for the rest of the data that he is withholding from the adversary. We begin this section with an overview of the DRSteg design, before presenting the detailed data structures and implementation considerations.

6.3.1 Overview of DRSteg

In DRSteg, each user must be able to protect his data with different passwords, so that he can surrender some data but not others. To achieve α-deniability for the data blocks that he is withholding, our approach is to (a) enforce a joint ownership for allocated disk blocks to prevent the adversary from associating with certainty a withheld block with any particular user, and (b) introduce dummy blocks that are operated on at runtime, so that changes to the withheld blocks can be plausibly explained by dummy operations.

We realize the joint ownership through a voting protocol. For every allocated block, m ownership shares are created and distributed to m users, including the user who requested for the block (also known as the creator). A block can subsequently be altered or freed only after all the m shares have been garnered from consenting owners. By following this policy, we ensure that the block is never deallocated without the creator's share, yet the creator of the block is obfuscated among the share owners. The creator may use an allocated block either for his data or as a dummy.

For each user, the disk blocks that hold his data are protected by one of his passwords $p_1, p_2, \ldots, p_n$. The number of passwords n is expected to vary from user to user, though we use the same symbol n across users for brevity. Moreover, the passwords are generated as a hash chain [73], i.e., $p_l = h(p_{l+1})$ for a hash function h and $1 \leq l < n$ (as illustrated in the upper part of Figure 6.2). By supplying any password p_l, $1 \leq l \leq n$, the user can access all the secret data at and below level l.

As for those disk blocks that are allocated as dummies, no bookkeeping information is maintained to track them directly; otherwise, the adversary can simply demand the bookkeeping information from the users, and with it discover the dummy blocks in the file system. Instead, a dummy block can only be identified through the cooperation of its owners: Each shareholder of the block checks whether it is protected with one of his passwords; if not, the block is a potential dummy – it may indeed be a dummy, or it may hold the data of some other user. It is freed in the same way as data blocks, i.e., after gathering m shares.

In the event of an attack, our DRSteg design allows a coerced user to supply some password p_t, $1 \leq t < n$, to the adversary and deny the existence of the passwords p_j for $t < j \leq n$. The data blocks that are protected by p_j then appear to be potential dummies, thus enabling the user to hide the existence of the data.

6.3.2 Detailed Design of DRSteg

Drawing on the approaches introduced above, we now put together the concrete DRSteg design. Each user u keeps track of a set of blocks A_u on which he currently holds a share. Moreover, each password p_l protects a set of data blocks $D_{u,l}$. The set difference $A_u - \cup_l D_{u,l}$ gives the blocks that exclude u's data, and dummy blocks are the allocated blocks that contain nobody's data, i.e., $\cap_u(A_u - \cup_l D_{u,l})$. Figure 6.2 depicts our detailed design for DRSteg (the encryption is done at the granularity of individual blocks).

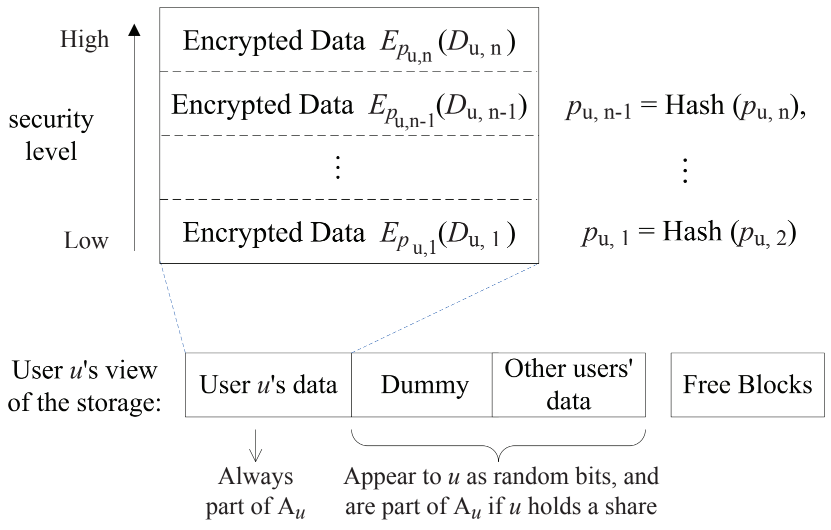

High
security level
Low
Encrypted Data $E_{p_{u,n}}(D_{u,n})$
Encrypted Data $E_{p_{u,n-1}}(D_{u,n-1})$
Encrypted Data $E_{p_{u,1}}(D_{u,1})$
$p_{u,n-1}$ = Hash ($p_{u,n}$),
$p_{u,1}$ = Hash ($p_{u,2}$)
User u's view of the storage:
User u's data
Dummy
Other users' data
Free Blocks
Always part of A_u
Appear to u as random bits, and are part of A_u if u holds a share

Figure 6.2: Key management and user view of the storage

Whenever a user u requires a disk block blk from the file system to write data or dummy patterns, a free disk block is allocated and shares of the block are also created. One share is given to u, while the remaining shares of blk are distributed to other users u', i.e., $A_u \leftarrow A_u \cup \{\mathsf{blk}\}$ and $A_{u'} \leftarrow A_{u'} \cup \{\mathsf{blk}\}$. If user u encrypts data with his password p_l and stores it in blk, then $D_{u,l} \leftarrow D_{u,l} \cup \{\mathsf{blk}\}$.

Any user u may propose the deletion of a block in his A_u. The deletion is effected only after all the users who hold shares of the block have acquiesced. Obviously, if the block holds the data of user u', he would relocate the data before supporting the deletion. This is to avoid leaving clues for differentiating between dummy and data blocks.

With DRSteg, user u can surrender any password p_t, $1 \leq t < n$ and claim that data blocks in $A_u - \cup_{1 \leq j \leq t} D_{u,j}$ are not his data. Claiming that data blocks in $D_{u,j}$ for $t < j \leq n$ are dummy blocks is plausible since they also appear in $A_{u'} - \cup_l D_{u',l}$ of other users u' who hold shares of the blocks.

Joint ownership of blocks

We implement the joint ownership of disk blocks through a voting protocol and two data structures – a set of encrypted user share boxes (USB) and a global voting table (GVT) in clear text. A USB is used to track the A_u of each user, and a GVT records the votes surrendered by users. Two other structures are additionally maintained in clear text in the storage: a list of the users' public keys, and a bitmap to track the allocation status of the disk blocks.

When a user allocates a disk block blk_i, he 1) sets the bit of this block to "1" in the bitmap; 2) creates m shares and writes them to the corresponding USBs; 3) writes the encrypted/random data content to the block. The format of each encrypted share is given as $E(K_{pub,u}, i)$, an encryption of i with a user's public key. The encrypted shares denote the ownership of this block. A block $\mathsf{blk} \in A_u$ if the share $E(K_{pub,u}, i)$ exists in the USB of user u. The m owners of a block include the creator and $m - 1$ other users randomly selected from the public-key list.

Any of the m owners can subsequently initiate the deletion of the block blk by writing i to the global voting table (GVT) and removing his share from his USB. To support the deletion, other owners also contribute their shares into GVT. When the number of accumulated shares of a block reaches m, this block can be removed from GVT and its bit in the bitmap is set to “0” (indicating that this block is free). The share constitution ensures that the block can be deallocated only when block creator signals his agreement by surrendering his share to the GVT.

Management of data blocks

In order to provide plausible deniability against multiple-snapshot attacks, disk blocks that contain data must be managed carefully so that they leave the same evidence as operations on dummy blocks.

First, consider the modification of secret data. By comparing snapshots, the adversary may discover that the content of a block changes before all the m shares are added into GVT. This would never happen to a dummy block according to our voting protocol. Therefore, instead of overwriting data blocks, each user always migrates his updated content to new blocks, and initiates the deletion of the outdated blocks in GVT so that they will be freed in due course. However, the initiation of the deletion operation is delayed, in order to break the temporal correlation between the allocation of new blocks and the deallocation of outdated blocks.

Next, consider the case where some user’s data block is registered for deallocation in GVT by other users. If the user never concurs, the adversary will suspect that the block contains data, since deallocation of dummy blocks are supported readily. To avoid suspicion, the user has to migrate the content to a fresh disk block, before relinquishing his share to the old data block.

In real implementations, the block creating operations are carried out immediately, but the voting (including removing shares from USB and writing block numbers into GVT) are delayed. We pass the voting operations to a background user process that survives beyond user log-off. The background process repeatedly initiates the deletion of a block in its pool after sleeping for a random duration. This makes the operations for data blocks plausible since the creation and voting could be caused by either creating and freeing dummy blocks or creating, modifying and freeing data blocks.

6.3.3 Discussions

Comparing to naive designs

There also exist alternatives in designing a multi-user steganographic file system. A naive one could simply let each user manage his own blocks (including data and dummy). Since dummy blocks are no longer shared, one has to create many more dummy blocks in order to achieve the same deniability compared to our design, when anonymous channels are used between the users and the storage server. When this channel is not anonymized, our design still provides similar security and disk utilization compared to the naive design. The deniability provided by DRSteg under both scenarios is analyzed in the next section.

Encryption of the block shares

Another security issue relates to the encryption of the shares in USB. If the shares are stored in clear text, it will be straightforward for an adversary to identify who the owners of any particular block are. By encrypting the shares, the owners of any block are obfuscated so long as multiple blocks have been allocated between snapshots. In this way, our approach safeguards shareholders from being earmarked to be the next target of coercion.

Organization of the user passwords

The last design issue concerns the organization of the user passwords. One option is to have only one password in each account and to give every user multiple accounts. Under coercion, a user reveals some of his accounts and tries to hide the remaining ones. However, this simple option fails when the adversary captures all the users of the system. When that happens, the adversary can check whether there are m shares among the surrendered accounts for every allocated block; if not, there must exist more user accounts. This is why we choose to allow multiple passwords (for different security levels) in each user account.

Organizing multiple passwords in a hash chain has been proposed in other stegfs [19, 57, 85], and its one-way property meets our requirements well. Under coercion attack, the disclosure from surrendering t independent passwords is the same as giving up the t lowest-level passwords in a hash chain. Thus, in our system design, the hash chain mechanism is chosen due to the performance and usability benefits gained compared to independent passwords.

6.4 Plausible Deniability of DRSteg

Having introduced the design of DRSteg, we now quantify the deniability it provides under a spectrum of progressively challenging attack scenarios. Based on the last and most demanding scenario, we then show how to operationalize the DRSteg design so as to sustain the system security above user-specified deniability thresholds. Table 6.1 summarizes the terms and notations which are used in the analysis.

6.4.1 Analysis of Deniability

We first expand Equation 6.1.

$$\mathsf{deny_i} = \Pr(\mathsf{H_i^{dummy}}|\mathsf{e_i}) = \frac{\Pr(\mathsf{e_i}|\mathsf{H_i^{dummy}}) \times \Pr(\mathsf{H_i^{dummy}})}{\Pr(\mathsf{e_i})} \tag{6.2}$$

According to our problem formulation in Section 6.2, the adversary is capable of taking multiple snapshots of the storage content. He may also augment the snapshots with secrets that he coerced from one or more users. The following attack scenarios differ on the amount of secrets thus extracted, and deserve particular

Notation	Explanation
$\mathsf{S} = \{\mathsf{s}_1, \mathsf{s}_2, \dots, \mathsf{s}_\mathsf{T}\}$	Snapshots (of the stegfs partitions) taken by the adversary.
$\mathsf{P}_\mathsf{t} = \{\mathsf{p}_1, \mathsf{p}_2, \dots, \mathsf{p}_\mathsf{t}\}$	Passwords revealed to the adversary under coercion.
$\mathsf{E} = \{\mathsf{e}_\mathsf{i}\} = \mathsf{S} \cup \mathsf{P}' \cup \mathsf{P}_\mathsf{t}$	Evidence possessed by the adversary.
$\mathsf{s}_\mathsf{k} = \{\mathsf{BLK}, \mathsf{USB}, \mathsf{GVT}\}_\mathsf{k}$	$\mathsf{BLK} = \{\mathsf{blk}_\mathsf{i}\}$: Blocks in the stegfs partition (blk_i is the i-th block). $\mathsf{USB} = \{\mathsf{USB}_u\}$: User share boxes ($\mathsf{USB}_u$ is the USB of user u). GVT: Global voting table.
$\mathsf{blk}_\mathsf{i} = \langle \mathsf{text}_\mathsf{i}, \mathsf{flag}_\mathsf{i} \rangle$	text_i: If blk_i is dummy, text_i contains random bits; If blk_i holds user data, $\mathsf{text}_\mathsf{i} = E(\mathsf{p}, \mathsf{plaintext}_\mathsf{i})$ flag_i: A flag indicating whether blk_i has been allocated.
$\mathsf{H}_\mathsf{i}^{\mathsf{dummy}}, \mathsf{H}_\mathsf{i}^{\mathsf{data}}$	Hypothesis that blk_i is a dummy/data block in s_T.

Table 6.1: Summary of notations used

attention in deploying DRSteg. These scenarios will be further evaluated in Section 6.5. In the following analysis, we consider the case where the evidence contains two snapshots. The analysis extends easily to multiple snapshots. Note that Equation 6.2 implicitly takes the frequency of these snapshots into consideration by evaluating e_i, i.e., the more frequently snapshots are taken, the more information e_i would include.

Passive-adversary scenario

In this scenario, the adversary may be curious and has not resorted to force, or he may not be ready to expose himself just yet. Thus he only relies on the snapshots collected, i.e., the evidence $\mathsf{E} = \mathsf{S}$. By comparing any two recorded snapshots (s_1, s_2), the adversary could observe a lot of user activities, e.g., new blocks being created, deleted, and etc.

Let us first consider the creation of new blocks. A block blk_i is created between s_1 and s_2 if flag_i changes from 0 in s_1 to 1 in s_2. Let $\mathsf{crt}^{\mathsf{data}}$ represent the net number of data blocks created between s_1 and s_2, and $\mathsf{crt}^{\mathsf{dummy}}$ the net number of dummy blocks created in the same period. $\mathsf{ttl}_{\mathsf{s}_2}$, $\mathsf{ttl}_{\mathsf{s}_2}^{\mathsf{dummy}}$, and $\mathsf{ttl}_{\mathsf{s}_2}^{\mathsf{data}}$ denote, respectively, the total number of allocated blocks, the total number of dummy blocks, and the total number of data blocks in $\mathsf{s_2}$. Given an evidence that $\mathsf{blk_i}$ is newly allocated, the probability that $\mathsf{blk_i}$ is a dummy block in $\mathsf{s_2}$ is calculated with Equation (6.2) as

$$\mathsf{deny_i} = \frac{\mathsf{crt^{dummy}}}{\mathsf{ttl_{s_2}^{dummy}}} \times \frac{\mathsf{ttl_{s_2}^{dummy}}}{\mathsf{ttl_{s_2}}} \Big/ \frac{\mathsf{crt^{data}} + \mathsf{crt^{dummy}}}{\mathsf{ttl_{s_2}}} = \frac{\mathsf{crt^{dummy}}}{\mathsf{crt^{data}} + \mathsf{crt^{dummy}}}$$

This derivation extends to block deletion and other evidence listed in Table 6.2. Denoting the number of data/dummy block operations between $\mathsf{s_1}$ and $\mathsf{s_2}$ by $\mathsf{op^{data}}$ and $\mathsf{op^{dummy}}$, the deniability can be calculated as $\mathsf{op^{dummy}}/(\mathsf{op^{data}} + \mathsf{op^{dummy}})$.

Evidence	**DRSteg operation**
$\mathsf{flag_i}$ changes from 0 to 1 and new shares appear in some USBs	Create $\mathsf{blk_i}$ as a new dummy or data block
A share of $\mathsf{blk_i}$ is moved from USB_u to GVT	User u votes to delete $\mathsf{blk_i}$
$\mathsf{flag_i}$ changes from 1 to 0, and $\mathsf{blk_i}$'s entry is removed from GVT	Delete $\mathsf{blk_i}$ as enough votes are present in GVT
Some combination of the above	Some combination of the above

Table 6.2: Evidences and the corresponding DRSteg operations

For an individual user u in DRSteg, let $\mathsf{op}_u^{\mathsf{data}}$ denote the number of data blocks operated on in $\cup_l D_{u,l}$ between $\mathsf{s_1}$ and $\mathsf{s_2}$, and $\mathsf{op^{dummy}}$ denote the number of dummy blocks operated on in the system. The deniability that DRSteg provides for u under this scenario is expressed as

$$\mathsf{deny_{u,i}} = \frac{\mathsf{op^{dummy}}}{\mathsf{op}_u^{\mathsf{data}} + \mathsf{op^{dummy}}} \tag{6.3}$$

Anonymous-channel scenario

Once the adversary starts to coerce users, by the victim isolation assumption in Section 6.2, one has to assume that all of the users have been captured and be wary about offering conflicting information to the adversary. In this scenario, we consider a victim u who discloses the passwords for up to level t of his files and attempts to hide his remaining data, when all the other users are compromised ($\mathsf{E} = \mathsf{S} \cup \mathsf{P}' \cup \mathsf{P_t}$). We assume that all the user requests were sent through an anonymous channel to the storage server, so that the adversary is not able to trace each request to a specific user.

With all the passwords of every user except u, the adversary not only sees all the data of the other users, he also uncovers the dummy blocks for which the ownership is limited to those users. The only outstanding blocks are those on which u holds a share (A_u). Figure 6.3 illustrates the distinction between various groups of blocks in the system, and also the ones used in the calculation of $\mathsf{deny_{u,i}}$.

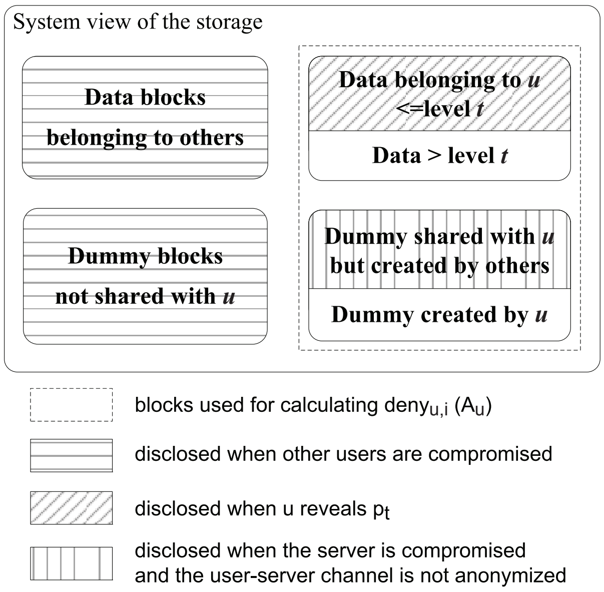

System view of the storage
Data blocks belonging to others
Data belonging to u <=level t
Data > level t
Dummy blocks not shared with u
Dummy shared with u but created by others
Dummy created by u
blocks used for calculating deny_u,i (A_u)
disclosed when other users are compromised
disclosed when u reveals p_t
disclosed when the server is compromised and the user-server channel is not anonymized

Figure 6.3: System view of allocated blocks

Taking into account the organization of the user data into different password levels n and $\mathsf{P_t}$, operations on data blocks in level t and below are disclosed to the adversary. Let $\mathsf{op}_{u,l}^{\mathsf{data}}$ denote the number of data blocks in $D_{u,l}$, and $\mathsf{op}_{u}^{\mathsf{dummy}}$ denote the number of dummy blocks recorded in USB_u. The deniability of a user u (who has revealed p_t) is a function of the undisclosed blocks held by him:

$$\mathsf{deny_{u,i}} = \frac{\mathsf{op}_{u}^{\mathsf{dummy}}}{\sum_{l>t} \mathsf{op}_{u,l}^{\mathsf{data}} + \mathsf{op}_{u}^{\mathsf{dummy}}} \tag{6.4}$$

The disclosed passwords do not affect $\mathsf{op}_{u}^{\mathsf{dummy}}$ in the above equation. Therefore, a bigger t improves the deniability for the data of user u being withheld from the adversary. This is intuitive, since a bigger t means that there is less user data to be hidden among the fixed pool of dummy blocks.

Worst-case scenario

When the user-server channel is not anonymized and the storage server is compromised by the adversary, the adversary is able to distinguish the creator from other share holders by monitoring the requests sent to the server. Under such a scenario, a user cannot utilize the dummy blocks that are not created by himself to provide deniability for his secret data (even if he is one of the owners of these dummy blocks). This leads to the worst-case deniability $\mathsf{deny_{u,i}}$ for DRSteg since $\mathsf{op}_u^{\mathsf{dummy}}$ in Equation 6.4 only contains dummy blocks created by user u himself.

6.4.2 α-deniable DRSteg

We now show how to operationalize the dummy manipulation mechanism to secure DRSteg under the worst-case scenario described above. Specifically, we demonstrate how to manipulate dummy data to maintain the deniability above a given threshold α_T, thus making DRSteg α_T-deniable.

Number of Dummy Blocks to Manipulate

Let $\sigma_{u,l} = \mathsf{op}_{u,l}^{\mathsf{dummy}}/\mathsf{op}_{u,l}^{\mathsf{data}}$. The number of dummy blocks operated on by u, $\mathsf{op}_u^{\mathsf{dummy}} = \sum_l \mathsf{op}_{u,l}^{\mathsf{dummy}} = \sum_l \mathsf{op}_{u,l}^{\mathsf{data}} \times \sigma_{u,l}$. Substituting into Equation (6.4), we have

$$\mathsf{deny_{u,i}} = \frac{\sum_l(\mathsf{op}_{u,l}^{\mathsf{data}} \times \sigma_{u,l})}{\sum_{l>t} \mathsf{op}_{u,l}^{\mathsf{data}} + \sum_l(\mathsf{op}_{u,l}^{\mathsf{data}} \times \sigma_{u,l})} \tag{6.5}$$

In order to ensure that every $\mathsf{blk_i} \in A_u$ meets the deniability threshold of α_T no matter which password level user u chooses to surrender, we need

$$\mathsf{deny_{u,i}} = \frac{\sum_l(\mathsf{op}_{u,l}^{\mathsf{data}} \times \sigma_{u,l})}{\sum_l \mathsf{op}_{u,l}^{\mathsf{data}} + \sum_l(\mathsf{op}_{u,l}^{\mathsf{data}} \times \sigma_{u,l})} > \alpha_T$$

Simplifying the above equation, we get

$$\sigma_{u,l} > \frac{\alpha_T}{1-\alpha_T} \tag{6.6}$$

Since $\sigma_{u,l} = \mathsf{op}_{u,l}^{\mathsf{dummy}}/\mathsf{op}_{u,l}^{\mathsf{data}}$, Equation (6.6) implies that to achieve the target deniability threshold α_T, the number of dummy blocks manipulated must be at least $\frac{\alpha_T}{1-\alpha_T}$ times $\mathsf{op}_{u,l}^{\mathsf{data}}$, the number of data operations.

Controlling dummy operations

Having determined the number of dummy blocks to manipulate, we give the procedures for controlling the dummy manipulation in DRSteg in order to achieve the deniability configured by users.

There are three types of operations on the dummy blocks – creating, deleting and voting – among which dummy creation is the easiest to control. When a user logs in at security level l, he configures σ_l (which is bigger than $\frac{\alpha_T}{1-\alpha_T}$). If x free blocks are allocated for creating or modifying a secret file, then after a random delay, the DRSteg client creates $x \cdot \sigma_l$ dummy blocks to maintain the deniability.

Deletion is more complex because a user does not know which blocks are really dummy blocks (he can only identify blocks that are not his data, as illustrated in Figure 6.2). To conceal the deletion of x data blocks, the DRSteg client has to delete $x \cdot \sigma_l$ dummy blocks. This is done by moving the shares of $x \cdot \sigma_l$ randomly selected blocks in $A_u - \cup_l D_{u,l}$ from USB_u to GVT after a random delay. Although some of these $x \cdot \sigma_l$ blocks may be data blocks of other users, the respective data owners will turn these (data) blocks into dummy anyway as explained next.

Now suppose that user u' logs in, and discovers that a block $\mathsf{blk} \in A_{u'}$ has been put up in GVT for deletion. If blk does not contain his data, i.e., if $\mathsf{blk} \in (A_{u'} - \cup_l D_{u',l})$, u' will support the deletion by adding his votes on blk in GVT. If blk is a data block of u' (i.e., $\mathsf{blk} \in \cup_l D_{u,l}$), then u' has to migrate the content to a new block before voting for the deletion. As discussed in Section 6.3.2, this is to avoid leaving clues that blk contains user data.

Security Discussions

There are several security concerns relating to dummy manipulation. First, in our current design, every block operation is either a direct data operation or the effect of a data operation. Besides introducing random delays, their association could be masked by breaking each of the dummy creations and block deletions into smaller steps and interleaving them with data block operations. In addition, DRSteg could initiate dummy operations independently of data operations. These enhancements will be incorporated in future work.

Second, the parameter $\sigma_{u,l}$ is of special interest to the adversary, who might force the victims to reveal their choices of $\sigma_{u,l}$. With the $\sigma_{u,l}$ values, the adversary may estimate the actual number of data block operations, thus limiting the victims' flexibility to attribute as dummy those data blocks that they are trying to hide. To substantiate his denial in the event of an attack, DRSteg furnishes each user u with a fake $\sigma_{u,t}^{\mathsf{fake}}$ at log-out, where t is the password level that the user is willing to disclose. $\sigma_{u,t}^{\mathsf{fake}}$ is calculated as the ratio between the number of blocks claimed to be dummy (including dummy blocks and hidden data blocks), and the number of revealed data blocks: $\sigma_{u,t}^{\mathsf{fake}} = (\Sigma_{l>t}\mathsf{op}_{u,l}^{\mathsf{data}} + \mathsf{op}_{u}^{\mathsf{dummy}})/\Sigma_{l\leq t}\mathsf{op}_{u,l}^{\mathsf{data}}$.

Another potential security threat is, if the adversary is able to take snapshots of the storage content with infinitesimal delay, he may be able to distinguish dummy blocks from data blocks. Troncoso et al. [103] showed that this distinction is possible because data blocks belonging to the same file are often accessed one after another, whereas dummy blocks are accessed individually and are not likely to exhibit the same access pattern. To mitigate against such a threat, one possible solution is to introduce dummy files into DRSteg. A dummy file would span several dummy blocks, which are then accessed sequentially like data blocks. In order to present similar access pattern as data files, dummy files should also be accessed frequently. Such an improvement in dummy file operations is left for future work.

6.5 Evaluation

6.5.1 Empirical Evaluation on Deniability

To investigate DRSteg's ability to maintain user-specified deniability thresholds under multiple-snapshot attacks, we perform an empirical evaluation by re-playing file operations logged in a typical office environment. We deployed a logger to record the file operations (operation type and time) on the computers of 12 graduate students in our lab. Over 9 days, we recorded more than 50,000 *user* file operations[1].

We begin by mirroring the user files of all 12 computers in DRSteg, which add up to about 1 Tbyte of data. We also initialize the same number of dummy blocks, making the original utilization of data blocks 0.5. The shares for data and dummy blocks are distributed randomly among the 12 users. We assume that users are automatically logged out from the stegfs system after some period of inactivity (10 minutes in our experiments), and they login again right before their next observed data operations. For each session, the user enters the password to one of his security levels l (randomly chosen by our simulator) and picks a $\sigma_{u,l}$ value (chosen to follow a power-law distribution $p(\sigma) \propto L(\sigma)\sigma^{-\xi}$ assuming that more users will tend to choose lower σ values to minimize overhead). We set $\alpha_T = 0.4$, $\sigma_{min} = 0.7$ and $\xi = 3.0$ for all users. The parameters and statistics are summarized in Table 6.3.

We use the first two days of logs to warm up DRSteg. As the remaining seven days of traces are executed, we take a snapshot of the disk image every 10 minutes. Figure 6.4 shows the deniability for one of the (randomly chosen) users by comparing each successive snapshot with the first one.

Figure 6.4(a) shows the deniability under the *passive-adversary* scenario, calculated with Equation 6.3. The upper graph gives the deniability with respect to block creation evidence, while the lower is for delete operations. As seen from the graphs, sharing dummy blocks among users enables individual users to enjoy high

[1]We assume that the operating system and software programs are not installed in the stegfs partition.

Simulation Parameters	**Value**
Number of users	12
α_T	0.4
Number of security levels	5
Interval before auto logout	10 mins
Average number of shares per block	3
User-log Statistics	**Value**
Total logging time	9 days
Number of file operations	50,113
Data blocks created	26.613 GB
Data blocks deleted	80.069 GB
Data blocks modified	160.317 GB
Simulated DRSteg Statistics	**Value**
Initial amount of data blocks	1011.34 GB
Initial amount of allocated blocks	2022.68 GB
Number of user sessions	294
Final amount of data blocks	970.60 GB
Final amount of allocated blocks	1995.89 GB

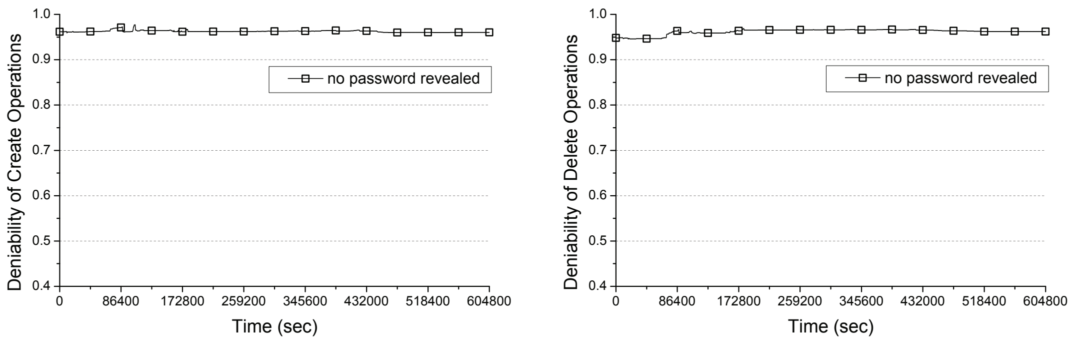
Table 6.3: Simulation parameters and statistics

deniability.

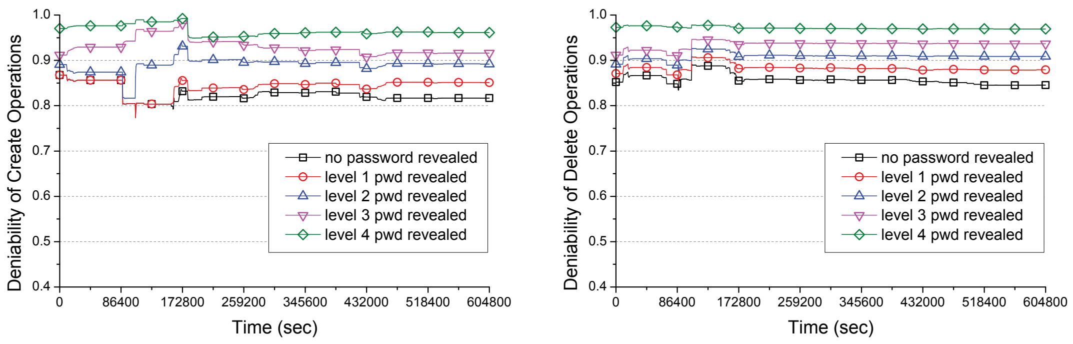
Next, we examine Figure 6.4(b) for the *anonymous-channel* scenario, which is calculated with Equation 6.4. Here, the selected user has revealed up to level t of his passwords (the lines in the graphs represent different settings of t), whereas the other users have revealed all their passwords. Since the selected user can only rely on the operations on dummy blocks which are recorded in his UMB, the deniability is lower than that in the previous scenario. Nevertheless, DRSteg still manages to achieve high deniability.

Turning to the *worst-case* scenario where the adversary is aware of the creator of every block, Figure 6.4(c) shows the deniability levels achieved. In this scenario, deniability is derived solely from operations on the dummy data created by the user himself, which explains the much reduced deniability. Even so, DRSteg manages to keep the deniability above the configured threshold of $\alpha_T = 0.4$.

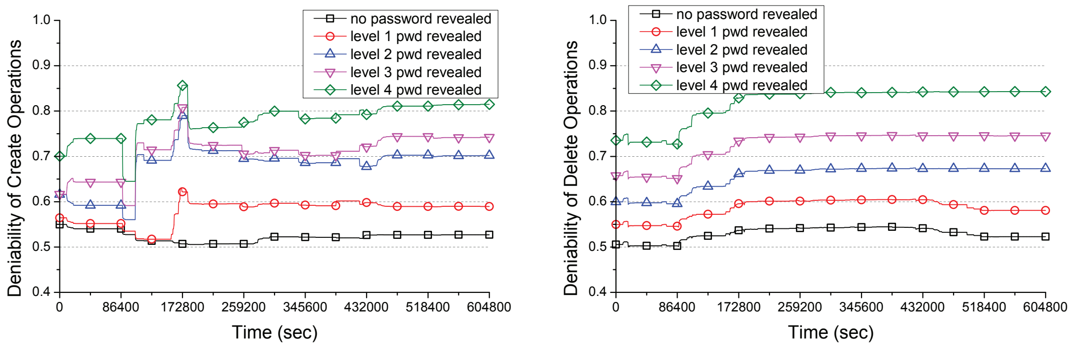
The deniability for the other 11 users are similar to the results in Figure 6.4



(a) Passive-adversary scenario



(b) Anonymous-channel scenario



(c) Worse-case scenario

Figure 6.4: Deniability of DRSteg under different scenarios

quantitatively and qualitatively. In particular, the lowest deniability observed for the worst-case attack scenario is 0.46. These results affirm the security property of our proposed DRSteg.

6.5.2 Implementation and Performance Evaluation

We have implemented DRSteg as a file system module in parallel with ext3 in Linux kernel 2.6, on the client machines which communicate with the shared storage through a server (see Figure 6.1). The client module manages the blocks in the shared storage automatically according to the password entered by the user. This includes creating new data and dummy blocks (and allocating shares to other owners), voting blocks for deallocation, etc. We explain below how the storage is organized by the system and benchmark the performance of DRSteg.

File system construction

In our DRSteg file system, the (remote) disk storage is partitioned into blocks of 1 Kbyte in size by default. A bitmap tracks the allocation status of the blocks: 1 corresponds to an allocated block and 0 a free block. An allocated block is either a dummy or a data block, both of which appear to contain random patterns.

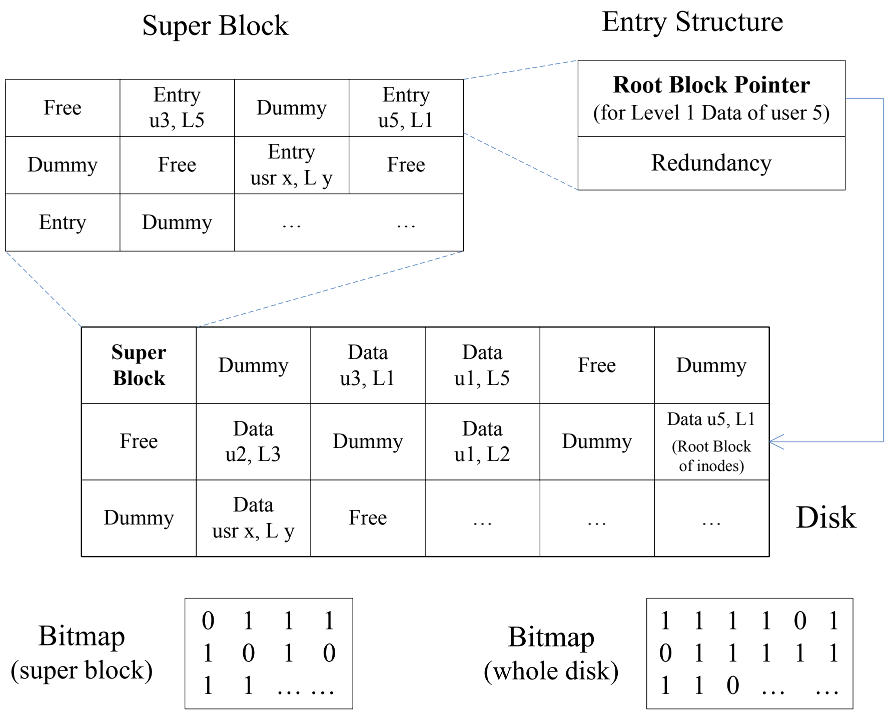


Figure 6.5: Block organization in DRSteg

To accelerate access to directories and files, DRSteg uses a designated storage area, called the *super block* (see Figure 6.5), to store inode structures so that they can be located efficiently. The super block is essentially a mini-DRSteg system for the addresses of inode roots, and is calved into fixed-size slots that are capable of holding one address each. A slot may be a free slot, a dummy slot, or may contain the encrypted address of an inode root (with redundancy so that it is distinguishable from random bits upon decryption). Each password level of a user is allocated one slot. Since the super block is expected to be only a few Kbytes in size, it can be scanned quickly to find the inode roots for each user. The super block has its own bitmap to track slot allocation, while it shares the same set of user share boxes and the global voting table with the main file system.

Performance Evaluation

The key parameters of the computing hardware for our experiments are listed in Table 6.4, while Table 6.5 summarizes the workload parameters and their default settings.

The first experiment is designed to study how well DRSteg performs. For comparison, we include StegCover, StegRand [19] and NSteg [85] as baselines. StegCover is configured with 20 cover files (the authors recommended 16 to 100 [19]). For StegRand, we use a replication factor of 4 to reduce the probability of data loss [79]. NSteg is set to populate 30% of the disk with dummy blocks during initialization. We also include two settings of the native Linux file system (ext3) in our tests. In the CleanDisk setting, data files are loaded into a freshly formatted native Linux partition, so that the files occupy contiguous disk blocks; with file operations translating to sequential I/Os, CleanDisk gives the best-case timings. In contrast, results of FragDisk are obtained with a well-used ext3 partition in which the free space is fragmented.

In the first experiment, we configure DRSteg with $\sigma_{u,l} = 0.25$, which produces a worst-case deniability of 0.2. For a given concurrency level, we generate file creation requests one after another for each user and measure the elapse time. Figure 6.6 shows the average write time for various file systems, with the number of concurrent users ranging from 1 to 32. Every performance result is averaged over 1000 observations.

Parameter	**Value**
CPU	Intel Duo Core 2.53GHz
RAM	2GB (1GB DDR2-667 x 2)
Hard Disk	SATA 7200rpm, 250 GB with 8MB cache

Table 6.4: Hardware parameters

Parameter	**Default Value**
Capacity of the test partition	40 Gbytes
Size of each disk block	1 Kbytes
Number of blocks for each file	1024
File access pattern	Interleaved

Table 6.5: Workload parameters

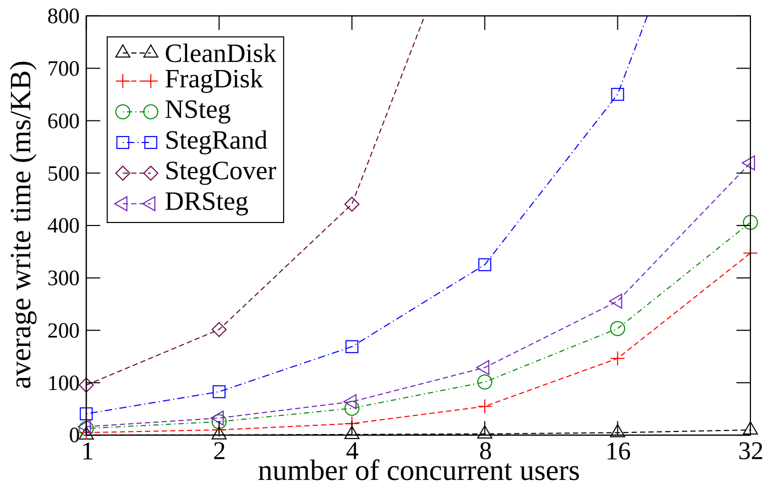


Figure 6.6: Performance compared with previous stegfs designs

The results show that StegCover is the worst performer; this is because each file operation translates into disk I/Os on several cover files. StegRand is also slow because it has to modify all the replicas. DRSteg and NSteg use a bitmap to track the status of disk blocks, so they can ensure data integrity with just one copy of each data file. Consequently, they are substantially faster than StegCover and StegRand. They are slower than FragDisk though, because they encrypt the protected files block by block and spread them across the disk, resulting in higher fragmentation.

Recall that DRSteg needs to write additional messages into the USBs during block creation and generates dummy operations dynamically. As the file creation requests in our experiment are issued one after another with no delay, the file system is fully loaded, leaving no idle period for DRSteg to schedule its dummy operations. Thus, the dummy operations add directly to the write times, and the observed timings represent the worst-case performance of DRSteg. For example, with $\sigma_{u,l} = 0.25$ it is roughly 30% slower than NSteg. This is the cost paid by DRSteg to achieve better security protection, compared to NSteg which is not able to relocate its dummy blocks.

In the second experiment, we investigate the performance of DRSteg under different load conditions. The load condition is determined by various factors, including the σ parameter that controls the amount of dummy operations, the concurrency level, and the activity level of each user. We model the activity level after a Poisson process with mean arrival rate of λ block operations per minute. The results are summarized in Figure 6.7, which plots the average write time against λ for several σ-concurrency combinations.

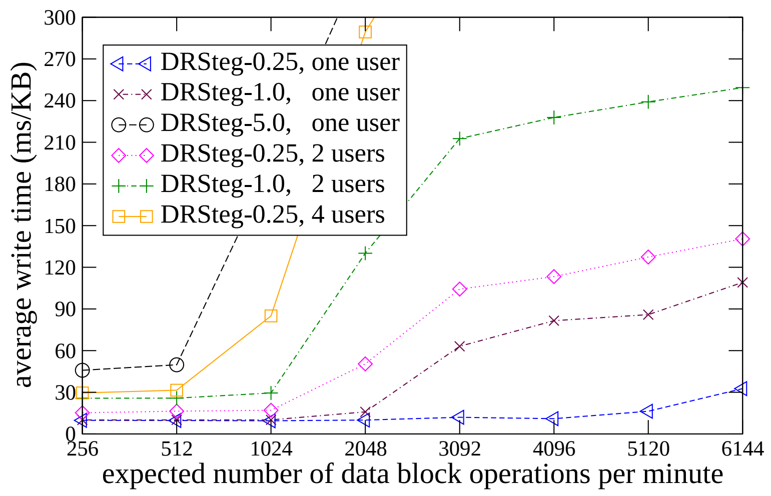


Figure 6.7: Trade-off between deniability and performance

We first consider the impact of λ. For every σ-concurrency combination, DRSteg's write time is short initially because there are ample lull periods during which dummy operations can be scheduled so as to reduce contention with data operations. Such opportunities diminish with increasing λ, leading to longer write times observed in the figure. Next, we compare the three σ-concurrency combinations with $\sigma = 0.25$. With the same σ and λ settings, raising the concurrency level introduces more contention between the data and dummy operations and lengthens the write time. Similarly, a bigger σ generates more dummy operations to cover the data operations, again resulting in longer write times.

In summary, our experiment demonstrates that DRSteg is capable of striking a wide range of trade-offs between deniability and system performance. If high deniability is required, the file system should be configured with enough resources to prevent it from becoming overloaded. On the other hand, to support a heavy workload, we could configure DRSteg for a lower deniability assurance.

6.6 Discussion

In this work, we address the threat to steganographic file systems (stegfs) that arises when the underlying storage is untrusted and shared by multiple users. In such systems, an adversary could obtain and analyze multiple snapshots of the storage content to deduce the existence of secret user data. To counter the threat, we introduce a Dummy-Relocatable Steganographic (DRSteg) file system that employs novel techniques to share and relocate dummy data at runtime. This enables users to surrender only some of their data, and attribute any unexplained changes across snapshots to dummy operations. The deniability enjoyed by users is configurable individually. DRSteg guarantees the integrity of the protected data, except where users voluntarily overwrite data under duress. A trace-driven simulation confirms the security of our scheme. Further experiments on a Linux prototype demonstrate that DRSteg is able to effectively trade off deniability with system performance.

Chapter 7

Dissertation Conclusion and Future Work

7.1 Summary of Contribution

This dissertation makes valuable contributions on utilizing diversity in software security and information hiding. Our first work systematically analyzed more than $6,000$ vulnerabilities published in the year of 2007, to validate the assumption that diverse software which provides similar functionalities is vulnerable only to different exploits. Our results show that the majority of the vulnerable application software products either do not have the same vulnerability, or cannot be compromised with the same exploit code. This work has been published in the Proceedings of the 6th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2009) [61].

In the second work, we proposed an intrusion detection scheme which builds on two diverse programs providing semantically-close functionalities to detect sophisticated attacks. Our model learns the underlying semantic correlation of the argument values in these programs, and consequently gains more accurate context information, which is effective in detecting attacks that manipulate erratic arguments. This work has been published in the Proceedings of the 7th International ICST

Conference on Security and Privacy in Communication Networks (SecureComm 2011) [63].

The third work investigated on the detailed iOS application permissions, while comparing to Android permissions. We also performed static analysis on over 1,000 pairs of applications that run on Android and iOS, the results of which reveal the detailed permission usage differences for Android and iOS third-party applications. This work has been submitted to a security conference at the time when this dissertation was submitted.

Finally, we introduced a Dummy-Relocatable Steganographic file system to provide deniability in multi-user environments where the adversary may have multiple snapshots of the disk content. The diverse ways of interpreting data in the storage allows a data owner to surrender only some data and attribute the unexplained changes across snapshots to the dummy data which are random bits. This work has been published in the Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC 2010) [62].

7.2 Future Direction

The four works presented in this dissertation mainly provide three future directions which can be followed with – intrusion detection, mobile security and steganographic file systems. Within these directions, mobile security is the most promising topic which has attracted growing attention in the research literature recently. For instance, there are 5 (out of 46) papers accepted in the 19th Annual Network & Distributed System Security Symposium (NDSS 2012), which are on mobile security. However, most of the existing mobile security research works focus on Android security, because it is an open platform which is relatively easy to analyze and implement with. In comparison to the literature on Android, there are relatively less studies on iOS platform.

One possible direction to take is to design and implement an information-flow tracking system for iOS devices (iPhone/iPad/iPod Touch) to monitor real-time privacy leakage on these portable devices. The advantage we have in investigating this research topic is that currently we already have static analysis tools on iOS, so it will be much easier to identify potential "harmful" third-party applications by first applying static analysis to large amount of applications before performing the dynamic analysis. Once we manage to design and implement such information-flow tracking system for iOS, the direct output would be hard evidences (behavior of real applications) that leak some private data to developers or advertising companies. Such observations will be newsworthy and are valuable to both research community and common iOS users.

Bibliography

[1] ActDroid on Android, https://market.android.com/details?id=actforex.trader ; ActPhone on iOS, http://itunes.apple.com/sg/app/actphone/id385112430 .

[2] HCPCS on Android, https://market.android.com/details?id=a1.com.HCPCSList ; HCPCS on iOS, http://itunes.apple.com/sg/app/hcpcs/id362345765 .

[3] eBuddy Messenger, iOS version: http://itunes.apple.com/sg/app/ebuddy-messenger/id320087242 ; Android version: https://market.android.com/details?id=com.ebuddy.android .

[4] Words With Friends Free, iOS version: http://itunes.apple.com/sg/app/words-with-friends-free/id321916506 ; Android version: https://market.android.com/details?id=com.zynga.words.

[5] AdWhirl Developer's Resources, https://www.adwhirl.com/home/dev .

[6] Flurry Product Updates, http://blog.flurry.com/updates/bid/33715/New-Flurry-SDK-Available-for-iPhone-OS-4-0-iOS .

[7] 3 Reasons iOS Has Better Security than Google Android. Pronet, June 2011. http://www.pronetadvertising.com/articles/3-reasons-ios-has-better-security-than-google-android.html .

[8] Android App Security Better than iPhones. GWL News, August 2010. http://www.geekwithlaptop.com/android-app-security-better-than-iphones .

[9] Android, iPhone security different but matched. CNET News, July 2010. http://news.cnet.com/8301-27080_3-20009362-245.html .

[10] Android Permission List. Android Manifest.permission API Level 8, http://developer.android.com/reference/android/Manifest.permission.html .

[11] eCryptfs, a POSIX-compliant enterprise-class stacked cryptographic filesystem for Linux. https://launchpad.net/ecryptfs .

[12] Encrypting File System in Windows XP and Windows Server 2003. http://www.microsoft.com/technet/prodtechnol/winxppro/deploy/cryptfs.mspx .

[13] IDApro, a multi-processor disassembler and debugger. Hex-Rays, http://www.hex-rays.com/products/ida/index.shtml .

[14] Smartphone Security Smackdown: iPhone vs. Android. InformationWeek, July 2011. http://www.informationweek.com/news/security/mobile/231000953 .

[15] TEMU and Vine. The BitBlaze Dynamic Analysis Component. http://bitblaze.cs.berkeley.edu .

[16] Trend Micro: Android much less secure than iPhone. Electronista News, January 2011. http://www.electronista.com/articles/11/01/11/trend.micro.warns.android.inherently.vulnerable/ .

[17] Why Android App Security Is Better Than for the iPhone. PCWorld News. August 2010. http://www.pcworld.com/businesscenter/article/202758/why_android_app_security_is_better_than_for_the_iphone.html .

[18] Ross J. Anderson and Eli Biham. Two practical and provably secure block ciphers: Bears and lion. In *Proceedings of the Third International Workshop on Fast Software Encryption*, pages 113–120, 1996.

[19] Ross J. Anderson, Roger M. Needham, and Adi Shamir. The steganographic file system. In *Proceedings of the 2nd International Workshop on Information Hiding*, pages 73–82, 1998.

[20] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS '07)*, pages 12–24, New York, NY, USA, 2007. ACM.

[21] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS '03)*, pages 281–289, New York, NY, USA, 2003. ACM.

[22] David Barrera, H. G üne ş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS '10)*, pages 73–84, New York, NY, USA, 2010. ACM.

[23] Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 48–62, 2006.

[24] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th conference on USENIX Security Symposium (SSYM'03)*, Berkeley, CA, USA, 2003. USENIX Association.

[25] Matt Blaze. A cryptographic file system for unix. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[26] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb 2012.

[27] Jesse Burns. Developing Secure Mobile Applications for Android. iSEC Partners, October 2008, http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf .

[28] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano. The design and implementation of a transparent cryptographic file system for unix. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 199–212, 2001.

[29] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Digest of 8th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 3–9, Tolouse, France, June 1978.

[30] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium*, 2005.

[31] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys '11)*, pages 239–252, New York, NY, USA, 2011. ACM.

[32] Lap chung Lam and Tzi cker Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 1–20, 2004.

[33] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems – A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, August 2006.

[34] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th conference on USENIX Security Symposium*, 2006.

[35] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th international conference on Information security (ISC'10)*, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[36] Rohit Dhamankar. SANS Top-20 Security Risks, 2007. http://www.sans.org/top20/2007/ .

[37] Claudia Diaz, Carmela Troncoso, and Bart Preneel. A framework for the analysis of mix-based steganographic file systems. In *Proceedings of the 13th European Symposium on Research in Computer Security*, pages 428–445, 2008.

[38] Josep Domingo-Ferrer and Maria Bras-Amorós. A shared steganographic file system with error correction. In *Proceedings of the 5th International Conference on Modeling Decisions for Artificial Intelligence*, pages 227–238, 2008.

[39] Jake Edge. Remote file inclusion vulnerabilities. Octobor 2006. http://lwn.net/Articles/203904/ .

[40] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '11)*, San Diego, CA, February 2011.

[41] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10)*, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[42] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, Berkeley, CA, USA, 2011. USENIX Association.

[43] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *IEEE Security and Privacy*, 7:50–57, January 2009.

[44] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS '11)*, pages 627–638, New York, NY, USA, 2011. ACM.

[45] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: attacks and defenses. In *Proceedings of the 20th USENIX conference on Security*, Berkeley, CA, USA, 2011. USENIX Association.

[46] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.

[47] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.

[48] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications. Technical report, University of Maryland, 2009.

[49] Gordon Lyon Fyodor. Remote os detection via tcp/ip stack fingerprinting. Technical report, INSECURE.ORG, October 1998.

[50] Debin Gao, Michael K. Reiter, and Dawn Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM conference on Computer and Communications Security*, pages 318–329, 2004.

[51] Debin Gao, Michael K. Reiter, and Dawn Song. Behavioral distance for intrusion detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, pages 63–81, 2005.

[52] Debin Gao, Michael K. Reiter, and Dawn Song. Behavioral distance measurement using hidden markov models. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection*, pages 19–40, 2006.

[53] Debin Gao, Michael K. Reiter, and Dawn Song. Beyond output voting: Detecting compromised replicas using HMM-based behavioral distance. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, July 2008.

[54] Ilir Gashi and Peter Popov. Fault tolerance via diversity for off-the-shelf products: A study with sql database servers. *IEEE Transactions on Dependable Secure Computing*, 4(4):280–294, 2007. Member-Lorenzo Strigini.

[55] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. P. Pfleeger, J. S. Quarterman, and B. Schneier. Cyberinsecurity: The cost of monopoly. Technical report, CCIA, 2003.

[56] Anup K. Ghosh and Aaron Schwartzbard. A study in using neural networks for anomaly and misuse detection. In *Proceedings of the 8th conference on USENIX Security Symposium*, 1999.

[57] Charles Giefer and Julie Letchner. Mojitos: A distributed steganographic file system. Technical report, Univerisity of Washington, 2004.

[58] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium*, 2004.

[59] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb 2012.

[60] Frank Graf and Stephen D. Wolthusen. A capability-based transparent cryptographic file system. In *Proceedings of the 2005 International Conference on Cyberworlds*, pages 101–108, 2005.

[61] Jin Han, Debin Gao, and Robert H. Deng. On the effectiveness of software diversity: A systematic study on real-world vulnerabilities. In *Proceedings of the Detection of Intrusions and Malware and Vulnerability Assessment*, pages 127–146, July 2009.

[62] Jin Han, Meng Pan, Debin Gao, and HweeHwa Pang. A multi-user steganographic file system on untrusted shared storage. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*, pages 317–326, New York, NY, USA, December 2010. ACM.

[63] Jin Han, Qiang Yan, Debin Gao, and Robert H. Deng. On detection of erratic arguments. In *Proceedings of the 7th International ICST Conference on Security and Privacy in Communication Networks (SecureComm '11)*, September 2011.

[64] Steven Hand and Timothy Roscoe. Mnemosyne: Peer-to-peer steganographic storage. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, pages 130–140, 2002.

[65] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS '11)*, pages 639–652, New York, NY, USA, 2011. ACM.

[66] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing Cross Site Request Forgery Attacks. In *IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm)*, 2006.

[67] J. Just, J. Reynolds, L. Clough, M. Danforth, K. Levitt, R. Maglich, and J. Rowe. Learning unknown attacks - A start. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID '02)*, 2002.

[68] J. Just, J. Reynolds, L. Clough, M. Danforth, K. Levitt, R. Maglich, and J. Rowe. Learning unknown attacks - A start. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection*, 2002.

[69] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS '03)*, pages 272–280, New York, NY, USA, 2003. ACM.

[70] Christine Kinealy. *This Great Calamity: The Irish Famine 1845-52*. Gill & Macmillan, 1995.

[71] David Kravets. Jailbreaking iPhone Legal, U.S. Government Says. ABCNews, http://abcnews.go.com/Technology/story?id=11254253 .

[72] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. On the detection of anomalous system call arguments. In *European Symposium on Research in Computer Security*, 2003.

[73] Leslie Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11), 1981.

[74] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10, 1966.

[75] Richard C. Linger. Systematic generation of stochastic diversity as an intrusion barrier in survivable systems software. In *Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences-Volume 3 (HICSS '99)*, Washington, DC, USA, 1999. IEEE Computer Society.

[76] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Asian Symposium on Programming Languages and Systems*, November 2005.

[77] Hiroshi Lockheimer. Android and Security. Google Mobile Blog, Feb 02, 2012. http://googlemobile.blogspot.com/2012/02/android-and-security.html .

[78] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 7:381–395, 2010.

[79] Andrew D. McDonald and Markus G. Kuhn. StegFS: A steganographic file system for Linux. In *Proceedings of the 3rd International Workshop on Information Hiding*, pages 462–477, 2000.

[80] C. C. Michael and Anup Ghosh. Simple, state-based approaches to program-based anomaly detection. *ACM Transactions on Information and System Security (TISSEC)*, 5(3):203–237, 2002.

[81] Nemo. The Objective-C Runtime: Understanding and Abusing. Phrack, Volume 4, Issue 66, http://www.phrack.org/issues.html?issue=66&id=4 .

[82] Adam J. O'Donnell and Harish Sethu. On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM conference on Computer and communications security (CCS '04)*, pages 121–131, New York, NY, USA, 2004. ACM.

[83] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference (ACSAC '09)*, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society.

[84] Gabor Paller. Dedexer. http://dedexer.sourceforge.net/ .

[85] HweeHwa Pang, Kian-Lee Tan, and Xuan Zhou. StegFS: A steganographic file system. In *Proceedings of the 19th International Conference on Data Engineering*, pages 657–668, 2003.

[86] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[87] J. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich. The design and implementation of an intrusion tolerant system. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN '02)*, 2002.

[88] J. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich. The design and implementation of an intrusion tolerant system. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN)*, 2002.

[89] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 199–212, 2009.

[90] Stephen A. Ross, Randolph W. Westerfield, and Bradford D. Jordan. *Fundamentals of corporate finance*. Irwin/McGraw-Hill, 8th. edition edition, 2008.

[91] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[92] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18:613–620, November 1975.

[93] Jason Sawin and Atanas Rountev. Improving static resolution of dynamic class loading in java using dynamically gathered environment information. *Automated Software Engineering*, 16:357–381, June 2009.

[94] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[95] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[96] Nicolas Seriot. iPhone Privacy. BlackHat Technical Security Conference: DC 2010, http://seriot.ch/resources/talks_papers/iPhonePrivacy.pdf .

[97] Amit Singh. *Mac OS X Internals: A Systems Approach*. Addison-Wesley, 2006.

[98] Charlie Sorrel. Apple Approves, Pulls Flashlight App with Hidden Tethering Mode. Wired. July, 2010. http://www.wired.com/gadgetlab/2010/07/apple-approves-pulls-flashlight-app-with-hidden-tethering-mode .

[99] Mark Stamp. Risks of monoculture. *Communications of the ACM*, 47(3), 2004.

[100] G. Tandon and P. Chan. Learning rules from system call arguments and sequences for anomaly detection. In *ICDM Workshop on Data Mining for Computer Security (DMSEC '03)*, pages 20–29, 2003.

[101] E. Totel, F. Majorczyk, and L. Me. COTS diversity based intrusion detection and application to web servers. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID '05)*, 2005.

[102] E. Totel, F. Majorczyk, and L. Me. COTS diversity based intrusion detection and application to web servers. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, 2005.

[103] Carmela Troncoso, Claudia Diaz, Orr Dunkelman, and Bart Preneel. Traffic analysis attacks on a continuously-observable steganographic file system. In *Proceedings of the 9th Information Hiding*, pages 220–236, 2008.

[104] Chris Trowbridge. An overview of remote operating system fingerprinting. Technical report, The SANS Institute, July 2003.

[105] Timothy Vidas, Nicolas Christin, and Lorrie Cranor. Curbing Android permission creep. In *Proceedings of the Web 2.0 Security and Privacy 2011 workshop (W2SP 2011)*, Oakland, CA, May 2011.

[106] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS '07)*, February 2007.

[107] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[108] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21:168–173, January 1974.

[109] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering (ICSE '08)*, pages 171–180, New York, NY, USA, 2008. ACM.

[110] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering (ICSE '81)*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[111] Charles P. Wright, Michael C. Martino, and Erez Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 197–210, 2003.

[112] Yongguang Zhang, Harrick Vin, Lorenzo Alvisi, Wenke Lee, and Son K. Dao. Heterogeneous networking: a new survivability paradigm. In *Proceedings of the 2001 workshop on New security paradigms (NSPW '01)*, pages 33–39, New York, NY, USA, 2001. ACM.

[113] Xuan Zhou, HweeHwa Pang, and Kian-Lee Tan. Hiding data accesses in steganographic file system. In *Proceedings of the 20th International Conference on Data Engineering*, pages 572–583, 2004.

[114] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb 2012.

Glossary

Commercial Off-The-Shelf (COTS) Commercially Off-The-Shelf is a Federal Acquisition Regulation term defining a nondevelopmental item of supply that is both commercial and sold in substantial quantities in the commercial marketplace, and that can be procured or utilized under government contract in the same precise form as available to the general public.

Common Vulnerabilities and Exposures (CVE) CVE is a list or dictionary that provides common names for publicly known information security vulnerabilities and exposures. http://cve.mitre.org/

Cross-site Request Forgery (CSRF) Cross-site Request Forgery is a type of malicious exploit of a website whereby unauthorized commands are transmitted from a user that the website trusts.

Cross-site Scripting (XSS) Cross-site scripting is a type of computer insecurity vulnerability typically found in Web applications (such as web browsers through breaches of browser security) that enables attackers to inject client-side script into Web pages viewed by other users.

Dalvik Executable (DEX) Dalvik is the process virtual machine (VM) in Google's Android operating system. Programs on Android are commonly written in a dialect of Java and compiled to bytecode, which are then converted from Java Virtual Machine-compatible .class files to Dalvik-compatible .dex (Dalvik Executable) files before installation on Android devices.

Denial of Service (DoS) A denial-of-service attack (DoS attack) is an attempt to make a computer or network resource unavailable to its intended users.

Dummy-Relocatable Steganographic (DRSteg) This is a multi-user steganographic file system we proposed for shared storage systems, which is able to provide plausible deniability of secret data in a multi-user environment in which the adversary could obtain multiple snapshots of the storage content.

Erratic Arguments Erratic arguments are function arguments which have the erratic property – in a particular execution context, only a subset of the argument values (possibly one) is legitimate while other values could potentially be malicious.

Executable and Linkable Format (ELF) In computing, the Executable and Linkable Format (ELF, formerly called Extensible Linking Format) is a common standard file format for executables, object code, shared libraries, and core dumps.

GNU C Library (glibc) The GNU C Library, commonly known as glibc, is the C standard library released by the GNU Project, which was originally written by the Free Software Foundation (FSF) for the GNU operating system.

GNU Debugger (GDB) The GNU Debugger, usually called just GDB and named gdb as an executable file, is the standard debugger for the GNU software system. It is a portable debugger that runs on many Unix-like systems and works for many programming languages, including Ada, C, C++, Objective-C, Free Pascal, Fortran, Java[1] and partially others.

Global Voting Table (GVT) In DRSteg, the global voting table records the votes surrendered by users, which is maintained in clear text in the storage.

Host-based Intrusion Detection System (HIDS) A host-based intrusion detection system (HIDS) is an intrusion detection system that monitors and analyzes the internals of a computing system and/or the network packets on its network interfaces.

Intrusion Detection System (IDS) An intrusion detection system (IDS) is a device or software application that monitors network and/or system activities for malicious activities or policy violations and produces reports to a Management Station.

National Vulnerability Database (NVD) The National Vulnerability Database is the U.S. government repository of standards based vulnerability management data represented using the Security Content Automation Protocol (SCAP), http://nvd.nist.gov/ .

Original Equipment Manufacturer (OEM) An original equipment manufacturer (OEM) manufactures products or components that are purchased by a company and retailed under that purchasing company's brand name.

Portable Executable (PE) The Portable Executable (PE) format is a file format for executables, object code and DLLs, used in 32-bit and 64-bit versions of Windows operating systems.

Private Information Retrieval (PIR) In cryptography, a private information retrieval (PIR) protocol allows a user to retrieve an item from a server in possession of a database without revealing which item he is retrieving.

Remote File Inclusion (RFI) Remote File Inclusion (RFI) is a type of vulnerability most often found on websites, which allows an attacker to include a remote file, usually through a script on the web server.

Software Substitutes Software substitutes refer to two software products that provide the same service.

Steganographic File Systems (Stegfs) A steganographic file system is a storage mechanism designed to give its users a high level protection against being compelled to disclose its contents. It delivers the secret content to any user who knows the password; but an attacker who does not possess this information can nether guess it, nor gain any information about whether the content is present.

System Call In computing, a system call is how a program requests a service from an operating system's kernel, which provides the interface between a process and the underlying operating system.

Universally Unique Identifier (UUID) A UUID is an identifier standard used in software construction, standardized by the Open Software Foundation as part of the Distributed Computing Environment.

User Share Box (USB) In DRSteg, each user has a user shared box, which is used to track the set of blocks on which the corresponding user holds a share. Shares in the user shared boxes are encrypted with the user's public key.