

# 國立交通大學

## 網路工程研究所

### 碩 士 論 文



利用行為相似性偵測Android平台惡意應用程式

Identifying Malicious Applications by Behavioral Similarity on  
Android Platforms

研 究 生：陳健宏

指導教授：林盈達 教授

中 華 民 國 一 百 零 一 年 六 月

利用行為相似性偵測 Android 平台惡意應用程式

Identifying Malicious Applications by Behavioral Similarity on Android  
Platforms

研 究 生：陳健宏

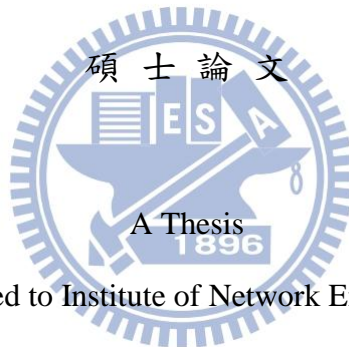
Student：Chien-Hung Chen

指導教授：林盈達

Advisor：Dr. Ying-Dar Lin

國 立 交 通 大 學

網 路 工 程 研 究 所



Submitted to Institute of Network Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2012

Hsinchu, Taiwan

中華民國一百零一年六月

# 利用行為相似性偵測 Android 平台惡意應用程式

學生：陳健宏

指導教授：林盈達

國立交通大學網路工程研究所

## 摘要

隨著行動裝置計算能力的提升與盛行，在手機上提供的應用程式越趨多樣化，但卻同時成為了系統安全上新的攻擊目標。對於目前流行的 Android 系統平台，攻擊者可以透過再包裝與混淆的技術，將惡意程式碼同時隱藏到多個看似一般的應用程式來進行散佈，使得 Android 平台上的惡意程式偵測與分析工作更加的費時和困難。然而，被打包惡意程式碼的應用程式即使有了不同的外表，但同樣的惡意程式碼仍然會產生出同樣的行為，因此我們提出了一套利用系統呼叫序列來進行應用程式的行為偵測方法，此方法能夠從多執行緒的惡意程式所產生的系統呼叫序列中找出共同子序列，並且利用貝氏機率模型來過濾出有較高機率出現在惡意應用程式，但較低機率在正常應用程式執行時出現的系統呼叫序列。最後我們能夠利用這些抽取出來的系統呼叫序列，對待檢測的應用程式所執行的系統呼叫序列中進行掃描。我們使用五個種類的被打包惡意程式碼的應用程式與一百正常的應用程式來進行準確率的評估，在所有的種類裡面，我們的方法可以得到 97.6% 的高準確率，在所有 25 個被檢測的惡意應用程式中，僅有一個沒有被辨識出來。

**關鍵字：**惡意應用程式，行為偵測，系統呼叫，Android

# Identifying Malicious Applications by Behavioral Similarity on Android Platforms

Student: Chien-Hung Chen

Advisor: Dr. Ying-Dar Lin

Department of Computer and Information Science

National Chiao Tung University

## Abstract

As mobile applications become popular, they become the new target of attackers. For Android platforms, adversaries can easily repackage the malicious code into the different benign applications for distribution. The work of detecting and analyzing the malicious application becomes a challenge of Android. Though, the repackaged applications have different outward appearances, the same malicious behaviors still appear during runtime. Therefore, we propose a behavior-based detection mechanism based on system call sequences. We extract the common system call subsequences of malicious applications and propose a comparison approach to deal with multiple threads produced by the applications. We also utilize the Bayes probability model to filter subsequences which have lower probability of appearance in the repackaged applications. Finally, we can detect repackaged applications by those extracted subsequences. In our experiment, we use five different types of repackaged applications and 100 benign applications to evaluate the accuracy rate. The detection result demonstrates that our approach has 97.6% high accuracy. We evaluate 25 repackaged applications and miss only one evaluated target.

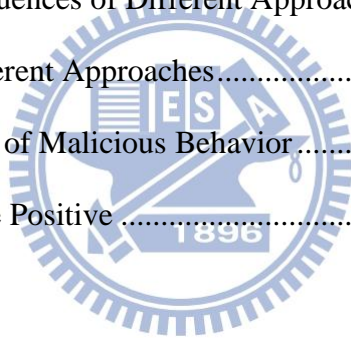
**Keywords:** malicious applications, behavior-based detection, system call, Android

# Contents

|   |    |
|---|----|
| List of Figures .....                                       | V  |
| List of Tables .....  | VI |
| Chapter 1. Introduction .....                               | 1  |
| Chapter 2. Background .....                                 | 4  |
| 2.1 Malware Differences between Desktops and Handhelds..... | 4  |
| 2.2 Related Works on Desktop Environment .....              | 5  |
| 2.3 Related Works on Android Environment .....              | 6  |
| Chapter 3. Problem Statement .....                          | 9  |
| 3.1 System Call Sequence Detection .....                    | 9  |
| 3.2 Problem Description .....                               | 11 |
| Chapter 4. Approach .....                                   | 12 |
| 4.1 Approach Overview .....                                 | 12 |
| 4.2 Extraction of System Call Subsequence .....             | 14 |
| 4.3 Sequence Evaluation with Probability of Appearing ..... | 16 |
| 4.4 Environment Implementation .....                        | 17 |
| Chapter 5. Evaluation.....                                  | 19 |
| 5.1 Experiment Environment .....                            | 19 |
| 5.2 System Call Sequences Distribution.....                 | 20 |
| 5.3 Accuracy of Detection .....                             | 22 |
| 5.4 Relationship of Training Samples and Sequences .....    | 24 |
| 5.5 Fixed-Length and Longest Common Subsequences.....       | 26 |
| 5.6 Case Study of Subsequences.....                         | 28 |
| Chapter 6. Conclusions and Future Works .....               | 31 |
| References.....   | 33 |

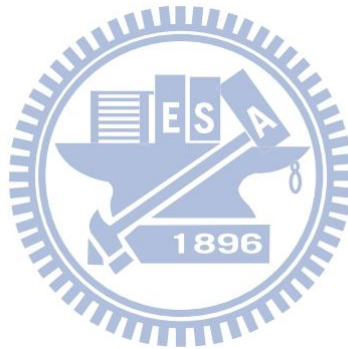
## List of Figures

|  |    |
|--|----|
| Figure 1. Flowchart of System Call Sequence Processing .....               | 12 |
| Figure 2. Example of Thread Trees .....                                    | 15 |
| Figure 3. Zygote Mode .....  | 17 |
| Figure 4. Experiment Environment.....                                      | 19 |
| Figure 5. Probability Distribution of Subsequences .....                   | 21 |
| Figure 6. Accuracy of Detection.....                                       | 23 |
| Figure 7. True Positive and True Negative of Different Training Sets ..... | 24 |
| Figure 8. Sequences of Different Number of Repackaged Applications.....    | 25 |
| Figure 9. Number of Subsequences of Different Approaches .....             | 27 |
| Figure 10. Accuracy of Different Approaches.....                           | 27 |
| Figure 11. The Subsequence of Malicious Behavior.....                      | 28 |
| Figure 12. The Case of False Positive .....                                | 29 |



## List of Tables

|   |    |
|---|----|
| Table 1. Related Works of Malicious Android Applications Detection .....  | 6  |
| Table 2. Defincation of Symbols .....                                     | 10 |
| Table 3. Number of Training Samples and Sequences .....                   | 20 |
| Table 4. The Number of Subsequences of 100% Probability .....             | 22 |
| Table 5. Detection Result.....  | 22 |
| Table 6. True Positive and True Negative .....                            | 23 |
| Table 7. Change of True Positive and True Negative.....                   | 26 |
| Table 8. The Relationship of Number of Samples and Sequences (Kmin) ..... | 26 |



## Chapter 1 Introduction

With the rapid increase of wireless mobile networks, the smart handheld have become popular and bring us new applications and integration with existing online services. Over the past decade, many operating systems have been developed to provide a comprehensive infrastructure for these devices. Among them, the Android platform, developed by Google, is one of the most popular mobile operating systems because of its open source. A large community of developers has designed many new applications for Android. Android provides an open environment called “Android Market” from which users can download applications. However, this scenario suffers from some weakness. An adversary can distribute their malicious applications that can launch various security attacks including escalating privilege, monitoring users’ behaviors, and compromising the sensitive information at the end devices [1-3]. Especially, different from the traditional attacking mechanisms on the desktop, an adversary can repackage the malware into the different benign applications to deceive users installing the affected applications on mobile devices. Hence, detecting malware from various applications becomes a challenge of Android [4].

### **Signature-Based Detection vs. Behavior-Based Detection**

Typically, the malware detection mechanisms can be categorized into two types. The first type is the signature-based detection mechanisms. We search the binary image for the suspicious malicious patterns and then extract the unique signature for each of them. Then we can identify malware by matching the binary image of these applications with a database of signatures. Although signature-based detection mechanisms are simple, efficient, and effective for the known existing malware, they suffer from the drawback of not being able to detect the previously unknown malware. To overcome the limitation, the behavior-based detection mechanisms have been



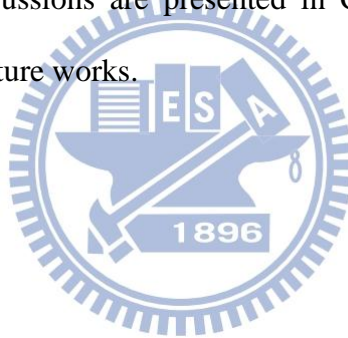
therefore proposed. These mechanisms attempt to observe suspicious runtime application behaviors. Independent of the binary image of malware, the behavior-based mechanisms could significantly increase the detection rate for unknown and obfuscated malware. Hence, the behavior-based mechanisms are more suitable to detect the repackaged malicious applications.

To identify Android malicious applications, many detection mechanisms have been proposed. For example, TaintDroid [5] tracks the information flow in the system to identify the sensitive information leakage from applications, but it only detects the malware that tries to obtain the sensitive data. In addition, ScanDroid [6] and Kirin [7] attempt to verify the permission and the information flow in program codes before users install applications. However, only codes with violated permissions in codes can be identified as the malicious applications. Others detection mechanisms utilize the kernel-based system calls to analyze the behaviors of malware. AAsandbox[8] counts all system calls used in Android system to identify the anomaly. Isohara, et al. [9] write a set of regular expression rules to matching in the repackaged applications. CrowDroid [10] utilizes system calls to clustering benign and malicious applications. But it suffers from a drawback of needing the original application to compare with the repackaged one. According to our observation, none of them can be utilized to detect all the repackaged applications efficiently.

To solve the problem of repackaged applications, we propose a novel behavior-based detection version which relies on system call sequences. The key idea is to observe the system call sequences that are trigger by applications, i.e., although a malicious code can camouflage into a benign application, the malicious behavior still appears in the system call sequences. Since the system call interface is embedded in the kernel space at the low layer of Android architecture, processes are hard to hide triggered system calls. By applying this concept, our approach extract common

system call sequences in a set of same type but are repackaged malware by longest common substring algorithm to find the longest sequence patterns. To filter out the benign behaviors, we also employ the Bayes probabilistic model for evaluating the appearance probability of extracted system call sequences to find the significant malicious sequences. Through this approach, the repackaged malware can be detected with high accuracy and the false positive rate can be reduced significantly.

The rest of this thesis is organized as follows. In Chapter 2, we give a brief survey of related works which describes both desktop and Android environments. In Chapter 3 and Chapter 4, we describe the details of the proposed mechanism on processing system call sequences and its implementation, respectively. Some experiment results and discussions are presented in Chapter 5. Finally, Chapter 6 gives the conclusions and future works.



## Chapter 2 Background

In this chapter, we describe the issue on how to detect repackaged malicious applications on the Android environment. Malware has been an issue on desktops many years ago, and many works have been proposed to detect malware. In recent years, malware has propagated to handhelds. Herein, we give a brief overview about malware behavior differences between desktops and handhelds. Some related works, such as malware detection by system calls on desktops and Android platforms, are also discussed.

### 2.1 Malware Differences between Desktops and Handhelds

Compared with desktops, malware detection in handhelds suffers from some limitations. Among them, one of the biggest problems is the power consumption. Unlike powerful desktops, handhelds are only supported with finite energy from batteries. This situation constrains us from exhausting the energy to detect malware. The other problem is the limitation of the kernel of the operating system. For instance, Android is built on the Linux kernel. Android applications are executed on the virtual machine, Dalvik, which isolates applications by marking each application run as its own user. Hence, it is hard to observe or to block the malicious application from our programs without the root privilege. Due to these significant drawbacks, traditional desktop based malware detection mechanisms cannot be performed on the Android environment.

Many changes in malware's purpose and behavior also lead us to improve detection methods for the new circumstance. Basically, attack methods can be categorized into two types: server-side attack and client-side attack. In a server-side attack, attackers directly aim to the potential vulnerabilities in a server which exposes its services to clients. Most active malware, like worms or some kinds of bots, usually

utilizes this avenue for spreading. Fortunately, handhelds do not always access the Internet services, a server-side attack cannot mount very well on handhelds. In contrast, client-side attacks target vulnerabilities in client applications which interact with the malicious server. Hence, attacker can construct a phishing site or embed the malicious code into an ordinary server and infect applications of a client when a client accesses the forgery information. For example, spyware and trojan horses are the two typical instances to infect systems through this scenario. In addition, an attacker can easily obtain the privacy information through the applications that have been infected by spyware and trojan horses, i.e., the secret information can be obtained through the SMS messages or phone calls. Compared with detecting worms and bots, it is a thorny problem to detect spyware and trojan horses on handhelds.

## **2.2 Related Works on Desktop Environment**

In the traditional desktop environment, signature-based detection is the most popular technique used in anti-virus and intrusion detection systems. Unfortunately, signature-based methods cannot detect the zero day malware. To overcome this security flaw, many behavior-based methods were proposed. Among them, system call is one of the popular techniques used to deeply monitor program behaviors. Forrest, et al. [11] records the normal system call behaviors of a specific program into a database. As the compromised program executing malicious code paths, the anomalous system call sequence patterns are detected because the system call patterns do not exist in the database. After that, many researches have extended the system call approach and apply system call sequences into different models, such as hidden Markov model [12], finite state automata [13], or Bayes model [14], to improve the detection efficiency. Unfortunately, those proposed methods need to collect all the behaviors in the specified programs. Collecting all behaviors in a large system, such as the Android environment, is computational infeasible.

Christodorescu, et al. [15] apply malware behavior sequences into dependence graphs and extract the subgraphs which do not appear in benign program for detecting malware variants. Rozenberg, et al. [16] also collect fix length short system call sequences from malware to detect unknown malware. In this work, we do the similar but more force on multi threads processing and discover more precise system call sequences to achieve the high accuracy for detecting repackaged applications.

### 2.3 Related Works on Android Environment

Table 1. Related Works of Malicious Android Applications Detection

| Analysis Approach | Paper Works [Reference #]                 | Factor                         | Target                   |
|-------------------|---|--------------------------------|--------------------------|
| Static            | <i>ScanDroid</i> [6]                      | Program Code                   | Application Verification |
|                   | <i>Kirin</i> [7]                          | Permission                     | Application Verification |
|                   | <i>DroidMOSS</i> [17]                     | Code Instructions              | Repackaged Applications  |
| Dynamic           | <i>TaintDroid</i> [5]                     | Data flow                      | Data Leaking             |
|                   | <i>Kernel-based behavior analysis</i> [9] | System Call Name and Parameter | Anomaly Behaviors        |
|                   | <i>AASandbox</i> [8]                      | Amount of System Call          | Anomaly Behaviors        |
|                   | <i>CrowDroid</i> [10]                     | Amount of System Call          | Repackaged Applications  |
|                   | <i>This Work</i>                          | System Call Sequences          | Repackaged Applications  |

Referring to Table 1, in the Android environment, many researchers focus on how to verify Android applications. For instance, Enck, et al. [5] developed a novel system called “TaintDroid”. The key idea is that they track the sensitive data by labeling data in the memory and detecting the runtime privacy leaking behavior of applications. Although TaintDroid system is simple and efficient, it only detects the malware that attempts to obtain the sensitive data. For other types of repackaged applications, TaintDroid cannot detect them well.

Kirin [6] utilizes a set of permission rules to block before installing applications which require unsafe permission combination. ScanDroid [7] extracts security specifications from manifest for automatically checking data flows in the application

code. They can identify malicious applications which violate permissions in codes.

AAsandbox [8] applies both static and dynamic approaches to analyze applications. This means that it decompiles the installation file and then searches for the suspicious patterns in the decompiled codes. During the application runtime, AAsandbox counts all system calls used in Android system. However, AAsandbox only sworks with specific malware, i.e., it cannot detect multiple types of malware. This is because its dynamic approach only detects abnormal system call usage.

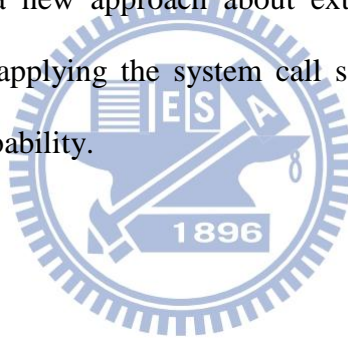
The method proposed by T. Isohara, et al. [9] collects the runtime information of applications through system calls. They search some keywords in the names and parameters of system calls to generate a set of regular expression rules. In addition, they utilize the generated rules to search suspicious system calls and detect Android malware. However, the malicious applications still have a chance to escape from being detected by encapsulating or obfuscating the keyword in parameters.

CrowDroid [10] is another typical method to evaluate applications by using system calls. After the system collects all the system calls used from a set of user devices during the application runtime, it adopts K-means clustering algorithm to classify the collected data into two groups, the benign group and the malicious group. And the malicious group can be used to identify the specified user who is running the repackaged application. CrowDroid needs a set of users to execute the same original application or the repackaged application, nevertheless, find the original application of all repackaged malicious applications is inefficient and impossible in Android environment.

DroidMOSS [17] is a system which aims to discover repackaged applications in the third-party marketplace. DroidMOSS calculates the similarity scores by comparing the author information and code instruction hash between the original applications and the repackaged applications in the different marketplaces. Finally, it

finds the repackaged applications in the third-party marketplaces since the repackaged application has a different appearance with the original application provided by the official market. However, DroidMOSS has two weaknesses. Let us consider the following scenario, i.e., once the malware is distributed both in the third-party marketplace and in the official market, DroidMOSS cannot distinguish the malware from the repackaged applications. The other is that not all of the original applications can be obtained from the official market.

As a result of these incomplete methods, our proposed method aims to extract longest system call sequences patterns from the same type of repackaged applications and utilize the patterns to detect repackaged malware without the original applications. Our contribution includes a new approach about extracting multi-thread common system call sequences and applying the system call sequence to detect repackaged applications with Bayes probability.



## Chapter 3 Problem Statement

To detect Android repackaged applications, we therefore proposed a behavior-based detection mechanism. In the beginning of this chapter, we describe the key idea that is adopted in the proposed mechanism. And then, we give the problem statement in subsection 3.2.

### 3.1 System Call Sequence Detection

To disseminate malware to users, attackers usually embed their malicious codes into the normal applications, and then publish these repackaged applications to the official Android market or the third-party market. This dissemination path is simple, efficient, and effective because the repackaged applications are like benign applications to fool users into obtaining the secret information. In order to detect the malicious repackaged applications, the proposed mechanism observes the behaviors during the execution period of repackaged applications rather than the outward appearance of repackaged applications. The key idea is that, even attackers can embed the malicious codes into various applications, the behaviors of executing applications are consistent with the barely malicious code instructions. Hence, the proposed mechanism observes the execution behaviors of application and then extracts the common behavior patterns to distinguish malware.

For collecting application behaviors, we utilize the kernel-based system calls. System calls provide interaction between processes and the operating system kernel for receiving service or resource requests. All service and resource requests can be observed by monitoring the system call interface, and the sequential requests form up a sequence of system calls. The observed system call sequence can be regarded as the request behaviors during the application executing. There are other ways to collect call sequences, such as function calls of higher layers also provide the similar



functionality to form up call sequences, but it would lead ambiguous because of its multi-interfaces. Let us consider a scenario. For example, Android applications can request a service of application framework through Android API or directly access the libraries through Java Native Interface to achieve the same functionality. The application can produce the same behavior but with different call interfaces, and the call sequence will be more complicated if we need to integrate more than one interfaces. It is also possible that the program gets the root permission and executes at lower level than Android API and JNI. Hence, the proposed mechanism used system calls not only simply indicate the running behavior of applications, but also record the behavior of malicious codes completely than using high level function call.

Therefore, for detecting malicious repackaged applications, we initially collect a set of same type repackaged applications  $M$ . Let  $M_i$  denotes the  $i$ -th repackaged application in the set  $M$ . Then, for each  $M_i$ , we record the corresponding  $S_i$  as a set of system call sequences. After that, we can extract the common system call subsequences  $S_{sign}$  from the set  $\{S_1, S_2, S_3, \dots, S_{|M|}\}$  to derive the common behavior patterns. Finally, for those undiscovered repackaged applications  $E_{mal}$  with the same malicious codes, we can detect them from a set of applications to be inspected  $E$  by  $S_{sign}$ .

Table 2. Definition of Symbols

|            |   |
|------------|---|
| $M$        | A set of same type repackaged applications.   |
| $M_i$      | The $i$ -th repackaged application in $M$ .   |
| $S_i$      | The set of recorded system call sequences from $M_i$ .  |
| $S_{sign}$ | The set of significant common system call subsequences which are extracted from the set $\{S_1, S_2, S_3, \dots, S_{ M }\}$ . |
| $E$        | A set of applications to be inspected.  |
| $E_{mal}$  | The set of true repackaged applications in $E$ .  |

### 3.2 Problem Description

The detail problem description is given as follows. Given the Android platform, a set of malicious repackaged applications, and a set of applications to be inspected, design an approach to extract the behavior patterns of applications by system call sequences, and aim to detect malicious repackaged applications with the same malicious codes.

The problem could be defined as in the following:

Given

- (1) the Android platform,
- (2) a set of malicious applications  $M$ , and
- (3) a set of applications to be inspected  $E$ ,

suppose

- (1)  $M_i$  is the  $i$ -th repackaged application in  $M$ ,
- (2)  $S_i$  is the set of recorded system call sequences from  $M_i$ ,
- (3)  $S_{sign}$  is the set of significant common system call subsequences which are extracted from the set  $\{S_1, S_2, S_3, \dots, S_{|M|}\}$  and
- (4)  $E_{mal}$  is the set of true repackaged applications in  $E$ ,

the objectives are

- (1) extract  $S_{sign}$  from  $S$  and
- (2) identify  $E_{mal}$  from  $E$  by  $S_{sign}$ .

## Chapter 4 Approach

In this chapter, we detail the processing of our system call sequence detection mechanism. In Section 4.1, we give an overview of system call sequence processing which we present. The details of our approach are illustrated in Section 4.2 and Section 4.3, respectively. Final, we describe the implementation issues in Section 4.4.

### 4.1 Approach Overview

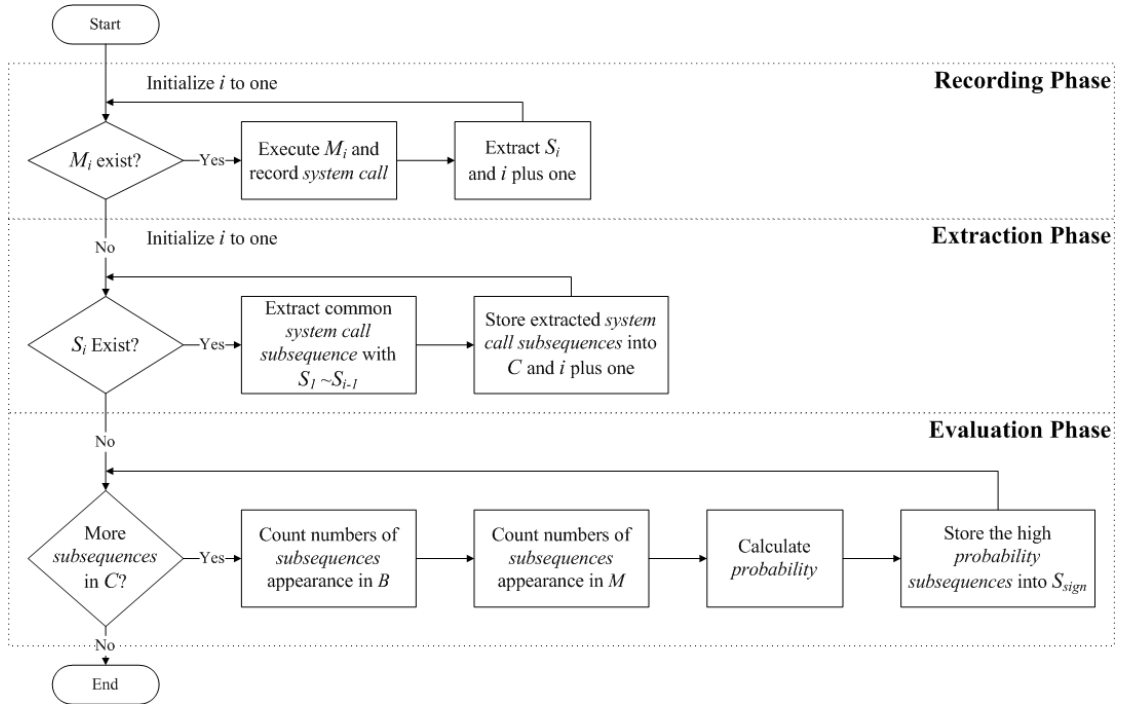


Figure 1. Flowchart of System Call Sequence Processing

Figure 1 shows the overview of our approach. For extracting the common behavior patterns, three phases, including recording phase, extraction phase, and evaluation phase, are involved in the proposed approach. In the following, we first describe these phases, respectively.

#### Recording Phase

In this phase, for all of applications in  $M$ , we sequentially select an application  $M_i$  from  $M$ , and then execute the  $M_i$  individually to record the system call requests during the Android system runtime. Next, we extract  $S_i$  from all of system call data by

the process ID of  $M_i$  and the corresponding child thread IDs. After all of  $S_i$  extracting, we can obtain  $S$  in the recording phase.

### Extraction Phase

The extraction phase aims to extract a set of common system call subsequences  $C$  from all of  $S_i$ . Since the same malicious code can trigger same system call sequences, we extract  $C$  by *Longest Common Substring* (LCSs) algorithm for comparing system call sequences of different repackaged applications. In addition, to improve the efficiency, we also present a mechanism for reducing the time cost of extracting within multi-thread system call sequences. Finally,  $C$  is obtained and each common system call subsequence in  $C$  is denoted as  $C_j$ , where  $j$  is the index of subsequences in  $C$ . It needs to address clearly that we extract substring in the system call sequences. The substring is a consecutive part of system call sequences, but we still denote it as subsequences since a substring is always a subsequence and most works use subsequences to call their system call combinations.

### Evaluation Phase

However, it is impossible for all of the extracted subsequence to be used as the detection patterns. This is because some system call subsequences not only appear in repackaged applications, but also in benign applications. For example, some extracted subsequences are too short to be a part of malicious code. So, in the evaluation phase, we evaluate appearance probability of subsequences to filter out non-discriminating subsequences. We leverage the Bayes probabilistic model to calculate the probability of  $C_j$  by counting the number of each subsequence appearance in a set of benign applications  $B$  and  $M$ . After that, we can indicate those non-discriminating subsequences in  $C$  and filter out them because they have the lower probability. After filtering, we can get  $S_{sign}$ , which has the higher probability, appearance in malware applications but have a lower probability appear in benign applications at the end of

this phase. If an application matches the same system call subsequence pattern of  $S_{sign}$  in execution time, we can claim that the application is repackaged with the same malicious code.

## 4.2 Extraction of System Call Subsequence

Since the common malicious behavior is the most resemble part of repackaged applications, the system call sequences which are collected from repackaged applications should also contain the common subsequences. It is our hope to extract the common subsequence of system call sequences to indicate the malicious behavior. In the extraction phase, suppose that each system call names as an element, the LCSs algorithm can find the longest common consecutive system call combination with two different system call sequences. Figure 2 illustrated how to extract subsequence from multiple applications in the extraction phase which contains two parts of procedure. When a new  $S_i$  is added to the procedure, we extract subsequences between  $S_i$  and all of the processed sequence sets  $\{S_1, S_2, S_3, \dots, S_{i-1}\}$  in the queue. After extracting the common subsequences, this procedure adds this application into the queue, saves all of the extracted subsequences into the storage, and processes  $S_{i+1}$  for the next target.

### Layering Multi-Thread Comparison

Android applications usually have multiple threads. Under this situation, we can get several system call sequences from different threads of an application. To extract system call subsequences by LCSs algorithm, the simpler way of processing multiple sequences is to exhaust all combinations of two sequences. However, it leads the heavy overhead if an application has many child threads. Suppose we have  $M$  applications and each application has  $N$  threads. The time complexity can be denote as  $O(N^M)$ . To overcome this drawback, we present a *Layering Multi-Thread Comparison* (LMTC) mechanism to reduce the processing overhead on multi-thread

system call sequences in the extraction phase. Since child threads have their parents, we can establish the thread tree for each  $S_i$  that depends on the relationship of threads of parent-child. All of the threads are separated into several layers in the thread tree. It is our observation, that same behaviors are usually appeared at the same layer. Even these behaviors are belonged to different applications. Since every application has a regular order of work processes, the order is also appeared in the relationship of threads of parent-child. Without loss of generality, we assume the malicious code pattern should be appeared at the same layer of process trees of different repackaged application. The comparison times of subsequence extraction can be reduced through only extracting the common subsequence with the other same layer system call sequences. Suppose each thread tree is layered into  $P$  layers in average, the time complexity of comparison can be denote as  $O(P(N/P)^M)$ .

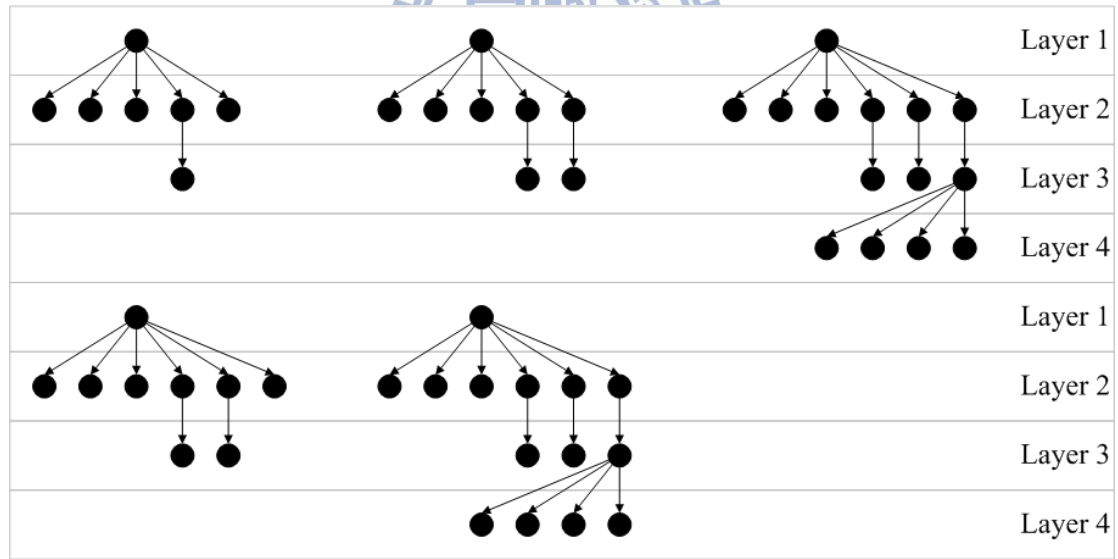


Figure 2. Example of Thread Trees

Figure 2 shows a clearer example of the process trees of malware “*DroidDream Light*”. The five thread trees are extracted from five different applications which are repackaged with the same malicious code. Obviously, they have varies tree structures and the different number of child threads. Each node denotes an individual thread, and the arrows indicate the relationship of parent-child between two threads. If we need to

extract common system call subsequences from these five process trees, for each sequence extraction, each thread tree needs to compare the chosen thread with the other four threads. The total number of extract combinations is 98784 ( $8 \times 7 \times 9 \times 14 \times 14$ ). However, only 5437 extract combinations are available by applying LMTC in this example. ( 1 ( $1 \times 1 \times 1 \times 1 \times 1$ ) combination in the first layer, 5400 ( $5 \times 5 \times 6 \times 6 \times 6$ ) combinations in the second layer, and 36 ( $2 \times 1 \times 2 \times 3 \times 3$ ) combinations in the third layer, respectively. It is worth noting that the fourth layer has no combinations since some process trees do not have threads in the fourth layer.)

### 4.3 Sequence Evaluation with Probability of Appearing

We get a set of system call subsequences after the extraction phase, but not all of the extracted subsequences can be guaranteed as the truly malicious applications that do not appear in the benign applications. Therefore, we have to evaluate the extracted subsequences in terms of the probability of appearance under the Bayes probability model. The probability formula is calculated as

$$P(M | C_j) = \frac{P(C_j | M) \cdot P(M)}{P(C_j | M) \cdot P(M) + P(C_j | B) \cdot P(B)}, \quad (1)$$

where the  $C_j$  is the system call subsequence that is used to evaluate.  $P(B)$  denotes the probability that the given applications are benign applications.  $P(M)$  denotes the probability that the given applications is a repackaged application.  $P(C_j | B)$  denotes the probability that the subsequences appear in benign applications, and  $P(C_j | M)$  denotes the probability that the subsequences appear in repackaged applications. Finally, we can get the probability  $P(M | C_j)$  that an application is a repackaged application, detected by the specific system call subsequence. In the first step of evaluation phase, we count the number of sequences occurrence in benign applications and malicious applications. Then, in the second step, we can calculate the

probability  $P(M|C_j)$  by the formula (1). After the processing of probability calculation, we obtain the subsequences that appeared in the malicious application has the higher probability than that of benign applications. And this information helps us to find  $S_{sign}$  which achieves the higher accuracy for detecting repackaged applications.

#### 4.4 Environment Implementation

To execute a large amount of applications, we deploy the Android emulator for easier managing the execution environment. When a new application executes each time, we establish a new environment and install such an application into the system to keep the integrity and correctness of environment. It can prevent the situation that an application is distorted by other applications that are not really relevant anymore. In the testing environment, the version of Android emulator we used is the Android version 2.1. According to our heuristic experiments, Android version 2.1 is the most suitable environment to execute samples.

##### System call recorder

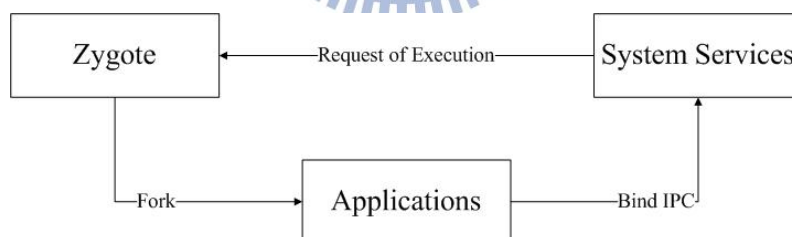


Figure 3. Zygote Mode

For recording the system calls during the period of system runtime, we use the tool “*strace*” [18] to trace system calls and signals in Linux kernel. We change the boot configuration of Android emulator to make *strace* attach on *Zygote*. As illustrating in Figure 3, *Zygote* is a monitor process of an Android system. It usually focuses on processing the request about executing a new Android application. The new applications will be forked from *Zygote* and executed. This procedure is called



*Zygote Mode*. Since all of the Android applications are forked from *Zygote*, *strace* can monitor all of the executed Android applications by tracing the child processes of *Zygote*. *strace* individually records system calls of different threads into separated files, and each thread produces system call sequences. After the time quantum is large enough to record system calls, we can extract the system call sequence of the application that we want to monitor by its process ID. Moreover, the child sequences can be traced by checking whether the thread had forked any child threads. The behavior of producing a child thread can be found by *clone* and *fork* system calls in the system call sequence. The system call sequences of child threads are also extracted for saving the complete behaviors.



## Chapter 5 Evaluation

We use five types of repackaged applications to evaluate the effectiveness of our approach. Initially, we describe the experiment environment and the used application samples in Section 5.1. In Section 5.2, we present the utility of system call sequences extraction and evaluation of our approach. Finally, the accuracy of malware detection, including false positive and false negative, are discussed in Section 5.3 and Section 5.4.

### 5.1 Experiment Environment

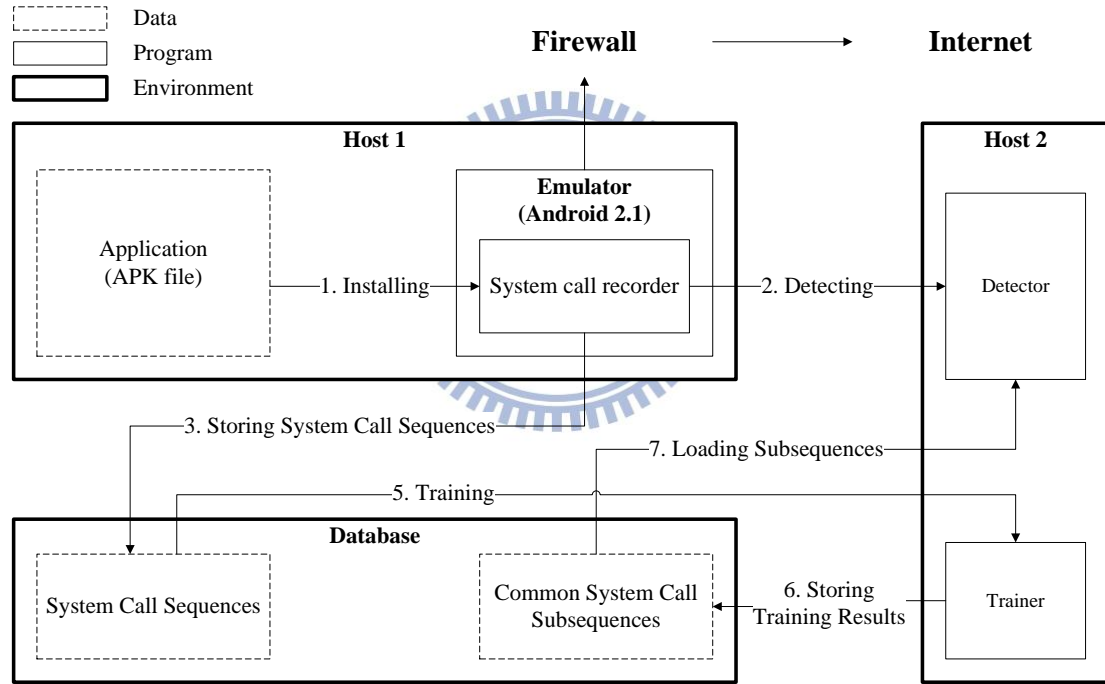


Figure 4. Experiment Environment

Figure 4 illustrates the experiment environment which includes three programs: system call recorder, trainer, and detector. The system call recorder is implemented in the emulator. The emulator is continuously connected to Internet for grabbing more behaviors of applications due to the reason that, some application behaviors need to interact with the Internet. The trainer program is responsible for the system call subsequences extraction and evaluation. And the detector program detects suspicious

applications resorting by matching with common system call subsequences that are trained by the trainer program before.

Table 3. Number of Training Samples and Sequences

| Malware Type            | Number of Training Samples | Number of Evaluation Samples | Number of Recorded Sequences | Number of Extracted Subsequences |
|-------------------------|----------------------------|------------------------------|------------------------------|----------------------------------|
| <i>Kmin</i>             | 10                         | 9                            | 181                          | 241                              |
| <i>Geinimi</i>          | 8                          | 7                            | 95                           | 156                              |
| <i>DroidDream</i>       | 4                          | 4                            | 90                           | 105                              |
| <i>BaseBridge</i>       | 4                          | 3                            | 86                           | 74                               |
| <i>DroidDream Light</i> | 3                          | 2                            | 30                           | 28                               |

For evaluating our approach, we prepare five different repackaged application types: *Kmin* [19], *Geinimi* [20], *DroidDream* [21], *BaseBridge* [22] and *DroidDream Light* [23]. Each repackaged application type has a number of applications, and we divide those repackaged applications into two parts as Table 2 listed. The first part is used to extract the common system call subsequences, and the second part to evaluate the false negative of detection rate of our approach. We also collect two sets of benign applications. The first set is used to evaluate the appearance probability of common system call subsequences for excluding useless subsequences, and the other set to evaluate the false positive of detection rate of our approach. The benign applications are collected from a third-party web site. For ensuring the benign property, we utilize several anti-virus tools [24] to check whether those applications are virus-free. For each of benign and repackaged application samples, we record the system call sequences for 3 minutes. Since the efficiency of LCSs algorithm can be lower when computing with long sequences, we only record the first 5000 system calls of each thread. However, only few threads can generate more than 5000 system calls in 3minutes.

## 5.2 System Call Sequences Distribution

Initially, we record the system call sequences of repackaged applications. The number of total recorded sequences of each type is showed in Table 3. After the system call sequences are recorded, by our extraction algorithm, we get another set of sequences that are the common system call subsequences. For each extracted common subsequence, we also apply a SHA1 digest algorithm for it to prevent the duplicate subsequences. We extract the subsequences of each type individually. Table 2 shows the result about the number of extracted sequences.

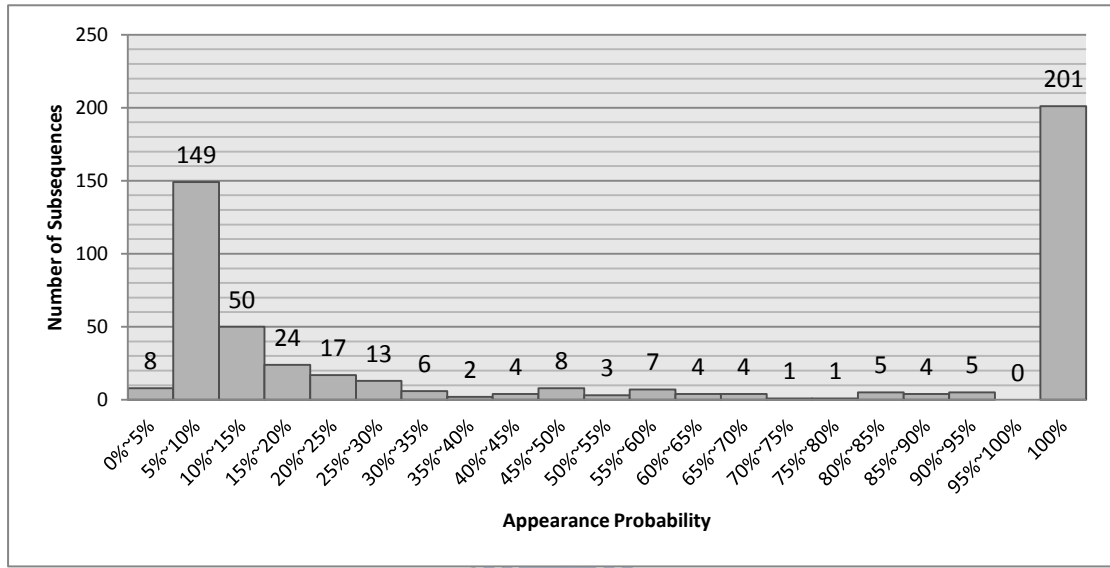


Figure 5. Probability Distribution of Subsequences

After the extraction, we count the number of subsequences appearance in the benign application set and the repackaged application set from which the subsequences are extracted for the training of Bayes probability. At first, we prepare 300 benign applications as the benign application set. The repackaged application set is also prepared as listed in Table 3. After the calculation by Bayes probability model, Figure 5 shows the probability distribution of subsequences. We calculate the probability in five different types of repackaged applications, and the number of subsequences is the sum of five results. Most of subsequences are distributed at the interval 10%~15% and 100%. The higher probability of subsequences means if the subsequences are discovered in an application, it has higher probability that the

application is repackaged. We select higher probability subsequences as the significant common system call subsequences.

Table 4. The Number of Subsequences of 100% Probability

| Malware Type           | <i>Kmin</i> | <i>Geinimi</i> | <i>DroidDream</i> | <i>BaseBridge</i> | <i>DroidDream Light</i> |
|------------------------|-------------|----------------|-------------------|-------------------|-------------------------|
| Number of Subsequences | 158         | 28             | 9                 | 3                 | 3                       |

We exclude the subsequences with lower probability than threshold, and we set the threshold as 100% initially. After excluding these subsequences, we can collect the significant common system call subsequences which have a probability that is higher than the threshold and Table 4 shows the final collected results.

### 5.3 Accuracy of Detection

Table 5. Detection Result

| Malware Type                   | <i>Kmin</i>             |                  |                  |                  |                  |
|--------------------------------|-------------------------|------------------|------------------|------------------|------------------|
| Sample Name                    | <i>Kmin - 1</i>         | <i>Kmin - 2</i>  | <i>Kmin - 3</i>  | <i>Kmin - 4</i>  | <i>Kmin - 5</i>  |
| Number of matched subsequences | 67                      | 36               | 29               | 47               | 58               |
| Sample Name                    | <i>Kmin - 6</i>         | <i>Kmin - 7</i>  | <i>Kmin - 8</i>  | <i>Kmin - 9</i>  |                  |
| Number of matched subsequences | 29                      | 26               | 25               | 52               |                  |
| Malware Type                   | <i>Geinimi</i>          |                  |                  |                  |                  |
| Sample Name                    | <i>Geimi - 1</i>        | <i>Geimi - 2</i> | <i>Geimi - 3</i> | <i>Geimi - 4</i> | <i>Geimi - 5</i> |
| Number of matched subsequences | 8                       | 13               | 1                | 7                | 3                |
| Sample Name                    | <i>Geimi - 6</i>        | <i>Geimi - 7</i> |                  |                  |                  |
| Number of matched subsequences | 5                       | 6                |                  |                  |                  |
| Malware Type                   | <i>DroidDream</i>       |                  |                  |                  |                  |
| Sample Name                    | <i>DD - 1</i>           | <i>DD - 2</i>    | <i>DD - 3</i>    | <i>DD - 4</i>    |                  |
| Number of matched subsequences | 6                       | 5                | 5                | 6                |                  |
| Malware Type                   | <i>BaseBridge</i>       |                  |                  |                  |                  |
| Sample Name                    | <i>Base - 1</i>         | <i>Base - 2</i>  | <i>Base - 3</i>  |                  |                  |
| Number of matched subsequences | 1                       | 1                | 1                |                  |                  |
| Malware Type                   | <i>DroidDream Light</i> |                  |                  |                  |                  |
| Sample Name                    | <i>DDL - 1</i>          | <i>DDL - 2</i>   |                  |                  |                  |
| Number of matched subsequences | 0                       | 1                |                  |                  |                  |

With regarding the detection accuracy, we prepare several repackaged

applications listed in Table 3 to evaluate the false negative of detection rate. In addition, we also prepare 100 benign applications to evaluate the false positive of detection rate. We put the significant common system call subsequences which are extracted from different types independently of the five type sets. Then, we take those subsequences as the pattern to match in the system call sequences of the same type of repackaged applications. An application is detected as a repackaged application if it has the same subsequence in its system call sequences. Table 5 shows the detected result. Most of repackaged applications can be detected by our approach, except the application *DDL – I* which cannot be detected by common subsequences that we trained in this experiment. We also evaluate the false positive rate with the benign application set.

Table 6. True Positive and True Negative

| Malware Type  | <i>Kmin</i> | <i>Geinimi</i> | <i>DroidDream</i> | <i>BaseBridge</i> | <i>DroidDream Light</i> |
|---------------|-------------|----------------|-------------------|-------------------|-------------------------|
| True Positive | 100%        | 100%           | 100%              | 100%              | 50%                     |
| True Negative | 100%        | 98%            | 100%              | 100%              | 100%                    |

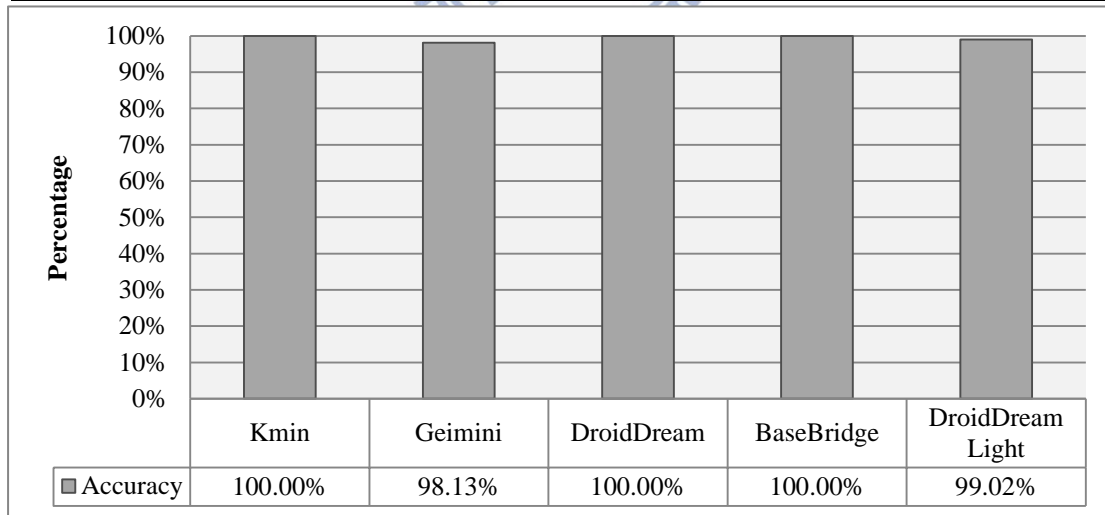


Figure 6. Accuracy of Detection

Suppose that true positive ( $TP$ ) denotes the percentage of repackaged applications detected correctly, and false negative ( $FN$ ) denotes the percentage of repackaged applications detected incorrectly; on the contrary true negative ( $TN$ )

denotes the percentage of benign applications which are not detected by any excluded subsequences, and false positive ( $FP$ ) be the percentage of benign applications which are incorrectly detected as repackaged applications.  $B$  denotes the number of benign applications.  $M$  denotes the number of repackaged applications. Table 6 demonstrates the true positive and true negative of detection rate that we evaluated with prepared samples. And the *Accuracy* can be calculated as

$$Accuracy = \frac{TP \cdot M + TN \cdot B}{(TP + FN) \cdot M + (TN + FP) \cdot B} \times 100\%. \quad (2)$$

Our approach has a good accuracy rate to detect repackaged applications. It is worth noting that, our approach has a very low false positive rate and a false negative rate. For all of five type samples, we only miss one evaluated target in 25 repackaged applications. For the 100 benign evaluation samples, our approach only has 2 false positive of benign samples. The accuracy is 97%. Referring to Figure 6, compared with other experimental results, only the true positive rate of *DroidDream Light* is significantly lower than that of our results, but we only miss one target. And this could be explained by an insufficient training and evaluation samples of *DroidDream Light*.

## 5.4 Relationship of Training Samples and Sequences

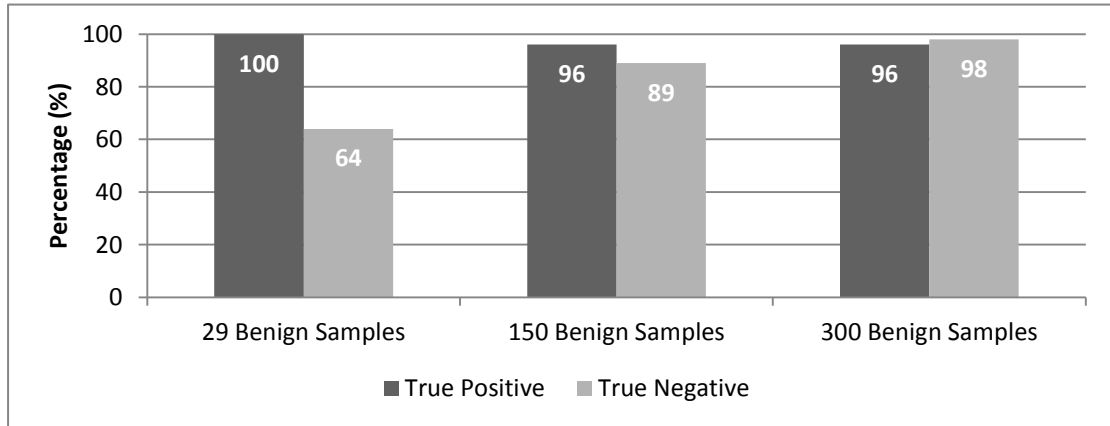


Figure 7. True Positive and True Negative of Different Training Sets

For more detail study about the accuracy, we put all of the system call

subsequences which are extracted from different types together. First, we change the number of benign training samples which are used to training the Bayes probability. The first set is the 300 benign samples that we initially used. The second set is the half of the first set. And the number of samples of third set is same as the total number of malicious training samples showed in Table 2.

All of three sets are trained with the same 29 malicious samples. We take these three sets to evaluate true positive and true negative in the same benign and malicious evaluation samples of Chapter 5.3. Figure 7 shows the result. If we add more benign samples to training, we can get a higher true negative rate. However, it is possible to get a lower true positive rate, since the more benign training samples we used will cause more subsequences are filtered. And the malware detection ability is also reduced because we get less subsequence.

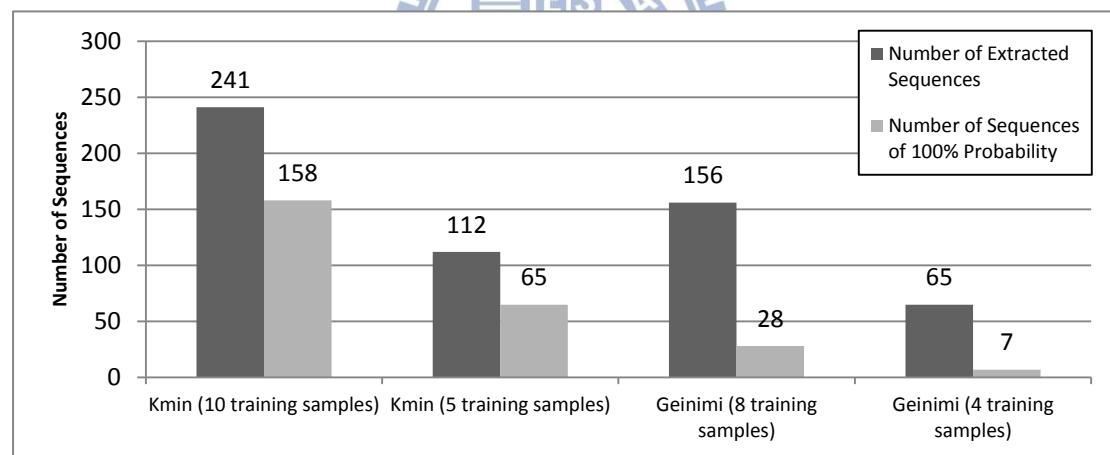


Figure 8. Sequences of Different Number of Repackaged Applications

The second part we evaluate is to reduce the number of malicious training samples. We select *Kmin* and *Geinimi* as two independent samples sets. For each of the malicious training samples set, we divide in half to get another two sets for observing the changes. All the four sets are trained with the same 300 benign samples. Figure 8 show that when we used less malicious samples to extract common subsequences, we got fewer subsequences. And the number of sequences of 100%



probability is also reduced in the similar proportion.

Table 7. Change of True Positive and True Negative

| Malware Type  | <i>Kmin</i><br>(10 samples) | <i>Kmin</i><br>(5 samples) | <i>Geinimi</i><br>(8 samples) | <i>Geinimi</i><br>(4 samples) |
|---------------|-----------------------------|----------------------------|-------------------------------|-------------------------------|
| True Positive | 100%                        | 100%                       | 100%                          | 100%                          |
| True Negative | 100%                        | 100%                       | 98%                           | 100%                          |
| Accuracy      | 100%                        | 100%                       | 98.13%                        | 100%                          |

We also evaluate the detection accuracy of these four sets. As Table 7 showed, if we use less malicious training samples, the true negative rate is increased. But it is possible to get a poor true positive rate if we reduce more malicious training samples. Since the less malicious training samples we used, the less subsequence can be extracted. Both of the miss match of benign samples and the detection ability of malicious samples are reduced.

Table 8. The Relationship of Number of Samples and Sequences (*Kmin*)

| Number of Malicious Training Samples | Number of Extracted Sequences | Number of Benign Training Samples | Number of Sequences of 100% Probability |
|--------------------------------------|-------------------------------|-----------------------------------|---|
| 10                                   | 241                           | 300                               | 158                                     |
| 10                                   | 241                           | 150                               | 161                                     |
| 5                                    | 112                           | 300                               | 65                                      |
| 5                                    | 112                           | 150                               | 67                                      |

Finally, we use *Kmin* as the example to show the relationship between the number of samples and the number of sequences in Table 8. It can be observed more malicious training samples result in more extracted sequences. However, there are less sequences can be obtained when more benign training samples are used for training. So, if we take more malicious samples to train for increasing the true positive rate, the benign training samples should also be increased to keep the high true negative rate.

## 5.5 Fixed-Length and Longest Common Subsequences

To identify the benefit of using the longest common subsequences, we compare

our approach with other works which use the fixed-length common subsequences. For example, Rozenberg, et al. [16] using 15 system calls as the length of subsequences. So we compare the accuracy of detection between using the longest common subsequences and the fixed-length common subsequences of length 15.

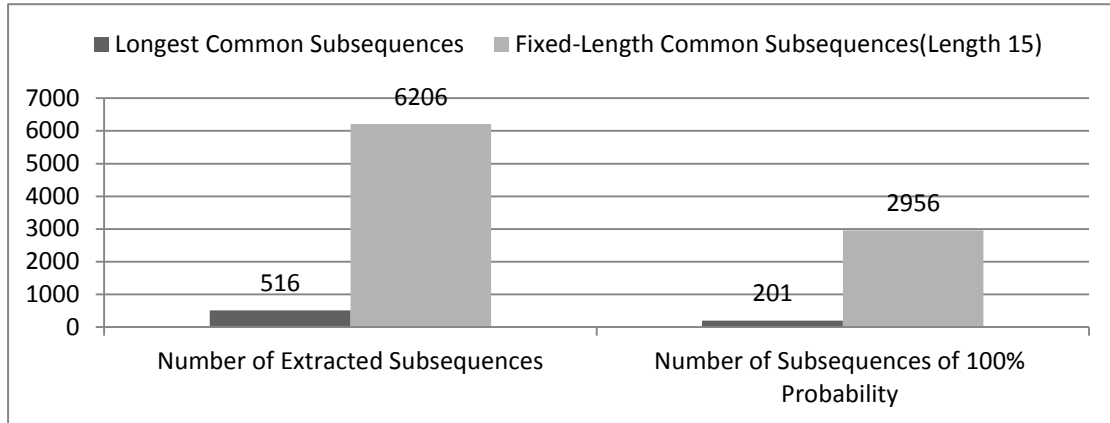


Figure 9. Number of Subsequences of Different Approaches

Figure 9 shows the number of extracted subsequences and the number of subsequences which doesn't appear in our benign training samples by using different approaches. For using the fixed-length common subsequences of length 15, the number of extracted subsequences is dramatically higher than using the longest common subsequences. After filtering by our training, the numbers of common subsequences of length 15 are still very high.

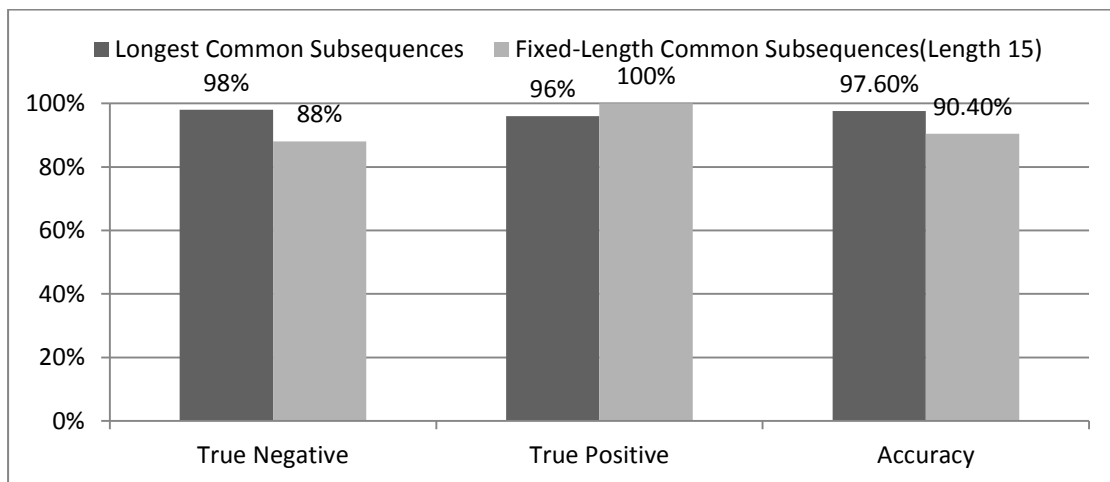


Figure 10. Accuracy of Different Approaches

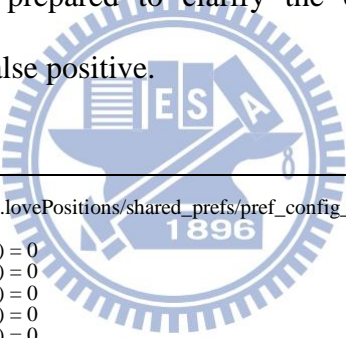
After the filtering, we use these two sets of common subsequences to evaluate

the accuracy with the same groups of repackaged applications and benign applications. As the Figure 10 indicates, using the longest common subsequences has higher true negative than using the fixed-length. In the 100 benign applications, fixed-length common subsequences have 12 false positives. But longest common subsequences only have 2. However, longest common subsequences has lower true positive rate since we have one false negative. Applying to the formula (2), the result shows that using the longest common subsequences has a higher accuracy than using fixed-length. The longest common subsequences reduce both the false positive rate and the number of subsequences which we use for detecting.

## 5.6 Case Study of Subsequences

Two case studies are prepared to clarify the content of subsequences we extracted and the reason of false positive.

### Malicious Behavior



```

1 stat64("/data/data/com.droiddream.lovePositions/shared_prefs/pref_config_setting.xml", 0xbea238c8) = -1 ENOENT
  (No such file or directory)
2 ioctl(10, 0xc0186201, 0xbea237f8) = 0
3 ioctl(10, 0xc0186201, 0xbea237f8) = 0
4 ioctl(10, 0xc0186201, 0xbea237f8) = 0
5 ioctl(10, 0xc0186201, 0xbea237f8) = 0
6 ioctl(10, 0xc0186201, 0xbea237f8) = 0
7 ioctl(10, 0xc0186201, 0xbea237f8) = 0
8 ioctl(10, 0xc0186201, 0xbea237f8) = 0
9 ioctl(10, 0xc0186201, 0xbea237f8) = 0
10 ioctl(10, 0xc0186201, 0xbea237f8) = 0
11 ioctl(10, 0xc0186201, 0xbea237f8) = 0
12 ioctl(10, 0xc0186201, 0xbea237f8) = 0
13 ioctl(10, 0xc0186201, 0xbea237f8) = 0
14 ioctl(10, 0xc0186201, 0xbea237f8) = 0
15 ioctl(10, 0xc0186201, 0xbea237f8) = 0
16 ioctl(10, 0xc0186201, 0xbea237f8) = 0
17 ioctl(10, 0xc0186201, 0xbea237f8) = 0
18 ioctl(10, 0xc0186201, 0xbea237f8) = 0
19 ioctl(10, 0xc0186201, 0xbea237f8) = 0
20 mprotect(0x41adf000, 8192, PROT_READ|PROT_WRITE) = 0
21 socket(PF_INET6, SOCK_STREAM, IPPROTO_IP) = -1 EAFNOSUPPORT (Address family not supported by
  protocol)
22 socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 26
23 getsockname(26, {sa_family=AF_INET, sin_port=htons(0), sin_addr=inet_addr("0.0.0.0")}, [16]) = 0
24 bind(26, {sa_family=AF_INET, sin_port=htons(0), sin_addr=inet_addr("0.0.0.0")}, 128) = 0
25 getsockname(26, {sa_family=AF_INET, sin_port=htons(60670), sin_addr=inet_addr("0.0.0.0")}, [16]) = 0
26 ioctl(26, FIONBIO, [1]) = 0
27 getsockname(26, {sa_family=AF_INET, sin_port=htons(60670), sin_addr=inet_addr("0.0.0.0")}, [16]) = 0
28 connect(26, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("184.105.245.17")}, 128) = -1
  EINPROGRESS (Operation now in progress)

```


Figure 11. The Subsequence of Malicious Behavior

We show a piece of subsequence which has malicious behaviors in Figure 9. It is a sequence which is extracted from malware “*DroidDream*”. Referring to the

subsequence, the application checks whether the file “pref\_config\_setting.xml” has existed at the first. If a device had been compromised by “*DroidDream*”, the file “pref\_config\_setting.xml” should exist. This examination is implemented to avoid stealing information from the same devices again. Since our environment is clear, the file “pref\_config\_setting.xml” should not exist in the file system. So, after the examination, the application starts to ask the remote server (184.105.245.17) for preparing to attack this device.

### The Reason of False Positive

In our experiment of Chapter 5.3, our approach still has two false positives when detecting a benign application as a malicious one. These two cases are caused by the same subsequence which is extracted from malware “*Geinimi*”.



```

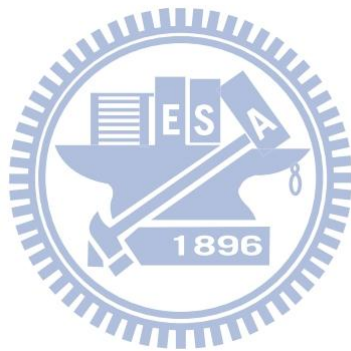
1    prctl(0x8, 0x1, 0, 0, 0) = 0
2    setgroups32(2, [3003, 1015]) = 0
3    setgid32(10028) = 0
4    setuid32(10028) = 0
5    gettid() = 251
.....
95   open("/data/app/com.moonbeam.android.magicshop.apk", O_RDONLY|O_LARGEFILE) = 24
96   lseek(24, 0, SEEK_CUR) = 0
97   lseek(24, 0, SEEK_END) = 3493907
98   lseek(24, 0, SEEK_SET) = 0
99   lseek(24, 3493885, SEEK_SET) = 3493885
100  read(24, "P", 1) = 1
101  read(24, "K", 1) = 1
102  read(24, "\5", 1) = 1
103  read(24, "\6", 1) = 1
104  lseek(24, 0, SEEK_CUR) = 3493889
105  lseek(24, 0, SEEK_CUR) = 3493889
106  lseek(24, 0, SEEK_END) = 3493907
107  lseek(24, 3493889, SEEK_SET) = 3493889
108  lseek(24, 3493889, SEEK_SET) = 3493889
109  read(24, "\0\0\0\0\30\1\30\1\2H\0\0\373\0075\0\0\0", 18) = 18
110  lseek(24, 0, SEEK_CUR) = 3493907
111  lseek(24, 0, SEEK_END) = 3493907
112  lseek(24, 3493907, SEEK_SET) = 3493907
113  lseek(24, 3475451, SEEK_SET) = 3475451
114  read(24, "PK\1\2\24\0\24\0\10\0\10\0\262\246T=j\260\257\322\34\0"..., 4096) = 4096
115  lseek(24, 3479547, SEEK_SET) = 3479547
116  read(24, "\0\0\0\367\226i;H\360\30PH\266\0\0H\266\0\0\23\0\0\0"..., 4096) = 4096

```

Figure 12. The Case of False Positive

Referring to Figure 10, the subsequence is the case of false positive which is executed when an application startup. This subsequence contents some of preparing operations before the system reads the content of applications. So, it is possible to appear both in repackaged applications and benign applications. However, the common subsequences of benign behavior should be filtered in our training. But this

subsequence contents 116 system calls which is too long. Therefore, it has small probability to happen again in other applications although the subsequence only contains benign behaviors.



## Chapter 6 Conclusions and Future Works

Due to openness and accessibility of Android, adversaries can repackage malicious code into the malicious applications as the benign applications. For users, the outward appearance of repackaged applications is like a normal application, and they are prone to distribute on marketplaces. Over the past few years, most of related works detect repackaged applications by comparing them with the original benign applications. However, it is computational infeasible to search all original applications in the Internet. To overcome this drawback, we present an approach with the concept on extracting the common behaviors of repackaged applications in system call sequences. The detection only requires a few repackaged applications with the same type to extract the common system call subsequences. In addition, our approach does not need to collect and compare the original benign applications with the repackages applications. We take those extracted common system call subsequences as the behavior patterns to detect repackaged applications.

In our experiment, we use five different types of repackaged applications to evaluate the accuracy rate. Our approach extracts 238 common system call subsequences from training samples, and the detection result demonstrates that our approach has higher true positive rate in detecting most repackaged applications. We evaluate 25 repackaged applications and only miss one evaluated target. Our approach also has higher true negative rate in verifying benign applications. The accuracy of detection rate is 97.6% in all of evaluation applications.

However, the evasion is still possible in the system call sequences detection. For example, the attacker can insert some system calls which are no effect to its behavior but can change the combination of system call sequences. To prevent this situation, we should allow some gaps exist in the sequence. For our LMTC algorithm, the

repackaged applications can produce the same thread into different layers to escape from our sequences extraction since our approach only compare between the sequences of same layers. But we still can compare between all of the sequences rather than only the same layers to prevent this scenario.

In the future, we hope to study application behaviors analysis. We only can record the behaviors which are automatically generated from applications. However, applications have more behaviors when users perform the related operations. Therefore, we need to design a tool to grab the complete behaviors in the applications when they are triggered. Moreover, we also hope to develop an on-device detector of system call sequences detection which can work like the anti-virus software that directly detects repackaged applications on mobile devices.



## Reference

- [1] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android security," *IEEE Security & Privacy Magazine*, vol. 7, no. 1, pp. 10–17, 2009.
- [2] T. Vidas, D. Votipka, and N. Christin, "All your droid are belong to us: A survey of current android attacks," *Proceedings of the 5th USENIX conference on Offensive technologies*, San Francisco, CA, USA, August 2011.
- [3] Y. Zhou, and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2012.
- [4] M. Zheng, P. P.C. Lee, and J. C.S. Lui, "ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems," *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'12)*, Heraklion, Crete, Greece, July 2012
- [5] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, Vancouver, BC, Canada, pp. 393–407, October 2010.
- [6] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," *Proceedings of the 16th ACM conference on Computer and communications security*, Chicago, IL, USA, pp. 235–245, November 2009.
- [7] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated security certification of Android applications," Technical report, University of Maryland, 2009.
- [8] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," *Proceedings of the 5th International Conference on Malicious and Unwanted Software (Malware 2010)*, Nancy, France, pp. 55–62, 2010.
- [9] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for Android malware detection," *Proceedings of the 7th International Conference on Computational Intelligence and Security*, Sanya, Hainan, China, pp. 1011–1015, December 2011.
- [10] I. Burguera, U. Zurutuza, and N. T. Simin, "Crowdroid: Behavior-based malware detection system for Android," *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, Chicago, IL, USA, pp. 15–25, October 2011.



- [11] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix process," *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, pp. 120–128, May 1996.
- [12] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, pp. 133–145, May 1999.
- [13] K. Wee, and B. Moon, "Automatic generation of finite state automata for detecting intrusions using system call sequences," *Proceedings of International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security*, St. Petersburg, Russia, 2003.
- [14] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *ACM Transactions on Information and System Security*, vol. 9, no. 1, pp. 61–93, February 2006.
- [15] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, Dubrovnik, Croatia, pp.5–14, September 2007.
- [16] B. Rozenberg, E. Gudes, Y. Elovici, and Y. Fledel, "A Method for Detecting Unknown Malicious Executables," *Proceedings of the 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, Changsha, China, pp. 190–196, November 2011.
- [17] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party Android marketplaces," *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, San Antonio, TX, USA, February 2012.
- [18] "strace," available at: <http://sourceforge.net/projects/strace/>
- [19] "Encyclopedia entry: Trojan:AndroidOS/Kmin.A," available at: <http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?Name=Trojan%3AAndroidOS%2FKmin.A> .
- [20] T. Strazzere, and T. Wyatt, "Geinimi Trojan Technical Teardown," *Lookout Mobile Security*, 2011.
- [21] "Lookout Mobile Security Technical Tear Down," Lookout Mobile Security.
- [22] "Android.Basebridge," available at: [http://www.symantec.com/security\\_response/writeup.jsp?docid=2011-060915-4938-99](http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99) .
- [23] "Security Alert: New DroidDream Light Variant Published to Android Market," available at: <http://blog.mylookout.com/blog/2011/07/08/security-alert-new-droiddream-light->

variant-published-to-android-market/ .

- [24] “VirusTotal - Free Online Virus, Malware and URL Scanner,” available at:  
<https://www.virustotal.com/> .

