Available online at www.sciencedirect.com

SciVerse ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Identifying android malicious repackaged applications by thread-grained system call sequences

Ying-Dar Lin [a], Yuan-Cheng Lai [b,*], Chien-Hung Chen [a], Hao-Chuan Tsai [c]

[a] Department of Computer Science, National Chiao Tung University, Hsinchu 30010, Taiwan
[b] Department of Information Management, National Taiwan University of Science and Technology, Taipei 10607, Taiwan
[c] Network Benchmarking Lab, National Chiao Tung University, Hsinchu 30010, Taiwan

ARTICLE INFO

Article history:
Received 19 February 2013
Received in revised form
5 August 2013
Accepted 16 August 2013

Keywords:
Malicious repackaged applications
Dynamic analysis
System call
Android
Longest common substring

ABSTRACT

Android security has become highly desirable since adversaries can easily repackage malicious codes into various benign applications and spread these malicious repackaged applications (MRAs). Most MRA detection mechanisms on Android focus on detecting a specific family of MRAs or requiring the original benign application to compare with the malicious ones. This work proposes a new mechanism, SCSdroid (System Call Sequence Droid), which adopts the thread-grained system call sequences activated by applications. The concept is that even if MRAs can be camouflaged as benign applications, their malicious behavior would still appear in the system call sequences. SCSdroid extracts the truly malicious common subsequences from the system call sequences of MRAs belonging to the same family. Therefore, these extracted common subsequences can be used to identify any evaluated application without requiring the original benign application. Experimental results show that SCSdroid falsely detected only two applications among 100 evaluated benign applications, and falsely detected only one application among 49 evaluated malicious applications. As a result, SCSdroid achieved up to 95.97% detection accuracy, i.e., 143 correct detections among 149 applications.

© 2013 Elsevier Ltd. All rights reserved.

1. Introduction

Smart handheld devices have become popular because they can provide many services with existing networks. Currently most handheld devices adopt the Android platform because of its openness and accessibility, and so many developers are attracted to design new applications for Android. However, as a result of its openness, an adversary can easily create malicious applications and spread them. Among them, malicious repackaged applications (MRAs) are undoubtedly common (Zhou & Jiang, 2012; Zheng et al., 2012). An adversary downloads popular benign applications, dissembles them, embeds malicious codes, reassembles them, and then submits the MRAs to official Android Market or third-party APP stores (Zhou & Jiang, 2012; Zheng et al., 2012; Enck et al., 2009; Vidas et al., 2011). For smart handheld devices, precisely detecting MRAs from downloaded applications is highly desirable.

The MRA detection mechanisms can be roughly categorized into two categories. The first category is *static analysis*, which searches malicious signatures in an application without executing it (Enck et al., 2009; Fuchs et al., 2009; Zhou

* *Corresponding author*. Tel.: +886 227376794. Fax: +886 227376777.
E-mail addresses: ydlin@cs.nctu.edu.tw (Y.-D. Lin), laiyc@cs.ntust.edu.tw (Y.-C. Lai), chchen.cs99g@nctu.edu.tw (C.-H. Chen), haochuan@nbl.org.tw (H.-C. Tsai).
0167-4048/$ – see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cose.2013.08.010

et al., 2012). Static analysis is simple and efficient in providing rapid detection and classification for known MRAs. Unfortunately, it suffers from hardly detecting unknown MRAs, because every repackaged application can have different signatures by code obfuscation and encryption. Most static analysis methods using traditional static signatures to detect unknown MRAs will fail since the code has already been changed to a different appearance. To overcome this limitation, another category, *dynamic analysis* (Enck et al., 2010; Isohara et al., 2011; Blasing et al., 2010; Burguera et al., 2011; Shabtai et al., 2011; Zhou et al., 2012), executes applications to monitor their runtime behavior, such as network access and memory modifications. Compared with static analysis, dynamic analysis can more effectively detect MRAs (Shabtai et al., 2011; Moser et al., 2007).

Several MRA detection mechanisms using dynamic analysis have been proposed on Android (Enck et al., 2010; Isohara et al., 2011; Blasing et al., 2010; Burguera et al., 2011; Shabtai et al., 2011; Zhou et al., 2012) and most of them used system calls (Isohara et al., 2011; Blasing et al., 2010; Burguera et al., 2011). Since system calls are embedded in the kernel space at the low layer of the Android architecture, it is not feasible to hide the system calls activated by applications. However, some mechanisms suffer from the drawback of requiring the original application to compare it with the malicious repackaged one (Burguera et al., 2011). For popular applications, we can easily find the original one in the official Android marketplace. However, not every MRA is repackaged from popular applications. Also, sometimes original applications are not in the official marketplace but in a third-party marketplace. Thus it is difficult to find the original application for every MRA. The authors in (Zhou & Jiang, 2012) tried to find original applications for verifying MRAs manually, but were unable to find some of them. Without the original application, these mechanisms cannot detect the corresponding MRA. On the other hand, some mechanisms only detect specific MRAs, such as those violating the permissions or leaking sensitive data (Enck et al., 2010). Some mechanisms only use the number of system calls, thus generating low detection accuracy (Blasing et al., 2010; Burguera et al., 2011).

To overcome these problems, this work proposes a new MRA detection mechanism, SCSdroid (System Call Sequence Droid), by utilizing *thread-grained system call sequences* during runtime. The key concept is that if an MRA can be camouflaged as a benign application, its malicious behavior would still appear in the thread-grained system call sequences. SCSdroid uses thread-grained system call sequences, rather than process-grained ones. A thread-grained system call sequence is the system calls recorded for a thread, while a process-grained system call sequence is the system calls recorded for a process. That is, the thread-grained system call sequences mean that the system calls produced by the process and the child-threads forked from it are independently recorded, while process-grained sequence means that the system calls produced by the process and all its child-threads are recorded together. Since malicious behavior always happens in a single thread, not across multiple threads, it is difficult to identify the malicious behavior if the system calls from the process and different threads are mixed together. Previously we actually tried to adopt process-grained system call sequences to identify MRAs and did not obtain acceptable results, although these results are not shown in this paper.

SCSdroid first captures the system call sequence of each thread at executing MRAs and then extracts the common subsequences, which are the common parts of these captured system call sequences. These extracted common subsequences can be only regarded as possibly malicious behavior of MRAs because they may also exist in benign applications. Therefore, the *Bayes Theorem* is adopted in SCSdroid to filter these non-discriminating common subsequences and then find the common subsequences which indicates the truly malicious behavior presenting in the MRAs.

SCSdroid has three advantages: (1) Even if MRAs have been encapsulated or obfuscated, the proposed mechanism, SCSdroid, can still capture the truly malicious behavior by extracting their common subsequences; (2) SCSdroid adopts thread-grained system call sequences, rather than process-grained ones, and can thus more precisely capture malicious behavior; and (3) Without requiring the original benign

Table 1 – Related work of MRA detection on Android.

Related work	Category	Key feature	Detecting Target	Main Drawbacks
Kirin (Enck et al., 2009)	Static	Permission	MRAs violating permissions	Only detects specific MRAs
ScanDroid (Fuchs et al., 2009)	Static	Data flow & permission in program codes	MRAs having the conflict between permission and data flow	Only detects specific MRAs
DroidMoss (Zhou et al., 2012)	Static	Code instructions	General MRAs	Needs the original application
TaintDroid (Enck et al., 2010)	Dynamic	Data flow	MRAs leaking sensitive data	Only detects specific MRAs
KBB (Isohara et al., 2011)	Dynamic	System calls and their parameters	General MRAs	Generates many false positives
AASandbox (Blasing et al., 2010)	Static & dynamic	Number of system calls	General MRAs	Generates low detection accuracy
CrowDroid (Burguera et al., 2011)	Dynamic	Amount of system calls	General MRAs	1. Needs a lot of users 2. Needs the original application
Andromaly (Shabtai et al., 2011)	Dynamic	Abnormal behavior	Malicious applications	Generates many false positives
SCSdroid [this work]	Dynamic	System call sequences	General MRAs	

applications, SCSdroid can extract the common subsequences from a few training MRAs.

The rest of this paper is organized as follows. Section 2 gives a brief survey of the related work on Android. Section 3 illustrates the details of SCSdroid. The experiment environment and experiment results are described in Section 4 and Section 5, respectively. Finally, Section 6 gives our conclusions.

2. Related work

As Table 1 summarizes, the MRA detection mechanisms on Android can be divided into two categories. The first is static analysis which scans the software for matching malicious patterns without executing it. *Kirin* checked permissions of applications at install-time and thus detected the MRAs that violates a given system policy (Enck et al., 2009). However, Kirin did not identify the MRAs that can establish communication links without requiring any permission. Similarly, *ScanDroid* extracted security specifications for automatically checking data flows and permissions in the application codes (Fuchs et al., 2009); it also suffered from the weakness that Kirin has. *DroidMoss* aimed to discover the MRAs in the third-party marketplaces. It calculated the similarity scores by comparing hash values of the author information and code instructions between the original application in the official market and the MRA in the third-party marketplace (Zhou et al., 2012). DroidMoss suffered from two weaknesses. The first one is that once the MRA has been distributed in both the official market and the third-party marketplace, DroidMoss cannot detect this MRA. The second is that it is difficult to obtain all of the original applications from the official market (Zhou & Jiang, 2012).

The second category is dynamic analysis which collects the runtime information at executing applications. *TaintDroid* tracked the sensitive data by labeling data in the memory and detected the privacy leaking in applications (Enck et al., 2010). However, it focused on the MRAs that attempt to obtain sensitive data. *KBB* (Kernel-based Behavior) collected the runtime information of applications through system calls (Isohara et al., 2011). KBB first generated a set of regular expression rules from the names and parameters of system calls of training MRAs. Then it could detect the unknown MRA by mapping its system calls and parameters with the regular expression rules. However, KBB generated many false positives. For example, 80 applications that matched with any signature only contain 37 malicious applications, were reported in the evaluation (Isohara et al., 2011).

AAsandbox is a hybrid approach of static and dynamic analyses (Blasing et al., 2010). It decompiled the installed application and searched for suspicious patterns in the decompiled codes. During the runtime, AAsandbox counted the number of all system calls to detect MRAs. However, the data obtained by AASandbox is very diverse, causing low detection accuracy (Blasing et al., 2010). *CrowDroid* is another typical mechanism to evaluate MRAs by counting the number of system calls (Burguera et al., 2011). CrownDroid collected all the system calls used from a set of users during the runtime. It adopted the K-means clustering algorithm to classify the collected data into two groups, the benign group and the malicious group, which can be used to identify the specified user who is running the MRA. CrowDroid needs a set of users to execute the same original application and the same corresponding MRA. Unfortunately, finding the original applications of all MRAs is usually inefficient and even impossible in Android marketplaces.

Andromaly collected several runtime features for detecting MRAs and evaluated different kinds of learning algorithms used in dynamic analysis for Android malware (Shabtai et al., 2011). This work is very good for evaluating learning algorithms used in dynamic analysis. However, it does not use real-world samples to evaluate the results. Also, most MRAs include benign and malicious behaviors because the malicious payload is enclosed into benign applications. Andromaly does not separate behaviors, with the result that the learned characteristics about malicious behavior may be mixed with benign behavior, and then false positives appear when identifying benign applications. In contrast, SCSdroid filters the system call sequences recorded from benign samples to avoid false positives.

3. The proposed SCSdroid

Although some literature uses system calls to detect MRAs, they suffer from a few drawbacks (Isohara et al., 2011; Blasing et al., 2010; Burguera et al., 2011; Shabtai et al., 2011). First, they use the number of system calls, but the metric is too coarse, resulting in low detection accuracy (Blasing et al., 2010; Burguera et al., 2011). Second, they need the original benign application so that they can distinguish the number of system calls between the original benign application and the corresponding MRA (Burguera et al., 2011). As a result, this work proposes SCSdroid, which uses the thread-grained system call sequences, because these sequences can be regarded as the actual behavior of the application.

3.1. Overview

The key concept of SCSdroid is that MRAs belonging to the same family, i.e., a group of MRAs which embed the same malicious codes into benign applications, will have *common* malicious behavior. Therefore SCSdroid extracts the common subsequences contained in the thread-grained system call sequences of MRAs belonging to the same family. Since some common subsequences may exist in both benign applications and MRAs, it is essential to filter those ambiguous common subsequences to obtain the truly malicious common subsequences. According to these truly malicious common subsequences, the evaluated application can be effectively identified as a MRA or a benign one.

Assume that a set of MRAs, M, of the same family has initially been collected. Let M_i denote the i-th MRA in the set M. When executing M_i, we obtain the corresponding system call sequence set S_i, which include the system call sequences of all threads. $S_{i,j}$ denotes the system call sequence of the j-th thread (the process is regarded as the first thread) in the set S_i and S is the set of all S_i. To efficiently obtain a common subsequence set, CS, from all pairs of S_i and S_j, the layering multi-thread

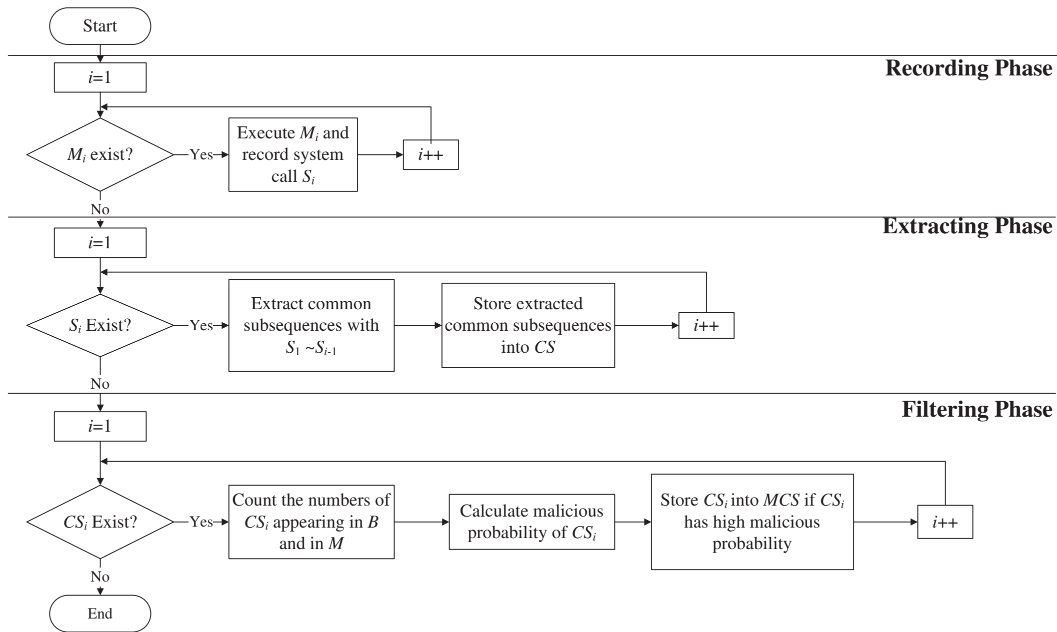

Fig. 1 – Flowchart in training phase of SCSdroid.

comparison algorithm is proposed. Finally, Bayes theorem is adopted to obtain the set of truly malicious common subsequences, *MCS*, from CS. CS_i and MCS_i represent the *i*-th elements in the set *CS* and *MCS*, respectively. Also let the notation $|X|$ denote the number of elements in the set *X*.

SCSdroid involves two phases: the training phase and the identifying phase. The set of *MCS*, will be obtained in the training phase, while the evaluated applications will be identified as MRAs or benign applications in the identifying phase. The flowchart in the training phase is shown in Fig. 1 and will be further explained in the next subsection. In the identifying phase, if an evaluated application has system call subsequences that exist in *MCS* at runtime, it is detected as a MRA.

3.2. *Training phase*

The training phase is composed of three steps: the recording step, the extracting step, and the filtering step. The recording step generates all S_i, the extraction step generates CS, and the filtering step generates *MCS*.

3.2.1. *Recording step*

In this phase, we initially collect a set of MRAs belonging to the same family, *M*, sequentially run the *i*-th MRA M_i in this set, and obtain the corresponding system call sequence set S_i in the Android system at runtime. Each system call sequence belonging to the set S_i is extracted by the process ID of M_i and the thread IDs of its descendant threads that are derived from M_i. For example, assume that the MRA M_1 forks two child threads. At first, the system call sequence is recorded according to process ID of M_1. Since M_1 forks two child threads, the recording step also records the system call sequences generated by these two child threads IDs. Hence, S_1, the system call sequence set of M_1, contains three system call sequences, $S_{1,1}$, $S_{1,2}$, and $S_{1,3}$. After recording all S_i, the overall system call sequence set $S = \{S_i | 1 \leq i \leq |M|\}$ can be obtained.

3.2.2. *Extracting step*

The goal of the extracting step is to extract the common subsequences by adopting the *Longest Common Substring* (LCS) algorithm to indicate the possibly malicious behavior existing in MRAs. Assume that there are two sequences “ABABC” and “BABCA”. The LCS algorithm will output the longest common subsequences, “BABC”, of these two sequences.

Since an Android application usually has multiple threads, each S_i will include many system call sequences which are extracted for different threads. In extracting common subsequences from the pair of S_i and S_j, a simple way is directly applying the LCS algorithm to compare one sequence in S_i with another sequence in S_j. Thus the total number of comparisons is $|S_i| \times |S_j|$. This overhead is heavy when applications contain many threads.

To reduce the overhead, we propose the *Layering Multi-Thread Comparison* (*LMTC*) mechanism to efficiently extract common subsequences from system call sequences. The thread tree for each S_i is established depending on the hierarchy of threads. We observe that the same behavior appear in the threads at the same layer even if the behavior belong to different applications. Thus the malicious behavior activated by malicious codes is very likely to appear at the same layer in the thread trees of different MRAs. Hence, LMTC only apply the LCS algorithm to find the common subsequences among $S_{i,x}$ and $S_{j,y}$ if they are at the same layer.

LMTC significantly reduces the number of comparisons in extracting common subsequences from multi-thread system call sequences. Assume in S_i, each thread tree has *P* layers on average and each layer except the first layer has the same number of threads. That is, there are $(|S_i| - 1)/(P - 1)$ threads for each layer except the first layer. Hence, in extracting common subsequences from the pair of S_i and S_j, the number of comparisons in each layer is $((|S_i| - 1)/(P - 1)) \times ((|S_j| - 1)/(P - 1))$. Since the first layer needs one comparison and there are other

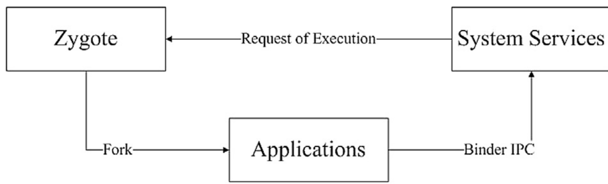

Fig. 2 – Zygote mode.

$P-1$ layers, the number of comparisons is only $1+(P-1)((|S_i|-1)/(P-1))\times((|S_j|-1)/(P-1))=1+(|S_i|-1)\times(|S_j|-1)/(P-1)$.

Note that the SHA-1 cryptographic hash function is used to prevent duplicate common subsequences in the extracting step.

3.2.3. *Filtering step*

The extracted common subsequences cannot be directly used in detecting MRAs because some of them appear in both MRAs and benign applications. These non-discriminating common subsequences do not denote truly malicious behavior and must be filtered out. Let the malicious probability of the common subsequence CS_i be denoted as $P(M|CS_i)$, which is the probability of an application being the MRA when the common subsequence CS_i is extracted from this application. We use Bayes Theorem to calculate this probability as

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

where $P(B)$ and $P(M)$ are the probabilities that given applications are benign applications and MRAs, respectively. $P(CS_i|B)$ and $P(CS_i|M)$ represent the probabilities that the common subsequence CS_i appears in benign applications and in MRAs, respectively.

To obtain $P(CS_i|B)$ and $P(CS_i|M)$, we first count each common subsequence appearing in how many benign applications and how many MRAs. Then, for each CS_i, the malicious probability $P(M|CS_i)$ is calculated by Equation (1). To obtain the set of truly malicious common subsequences MCS, we extract the common subsequences that are only appear in MRAs, but not in benign applications. Thus the common subsequence CS_i with 100% malicious probability, i.e., $P(M|CS_i)=100\%$, are reserved in the set MCS. That is, the subsequences in MCS should be the truly malicious behavior.

4. Experiment environment

For easy management of the execution environment, the emulator of Android version 2.1 was adopted because this is the most suitable version to execute samples. Before executing any application, we re-established a new execution environment to maintain its integrity and correctness, and only installed the application. This can prevent the application from being obstructed by other unrelated executing applications.

We used an emulator, rather than a real Android device, to do our experiments because of two reasons. First, conducting the experiments on an Android device takes a lot of efforts. We have to install a MRA or a benign application, record its system call sequences, identify it, and then clean it (recover the original image), in an Android device. For each application, we have to manually do the same procedure to obtain the

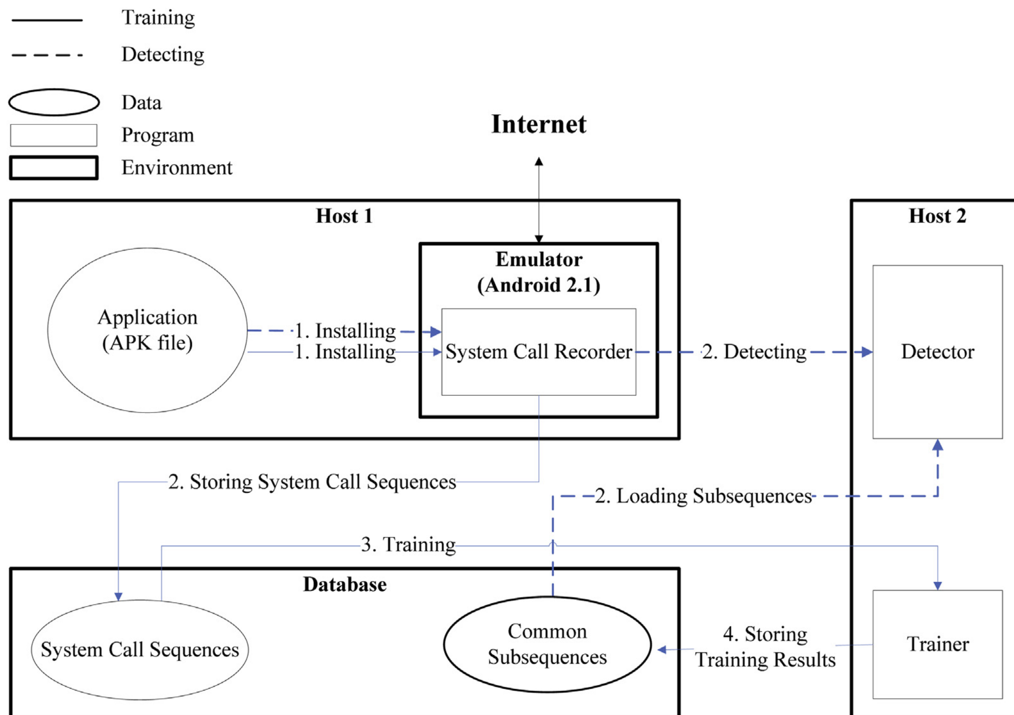

Fig. 3 – Experimental environment.

Table 2 – The numbers of training and evaluated samples.

Malware family	Number of training samples	Number of evaluated samples
Kmin	10	9
ADRD	9	9
AnserverBot	9	9
Geinimi	8	7
Zsone	6	6
DroidDream	4	4
BaseBridge	4	3
DroidDream Light	3	2

testing results. However, using an emulator, we can write some programs to automatically operate the procedure. Second, we actually spent a lot of efforts to detect some MRAs in a Android device. The detecting results on an Android device fully matched those on an emulator. Thus the testing results on an Android emulator should be convincing.

To record the system calls at runtime, the tool "*strace*" (Strace) was used to trace system calls in the Linux kernel. The boot configuration of the Android emulator was changed to make *strace* attach to *Zygote*, which is a process that focuses on processing the request for executing a new Android application (Ehringer, 2010). As shown in Fig. 2, a new application is forked from *Zygote* and then executed in the Dalvik virtual machine architecture (Ehringer, 2010). Since all of the Android applications are forked from *Zygote*, *strace* monitors all of the executed Android applications by tracing the child processes of *Zygote* and individually records system calls of different processes into separate files by monitoring their process IDs. Note that when the application accesses some system services, it must utilize binder Interprocess Communication (IPC) to feedback the request of execution. Moreover, whether the thread has forked any child threads can be found by *clone* and *fork* system calls in the system call sequence. The system call sequences of child threads are also extracted in order to memorize the complete behavior.

Fig. 3 illustrates the experimental environment, including the system call recorder module, the trainer module, and the detector module. Since some behavior of the application might interact with the Internet, the system call recorder module is implemented in the emulator, which continuously communicates with the Internet to capture all network-related behavior of the application. The trainer module is responsible for extracting the truly malicious common subsequences, *MCS*, and the detector module detects the evaluated applications by using *MCS*.

Eight families of MRAs, including *Kmin* (Encyclopedia), *ADRD* (Android), *AnserverBot* (Zhou & Jiang, 2011), *Geinimi* (Strazzere & Wyatt, 2011), *Zsone* (Security), *DroidDream* (Lookout), *BaseBridge* (Android) and *DroidDream Light* (Security alert), were used to evaluate the effectiveness of SCSdroid. The MRAs in each family are divided into two sets, training samples and evaluated samples. The training samples are used to extract the common subsequences from the recorded system call sequences, while the evaluated samples are used to evaluate the SCSdroid performance. The numbers of training malicious samples and evaluated malicious samples are shown in Table 2. In addition, the training benign samples are used to calculate the malicious probabilities of the common subsequences for SCSdroid while the evaluated benign samples are used to evaluate the SCSdroid performance. There are 300 training benign samples and another 100 evaluated benign samples that were downloaded from the Google official market and third-party markets, and had been confirmed as benign applications by utilizing several antivirus tools (VirusTotal).

5. Experiment results

Let the true positive ratio (*TP*) denote the ratio of the MRAs that are correctly detected, and the false negative ratio (*FN*) denotes the ratio of the MRAs that are falsely detected as benign applications. On the contrary, the true negative ratio (*TN*) denotes the ratio of benign applications that are correctly detected, and the false positive ratio (*FP*) denotes the ratio of benign applications that are falsely detected as MRAs. Moreover, N_B and N_M denote the numbers of the evaluated benign applications and MRAs, respectively. Then a performance metric, detection accuracy, denoted as *DA*, is defined as

$$DA = \frac{TP \times N_M + TN \times N_B}{N_M + N_B} \times 100\%. \quad (2)$$

Some experiments were conducted to make some observations: (1) For MRAs, what is the distribution of malicious probability for the extracted common subsequences? (2) Can SCSdroid detect the evaluated MRAs with high detection

Table 3 – The numbers of sequences and common subsequences in each malware type.

Malware family	Number of sequences (*S*)	Number of common subsequences (*CS*)	Number of malicious common subsequences (*MCS*)
Kmin	181	241	158 (65.6%)
ADRD	82	102	28 (27.4%)
AnserverBot	192	387	79 (20.4%)
Geinimi	95	156	28 (17.9%)
Zsone	51	63	7 (11.1%)
DroidDream	90	105	9 (8.6%)
BaseBridge	86	74	3 (4.1%)
DroidDream Light	30	28	3 (10.7%)

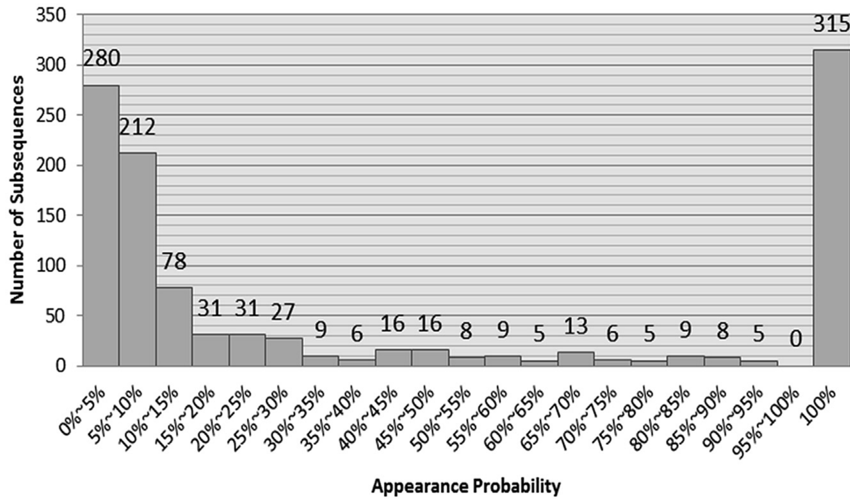

Fig. 4 – The distribution of malicious probabilities of subsequences.

accuracy? (3) What are the effects of the number of training samples on the detection accuracy? (4) Compared with fixed-length common subsequences, do the longest common subsequences adopted in SCSdroid have the better detection accuracy?

5.1. *Distribution of the malicious probability*

Table 3 shows the number of total recorded sequences for each family of MRAs. By applying the LMTC mechanism, the common subsequences set *CS* can be obtained from the recorded system call sequences. For example, 241 different common subsequences are extracted from ten training MRAs infected by *Kmin*.

Table 3 also shows the number of the common subsequences with 100% malicious probability in each family. We observed wide ranges from 4.1% (3/74, *BaseBridge*) to 65.6% (158/241, *Kmin*). One reason might be that there are few training MRA samples in some families. Another reason might be that the behavior of some MRAs is similar to those of benign applications, and thus only a few extracted common subsequences do not appear in benign applications, such as *DroidDream* and *BaseBridge*.

Fig. 4 shows the distribution of the malicious probability of the common subsequences for five families of MRAs. Many common subsequences have malicious probability of 5%–15%, meaning that they also appear in many training benign applications and are not applicable to be chosen into *MSC*. On the contrary, about half common subsequences have the malicious probability of 100%. Since their amount is enough, we only choose them as *MSC*, i.e., the common subsequences only appear in MRAs and do not appear in any benign application.

5.2. *Detection accuracy*

Fig. 5 shows the results of TP, TN, and DA for five families of MRAs. In the experimental results, 2 of 100 evaluated benign applications were detected as the MRA *Geinimi*. As a result, the TN is (98/100) = 98%. Also, since two false positives exist among 100 benign applications, its DA is (7 + 98)/(7 + 100) = 98.13%. On the other hand, one evaluated *DroidDream Light* and two evaluated ADRDs were not detected as MRAs. Hence, the TP of DroidDream Light is (1/2) = 50% and its DA is (1 + 100)/(2 + 100) = 99.02%. The TP of ADRD is (7/9) = 77.78% and its DA is (7 + 100)/(9 + 100) = 98.17%. The TP of

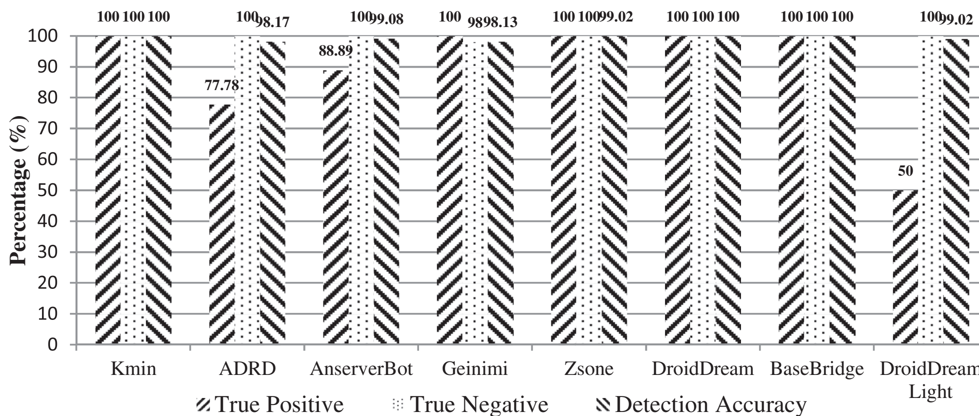

Fig. 5 – TP, TN, and DA for different families of MRAs.

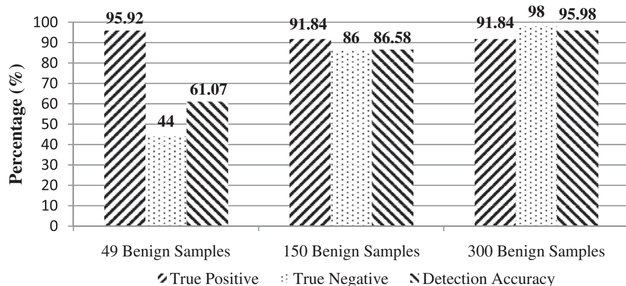

Fig. 6 – TP, TN, and DA for different numbers of training benign samples.

AnserverBot is (8/9) = 88.89% and its DA is (8 + 100)/(9 + 100) = 99.08%.

In overall, SCSdroid can effectively distinguish most benign applications and MRAs, albeit with a few positives and false negative. SCSdroid can achieve the overall detection accuracy of (9 + 7 + 8 + 7 + 6 + 4 + 3 + 1 + 98)/(9 + 9 + 9 + 7 + 6 + 4 + 3 + 2 + 100) = 95.97%.

5.3. *Effects of the number of training samples*

To further clarity the effects of the number of training samples on TP, TN, and DA, we first changed the number of training benign applications and then changed the number of training MRAs. First, we fixed the original 49 training MRAs and changed the number of training benign applications as 49, 150, 300 samples. Fig. 6 shows the results for different numbers of training benign samples. The more the training benign samples, the better the TN. The reason is that more non-discriminating common subsequences will be filtered when there are more training benign samples. Thus the obtained *MCS* is purer. On the contrary, the TP is slightly reduced. Four false negative occurred at using 150 and 300 benign samples. The reason might be insufficient training MRA samples and too many benign samples simultaneously.

Then the number of training MRA samples was changed. We only focused on *Kmin* and *Geinimi* because the numbers of MRAs in other families were too small. Fig. 7 shows the results for different numbers of training MRA samples. In this figure, the TP is unchanged, meaning that SCSdroid using fewer MRAs can still capture enough truly malicious subsequences. However, it is somewhat strange that the TN slightly decreased with more training MRA samples for *Geinimi*. This might be that the extracted common subsequences might involve some harmless system call sequences, such as repeatedly open files, read files, and close files. Since the common subsequence does not exist in the training benign samples, it is reserved in *MCS*. Once the evaluated benign application contains this subsequence, falsely detecting it as an MRA is possible. Hence, for some families of MRAs, if the number of training MRA samples increase, it is essential to increase the number of training benign samples to obtain a better TN.

Table 4 – Number of subsequences between different approaches.

Mechanism	Number of common subsequences (*CS*)	Number of malicious common subsequences (*MCS*)
Longest common subsequences	614	201
Fixed length subsequences (length = 15)	6206	2956

5.4. *Fixed length and longest common subsequences*

On a personal computer, Rozenberg et al. divided the system call sequence into fixed-length sequences and found their fixed-length common subsequences to detect malicious files (Rozenberg et al., 2011). They claimed that the subsequences with the fixed length equal to 15 had the best detection results. In this study the fixed-length subsequences and longest common subsequences are compared to verify the superiority of the latter.

Table 4 shows the numbers of common subsequences and the malicious common subsequences in different approaches for 29 training MRAs including *Kmin* (Encyclopedia), *Geinimi* (Strazzere & Wyatt, 2011), *DroidDream* (Lookout), *BaseBridge* (Android) and *DroidDream Light* (Security alert). The fixed-length approach generated more common subsequences and malicious common subsequences than SCSDroid which uses the longest common subsequences.

Next, TP, TN, and DA were evaluated by using these two malicious common subsequences. As Fig. 8 shows, SCSdroid has a high TN because it falsely detected only two benign applications among 100 evaluated benign samples as the

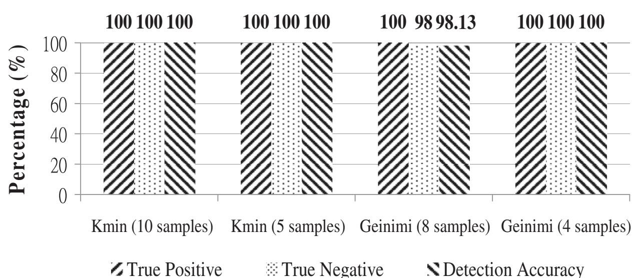

Fig. 7 – TP, TN, and DA for different numbers of training MRA samples.

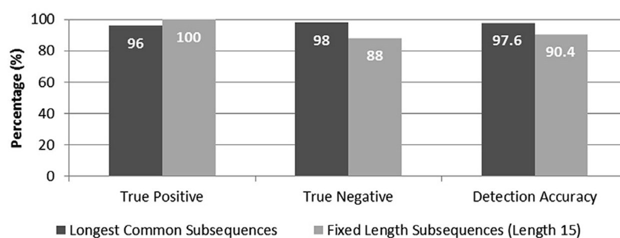

Fig. 8 – TP, TN, and DA between fixed length and longest common subsequences.

```
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)
```

Fig. 9 – The common subsequence of malicious behavior.

MRAs. However, the fixed length approach falsely detected 12 benign applications as MRAs. This is because the fixed length approach generated 2956 subsequences to detect MRAs. It is very likely to generate a higher FP. On the contrary, SCSdroid falsely detects one MRA as a benign one. The reason is that SCSdroid extracts fewer subsequences than the fixed length approach. In overall, SCSdroid has a much better DA than the fixed length approach.

5.5. *Case study*

Two further cases were investigated. In the first case the common subsequence of malicious behavior was studies, and in the second a false positive existing in our experiments was investigated.

5.5.1. *Malicious behavior*

Fig. 9 shows a piece of the common subsequence which has malicious behavior. This common subsequence was extracted from the MRA "*DroidDream*". The primary goal of the common subsequence is to allow an attacker to steal information. Referring to Fig. 9, the application initially checks whether the file "pref_config_setting.xml" exists in the file system. If the handheld was compromised by *DroidDream*, the file "pref_config_setting.xml" exists in the file system. This checking can avoid stealing information from the same device. In this

```
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\260l\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\0"..., 4096) = 4096
```

Fig. 10 – The case of false positive.

case, the file “pref_config_setting.xml” does not exist since our environment has not been compromised initially. Then, the application starts to query the remote server (184.105.245.17) which will silently establish the file “pref_config_setting.xml” to attack this device.

5.5.2. *False positive*

In our experiments, SCSdroid had two false positives, which were caused by the same common subsequence extracted from the MRA “*Geinimi*”. Fig. 10 contains the common subsequence that leads the false positive. It involves the preparation operations when the application activates, that is, it contains some operations before the system reads the substance of the application. Although these operations are the harmless behavior, they are not filtered in the filtering phase. This is because the common subsequence includes 116 system calls, i.e., it is very long. The preparation operations of most applications are very similar but with a few differences. Hence, if the extracted subsequences of evaluated MRAs involve preparation operations of many system calls, a few variation of system call sequences might happen and thus there is a small probability to lead the false positive.

6. Conclusions

This work proposes a MRA detection mechanism, SCSdroid, based on thread-grained system call sequences. SCSdroid first captures the system call sequence of each thread. Then SCSdroid utilizes the proposed LTMC mechanism to efficiently compare the pair of system call sequences in the same layer and adopts the LCS algorithm to effectively extract the common subsequences of MRAs. Finally, SCSdroid adopts Bayes Theorem to find the truly malicious common subsequences. According to these truly malicious common subsequences, the evaluated application can be effectively identified as a MRA or a benign one. Experimental results show that SCSdroid falsely recognizes only one malicious application among 25 MRAs and only two benign applications among 100 benign applications, achieving a high detection accuracy of 97.6%.

Some interesting observations can be summarized as follows. The first is that the behavior of some families of MRAs, such as DroidDream and BaseBrige, is very similar to that of benign applications. Only a few subsequences do not appear in benign applications and thus it is more difficult to detect these MRAs. The second is that for some families of MRAs, if more training malicious samples exist, it is essential to simultaneously increase the number of training benign samples to obtain a better TN. The third is that the longest common subsequences can extract purer subsequences to more correctly detect the MRAs, compared with the fixed length subsequences.

There are a number of directions for future study that will improve SCSdroid. First, SCSdroid can detect unknown MRAs which belong to the family we have already trained. However, it cannot detect MRAs belonging to an untrained family since the system call sequences for the family had not been obtained. Thus we want to extend SCSdroid to a more generic approach which can detect any untrained family of MRAs in the future. Second, SCSdroid can only record the behaviors which are automatically generated from applications. However, applications have more behaviors when users perform the operations. Therefore, it is necessary to consider how to grab the complete behaviors in the applications. Third, from our sample set, it is preliminarily verified that SCSdroid can detect MRAs very well. Currently some datasets with more Android malware have been collected (Zheng et al., 2012; Zheng et al.). Using these sample sets to validate the applicability of SCSdroid in real world scenarios is a good future direction.

Acknowledgments

This work was supported in part by National Science Council and Institute of Information Industry in Taiwan.

REFERENCES

Android.Adrd|symantec. Available at: http://www.symantec.com/security_response/writeup.jsp?docid=2011-021514-4954-99.

Android.Basebridge. Available at: http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99.

Blasing T, Batyuk L, Schmidt A-D, Camtepe SA, Albayrak S. An android application sandbox system for suspicious software detection. In: Proceedings of the 5th international conference on malicious and unwanted software (Malware 2010), Nancy, France 2010. p. 55–62.

Burguera I, Zurutuza U, Simin NT. Crowdroid: behavior-based malware detection system for Android. In: Proceedings of the 1st ACM workshop on security and privacy in smartphones and mobile devices, Chicago, IL, USA October 2011. p. 15–25.

Ehringer D. The Dalvik virtual machine architecture. Available at, avidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf; March 2010.

Enck W, Ongtang M, McDaniel P. Understanding Android security. IEEE Security & Privacy Magazine 2009;7(1):10–7.

Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification. In: Proceedings of the 16th ACM conference on computer and communications security, Chicago, IL, USA November 2009. p. 235–45.

Enck W, Gilbert P, Chun B-G, Cox LP, Jung J, McDaniel P, Sheth AN. 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, Vancouver, BC, Canada October 2010. p. 393–407.

Encyclopedia entry: Trojan:AndroidOS/Kmin.A. Available at: http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?Name=Trojan%3AAndroidOS%2FKmin.A.

Fuchs AP, Chaudhuri A, Foster JS. SCanDroid: automated security certification of Android applications. Technical report. University of Maryland; 2009.

Isohara T, Takemori K, Kubota A. Kernel-based behavior analysis for Android malware detection. In: Proceedings of the 7th international conference on computational intelligence and security, Sanya, Hainan, China December 2011. p. 1011–5.

Lookout mobile security technical tear down. Lookout Mobile Security.

Moser A, Kruegel C, Kirda E. Limits of static analysis for malware detection. In: Proceeding of annual computer security applications conference, Miami December 2007. p. 421–30.

Rozenberg B, Gudes E, Elovici Y, Fledel Y. A method for detecting unknown malicious executables. In: Proceedings of the 2011

IEEE 10th international conference on trust, security and privacy in computing and communications, Changsha, China November 2011. p. 190–6.
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/.
Security alert: Zsone Trojan found in Android market. Available at: https://blog.lookout.com/blog/2011/05/11/security-alert-zsone-trojan-found-in-android-market/.
Shabtai A, Kanonov U, Elovici Y, Glezer C, Weiss Y. 'Andromaly': a behavioral malware detection framework for Android devices. Journal of Intelligent Information Systems January 2011;38(1):161–90.
Strace. Available at: http://sourceforge.net/projects/strace/.
Strazzere T, Wyatt T. Geinimi trojan technical teardown. Lookout Mobile Security; 2011.
Vidas T, Votipka D, Christin N. All your droid are belong to us: a survey of current Android attacks. In: Proceedings of the 5th USENIX conference on offensive technologies, San Francisco, CA, USA August 2011.
VirusTotal – free online virus, malware and URL scanner. Available at: https://www.virustotal.com/.
Zheng M, Lee PPC, Lui JCS. ADAM: an automatic and extensible platform to stress test Android anti-virus systems. In: Proceedings of the 9th conference on detection of intrusions and malware & vulnerability assessment (DIMVA'12), Heraklion, Crete, Greece July 2012.
Zheng M, Sun M, Lui JCS. DroidAnalytics: a signature based analytic system to collect, extract, analyze and associate Android malware. Available at: http://arxiv.org/abs/1302.7212.
Zhou Y, Jiang X. Dissecting Android malware: characterization and evolution. In: Proceedings of the 33rd IEEE symposium on security and privacy, San Francisco, CA, USA May 2012. p. 95–109.
Zhou Y, Jiang X. An analysis of the AnserverBot Trojan. Technical report; September 2011.
Zhou W, Zhou Y, Jiang X, Ning P. Detecting repackaged smartphone applications in third-party Android marketplaces. In: Proceedings of the 2nd ACM conference on data and application security and privacy, San Antonio, TX, USA February 2012. p. 317–26.
Zhou Y, Wang Z, Zhou W, Jiang X. Hey, you, get off of my market: detecting malicious apps in official and alternative Android markets. In: Proceedings of the 19th network and distributed system security symposium, San Diego, CA, USA February 2012.

Ying-Dar Lin is professor of computer science, founder and director of the Network Benchmarking Lab (www.nbl.org.tw), and founder of the Embedded Benchmarking Lab (www.ebl.org.tw) at National Chiao Tung University. His research interests include design, analysis, implementation, and benchmarking of network protocols and algorithms; quality of service; network security; deep-packet inspection; P2P networking; and embedded hardware/software codesign. He is an IEEE fellow and on the editorial boards of IEEE Transactions on Computers, Computer, IEEE Network, IEEE Communications Magazine Network Testing Series, IEEE Communications Surveys and Tutorials, IEEE Communications Letters, Computer Communications, Computer Networks, and IEICE Transactions on Information and Systems. Contact him at ydlin@cs.nctu.edu.tw.

Yuan-Cheng Lai received the Ph.D. degree in computer science from National Chiao Tung University in 1997. He joined the faculty of the Department of Information Management at National Taiwan University of Science and Technology in 2001 and has been a professor since 2008. His research interests include wireless networks, network performance evaluation, network security, and content networking. He can be reached at laiyc@cs.ntust.edu.tw.

Chien-Hung Chen performed this research while at National Chiao Tung University. He is now an engineer at TrendMicro. His research interests include embedded system design, multimedia applications, and performance evaluation. He has an MS in computer science from National Chiao Tung University. Contact him at cfhung@cs.nctu.edu.tw.

Hao-Chuan Tsai received the BS degree in mathematics in 2002 from SooChow University, Taipei, Taiwan, the MS degree in computer science and information engineering in 2004 from Fu Jen Catholic University, Taipei, Taiwan, and the Ph.D degree in computer science and information engineering in 2010 from National Chung Cheng University, Chiayi, Taiwan. He is currently a Post Doc Researcher at Network Benchmarking Lab, National Chiao Tung University, Taiwan. His research interests include cryptography, image hiding, and malware detection algorithms.