Chen J, Alalfi MH, Dean TR *et al.* Detecting Android malware using clone detection. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 30(5): 942–956 Sept. 2015. DOI 10.1007/s11390-015-1573-7

# Detecting Android Malware Using Clone Detection

Jian Chen [1] (陈 健), Manar H. Alalfi [2], *Member, ACM, IEEE*, Thomas R. Dean [1], and Ying Zou [1] (邹 颖)

[1] *Department of Electrical and Computer Engineering, Queen's University, Kingston, K7L 3N6, Canada*
[2] *School of Computing, Queen's University, Kingston, K7L 3N6, Canada*

E-mail: jian.chen@queensu.ca; {alalfi, dean}@cs.queensu.ca; ying.zou@queensu.ca

Received March 27, 2015; revised July 27, 2015.

**Abstract** Android is currently one of the most popular smartphone operating systems. However, Android has the largest share of global mobile malware and significant public attention has been brought to the security issues of Android. In this paper, we investigate the use of a clone detector to identify known Android malware. We collect a set of Android applications known to contain malware and a set of benign applications. We extract the Java source code from the binary code of the applications and use NiCad, a near-miss clone detector, to find the classes of clones in a small subset of the malicious applications. We then use these clone classes as a signature to find similar source files in the rest of the malicious applications. The benign collection is used as a control group. In our evaluation, we successfully decompile more than 1 000 malicious apps in 19 malware families. Our results show that using a small portion of malicious applications as a training set can detect 95% of previously known malware with very low false positives and high accuracy at 96.88%. Our method can effectively and reliably pinpoint malicious applications that belong to certain malware families.

**Keywords** Android, malware, clone detection

## 1 Introduction

Smartphones and mobile devices are incredibly popular and widely adopted. The Android platform is a widely used open source operating system for mobile devices, and it accounts for over 84.7% of the global smart phone market in the second quarter 2014 according to International Data Corp. (IDC)①. Android is supported by the large number and wide variety of feature-rich applications. For example, there are already more than 1 400 000 applications (apps) in Google Play (December 2014) and there have been more than 50 billion downloads② from this pool of apps. These apps provide useful features, but also become a target for criminals and other miscreants and bring certain privacy and security risks. Malware, short for malicious software, is any software used to disrupt computer operation, gather sensitive information, or gain access to private computer systems③, such as viruses, worms, Trojan horses, and spyware.

Android is an easy target for attackers due to the open policy that allows anyone to publish apps in either official markets or third-party markets. Indeed, apps can often be downloaded from arbitrary web sites and installed. Decompiling the apps is also easy due to the structural characteristics of the app building process, making them vulnerable to forgery or modification attacks. A large quantity of malware has been found hidden in applications[1]. These applications are also known as repackaged applications, because they look like and work as existing genuine applications, but they contain new code that misbehaves in the background. Android accounted for 97% of all mobile malware in

Regular Paper
Special Section on Software Systems
The research is supported by the Ontario Research Fund of Canada.

①http://aptgadget.com/android-rose-to-above-84-7-in-smartphone-market-in-q2-2014/, Aug. 2015.

②http://en.wikipedia.org/wiki/Google_Play, March 2015.

③http://en.wikipedia.org/wiki/Malware, March 2015.

2014④. Therefore, Android malware detection has become increasingly important.

Researchers have explored different ways to identify these threats. Prior studies[2-3] showed that there are many cloned applications in mobile markets. It is straightforward to reverse-engineer an Android application and repackage it with additional malicious functionalities. If we can obtain the malicious code from the malware version of the application, then we might be able to use the malicious code as a malware pattern to identify other malware applications at the source code level. Hence, code clone detection technique could be an ideal technique for malware detection. Code clone detection is used to identify duplicated or similar code. Applying clone detection techniques to detect malware apps should provide more accurate results. There may also be additional benefits including detecting bugs, program understanding, and finding usage patterns.

Clone detection is an active research area and has been investigated to detect malicious software[4-5]. Various tools for detecting clones within and between source files have been developed by researchers with varying degrees of efficacy[6]. The NiCad[7] clone detection tool is one such tool that has proven effective in finding near-miss clones in source code. In this paper, we demonstrate the detection of malware in Android applications using a static clone detection method. Our hypothesis is that near-miss clone detection will provide a means of detecting mutations of known malicious code. We collect both malicious apps and benign applications. The malicious apps are divided into the training set and the evaluation set. The training set is used to form malware faulty signatures, which are used to find the malware in the evaluation set. We address the following research questions.

*RQ*1. Can we use clone detection techniques to generate a signature set of malicious code extracted from malicious apps?

The clone detection result presents the clone class information. A clone class is the clustering of a set of similar or identical code fragments. The same malware family should contain the same malicious code. Thus, the common code can be identified and extracted using clone detection. If we apply clone detection to apps within the same malware family, we can extract the source code based on the clone class information to generate the malware signature of that specific malware family.

*RQ*2. Can we use the malware signature set to detect similar malicious apps in the rest of the malicious set?

Clone detection identifies similar or identical code. Hence, we can use clone detection as a pattern matching engine to find the similar malware signature pattern in the evaluation set. We can achieve the highest accuracy at 96.88% in finding malware.

*RQ*3. Can we find the variants of one malware family in the entire malicious apps?

Malware can evolve over time. The changes can be incremental. Therefore, the new versions should contain similar code to the original. We use known malware to detect variants in the same family of malware using the near-miss clone detector. There are four variants of DroidKungFu malware family, and we can use DroidKungFu1 malware signature to identify the other variants.

In this paper, we make the following contributions:

1) conducting experiments to evaluate clone detection to identify Android malware;

2) evaluating the use of clone detection as a pattern matching engine for Android malware detection.

The detector is intended more for use by app markets rather than an approach that would be embedded inside an Android device. The particular apps we are interested in are those that repackage an existing app while adding a malware package. The malware is often also copied from other apps and included with a few, if any, changes.

This paper is organized as follows. Section 2 summarizes related studies in Android malware detection. Section 3 presents our approach in using clone detection to detect malware. Section 4 explains our study design and details the tools. Section 5 explains the research questions and describes the study results and findings. Section 6 lists some threats to the validity of the study. Finally, Section 7 summarizes and concludes the paper.

## 2 Related Work

In Android malware detection field, researchers have presented various approaches to detect malware by applying static analysis, dynamic analysis, and signature-based techniques.

Signature-based malware detection is a popular technique that is similar to our approach. Patterns are derived from known malware and used to identify a malware family. In general, these patterns are sequences of

④http://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/, Aug. 2015.

bytes of instructions[8]. Previous work[9] detected these syntactic patterns using semantic-preserving transformations and considered semantics-aware malware detection. Our approach can be seen as a form of signature generation. However, our signatures are at a higher level compared with templatized instruction sequences[9] and we directly compare the pattern at the source code level. Furthermore, the underlying signature matching techniques are also very different.

An example of approaches using signature-based technique is feature hashing detection. Juxtapp[10] is a code similarity detection system based on feature hashing among Android apps. Basic blocks are generated and labeled from XML representation converted from the DEX file. $k$-grams of opcodes from a code sequence within each basic block of an app are extracted using a moving window of size $k$ as features and then a hash function is applied to feature hash the $k$-grams into bit vectors. The Jaccard similarity between two bit vectors is computed to determine the similarity between two apps. Juxtapp is able to effectively identify various Android security issues, including buggy code, piracy, repackaging, and malware. DroidMOSS[2] applies fuzzy hashing technique to detect malware. App's instructions and author information are considered as features. Instead of processing the entire program instruction set, the instruction sequence is divided into smaller pieces to compute a hash value for each piece. All computed hash values are combined into the final fingerprint of an app. The similarity of two apps is determined by a similarity score, which is from the calculation of the edit distance between two fingerprints. DroidMOSS can efficiently identify the repackaged smartphone apps.

Both dynamic and static taint analyses have been proposed for tracking information flow in mobile applications. TaintDroid[11] is an example of dynamic taint analysis that tracks threat information flow by instrumenting the Dalivk virtual machine. FlowDroid[12] is a highly precise static taint analysis for Android applications.

Zhou and Jiang[1] collected more than 1 200 malware samples and aimed to systematize or characterize existing Android malware. Juxtapp[10] extracts the DEX file, and analyzes it for code similarity analysis among Android applications. Crowdroid[13] applies dynamic analysis to analyze application behaviours for detecting Android malware. Static analysis[14] is based on source code or binaries inspection looking at suspicious patterns. DroidMat[15] presents a static feature-based mechanism to provide a static analyst paradigm for detecting the Android malware.

DNADroid[3] is a tool that detects malicious applications through the establishment of a program dependence graph (PDG). A PDG represents the dependences of each operation in a program. DNADroid uses dex2jar⑤ to convert Dalvik byte codes to java byte codes so that DNADroid can utilize WALA⑥ to construct PDGs for every method. The detection of similarity between two applications is based on the comparison of matched PDG pairs. PDG technique is an often-used means in clone detection. It uses semantic information about the program, so the result has a better accuracy.

AnDarwin[16] extends DNADroid to avoid comparing apps pairwise and uses Android apps' semantic information to detect similar apps. AnDarwin starts with computing a PDG of the Android application. The semantic vectors are then extracted from the PDG to represent the application. AnDarwin identifies similar apps by clustering semantic vectors through more efficient algorithm locality-sensitive hashing (LSH)[17] to improve the scalability. One advantage of AnDrarwin is its reliability because it analyzes the apps only at the Java byte code level and does not depend on other information. In our approach, we use a further level of code, Java source code. We identify similar apps using clone detection technique at Java source code level.

Some novel semantic-based approaches have been developed to detect Android malware. Pegasus[18] is one of the new techniques that apply model checking on a new program representation, Permission Event Graph (PEG), to verify the policies specified by users. PEG presents the Android event dependencies and their API/permission level behaviours. It captures the semantic information about an app to model the effects of the event system. However, Pegasus needs users to write the policies of an app's behaviours using temporal logic formulas. Apposcopy[19] is another semantics-based tool, which constructs a new form of program representation, Inter-Component Call Graph (ICCG), with specific control and dataflow properties as a part of a malware detection analysis. It extracts semantic patterns and generates a unique signature matching that particular malware. The constructions of both PEG and ICCG are based on Java byte code and the semantic information is extracted from the Java byte

⑤https://code.google.com/p/dex2jar/, Mar. 2015.

⑥http://wala.sourceforge.net/wiki/index.php/Main_Page, Mar. 2015.

code. Our method is based on the pattern-matching approach, which is purely syntactic.

In this paper, we demonstrate a clone detection approach to detect malware in the Android platform. Clone detection technique has been investigated to detect malicious software. Walenstein and Lakhotia[5] showed that it is possible to find the evidence where parts of one software system match parts of another by comparing one malicious software family with another. Bruschi *et al.*[4] demonstrated a method to detect self-mutating malware (a particular form of code obfuscation) with clone detection techniques.

## 3 Proposed Approach

Android malware detection techniques use static, dynamic, or combined program analysis. Our approach uses static analysis to perform the detection of malware in Android applications. Our intention is to develop an approach that would identify all possible malicious apps to achieve a high recall and precision.

The general approach of the malware detection is shown in Fig.1.

APK files[7] are the files used to distribute Android applications and are used as input for a reverse engineering step to obtain Java source code files. The source files are then passed to the clone detection phase that comprises two phases: signature generation and signature matching. The NiCad clone detector is used in the clone detection phase. It is used to identify clone classes for these two phases.

In order to understand the general approach, we present basic information about the building processes of Android apps and code clone detection techniques.

### 3.1 Building Android Apps

Android is a Linux-based smart phone operating system designed by Google to run Android apps. Android apps are written in Java and distributed as APK files, which are similar to Java jar files. The APK file is a zip archive which contains all the code and data needed to install and run the app. The types of files included in the app are:

• DalVik Executable (DEX) file: the executable file resulting from the compilation of Java source code;

• Manifest file: a file containing app properties such as privileges, the app package file, and version;

• eXtensible Markup Language (XML) file: a file in which the user interface (UI) layout and values are defined;

• Resource file: a file containing resources required for app execution, such as images.

Fig.2 shows the series of steps for the building and packaging of files that make up the APK. First, the Java source code is compiled using the Java compiler (included in the Java development kit) producing a class file that runs on the Java virtual machine (JVM). The class file is converted to a DEX file using the dx converter included in the Android SDK. The DEX file runs on the Dalvik virtual machine.

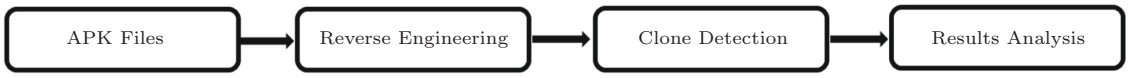


Fig.1. General process procedure of malware detection.

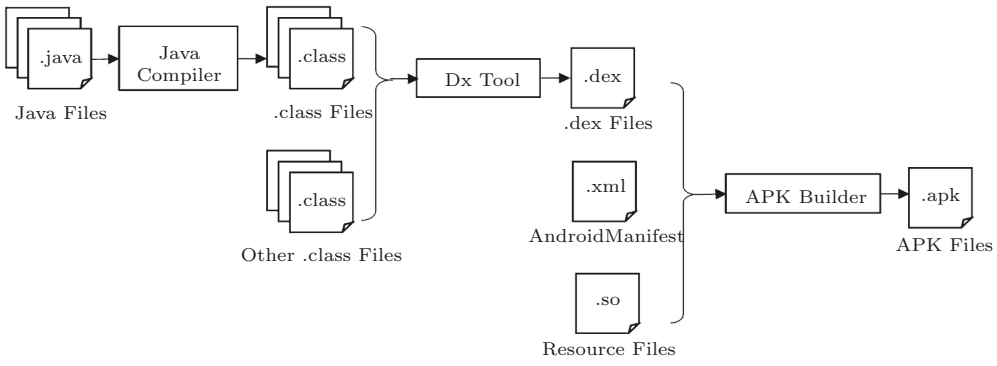


Fig.2. Android app building process. When we de-compile an APK file, it is from right to left.

[7]http://developer.android.com/google/play/expansion-files.html, Mar. 2015.

The manifest and other XML files needed for app execution are encoded in binary form. The manifest document contains a number of parameters that the Android framework needs in order to run the app. This includes the names of the activities, which are the different screens of the app, the permissions the app requires and the API version. Developers may also use this XML document to store any additional information the app may use. For example, advertising parameters are sometimes specified here. After that, the dex, XML, manifest, and resource files are packaged in an APK file, which is in ZIP format. The initially created APK file does not include the developer signature, which is needed in order to distribute it. The unsigned APK file can be self-signed with the developer's private key using Jarsigner. The developer's signature and the public key are then added to the APK file, which completes the Android app building process.

### 3.2 Code Clone Detection

Clones are segments of code that are similar according to some definition of similarity[20]. Roy *et al.*[6] demonstrated that NiCad can yield an outstanding result among the text-based techniques and tools. We adapt NiCad, a near-miss code clone detection tool, to help us identify the malicious code clones in malware apps. There are two sets of results generated by NiCad, each reported in both HTML and XML formats. First, the results of the comparison are reported as clone pairs. A clone pair is a pair of code fragments for which one is a clone of the other. Second, the clones in the input source are grouped into clone classes. Each clone class contains all the clones in the input which are similar and differ in the number of lines only up to the specified difference threshold. Both formats contain the source of the clones, specifying the degree of similarity, start and end line numbers of the clones found, and the size of the clones. NiCad provides the ability to find clones at various granularities (classes, functions, blocks, statements, etc.), with varying degrees of near-miss similarity (e.g., 70%, 80%, 90% or 100% similar). NiCad can operate in two modes. In the first mode, called standard mode, it is given a set of code files. NiCad identifies the clones and clusters these clones into classes. In the second mode, called increment mode, it is given two sets of files. NiCad finds elements in the second set that are similar within the threshold to the elements of the first set.

NiCad is based on TXL[21], which is a programming language specifically designed to support computer software analysis and source transformation tasks. TXL is a structural transformation and parser-based language. For instance, it parses the Java source code based on a TXL Java grammar.

### 3.3 Overview of Our Approach

Fig.3 shows the detailed clone signature generation process steps and malware signature matching steps of our approach. Our approach consists of three main stages: file selection, reverse engineering, and clone detection. Detailed description of the stages is presented in the following subsections.

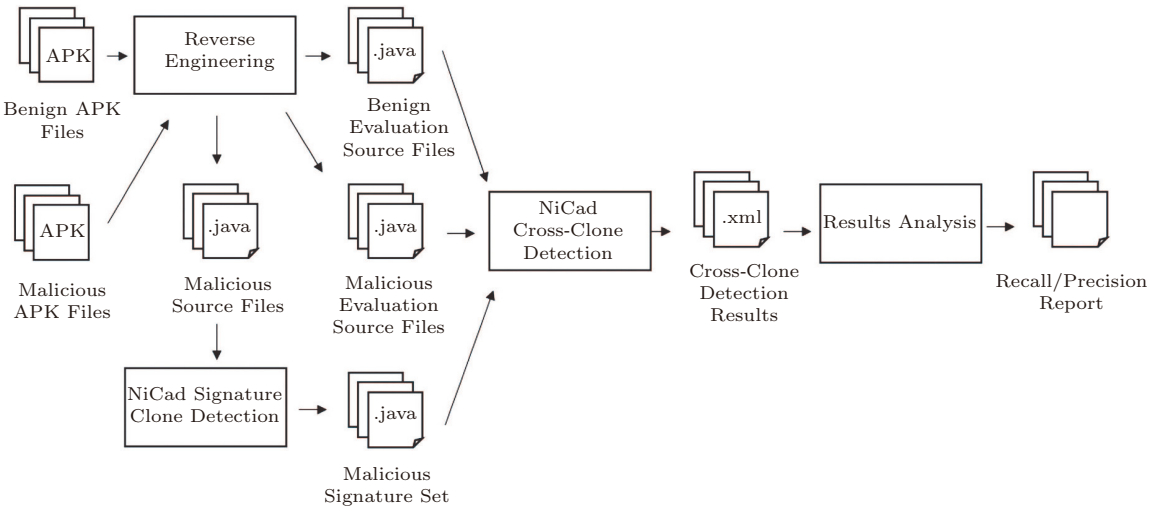


Fig.3. Clone signature process steps and malware signature matching steps of our approach.

#### 3.3.1 File Selection

We select two sets of Android applications: one contains known malicious applications, while the other contains known benign applications. The first set of malicious applications are divided into two sets, a training set and an evaluation set.

#### 3.3.2 Reverse Engineering

Reverse engineering of the Android apps is done primarily through the decompilation of the dex file, which can be decompiled into Java code. To obtain the Java code, tools such as dex2jar can be used to convert the Dalvik VM bytecode into JVM bytecode and a Java decompiler such as JD-CORE[⑧] can then be used to recover the Java code. This allows us to do the clone detection on high level code. Since the byte codes were optimized during the conversion from Java byte codes to Dalvik byte codes, the generated Java is not identical to the original Java source code. But since all of the malware apps were decompiled by the same converter and they were optimized by the same process, the source code recovered by the reverse engineering process is similar.

#### 3.3.3 Clone Detection

Two clone detection phases are used: signature generation and signature matching. In the first phase, NiCad is used to find the clone classes within the malicious training set. NiCad can compute clones at various levels of abstraction. For our approach, we search for clones at the function level. This allows partial matching of classes in two ways. First based on the threshold of similarity, the methods may be slightly different. But also, our classes with additional methods are also matched based on the subset of similar methods. That is the sets of similar malware. Different malware will be clustered into different classes of clones. We then take one exemplar from each of the clone classes to act as a signature for that class. This set of exemplars is called the signature set. In the second phase, NiCad is used in incremental mode to find clones of the members of the signature set in the malicious and benign evaluation sets. NiCad gives us the clone report, which we can do the further investigation of the malware analysis.

#### 3.3.4 Results Analysis

Finally, we analyze our results to evaluate our approach to see if we can find the malware Android applications effectively. We evaluate our approach through two parameters, recall and precision, based on the two evaluation sets (malicious evaluation set and benign evaluation set). Recall is the fraction of all relevant files retrieved by a query. It is a measure of how many documents are missed. It is defined as[22]:

$$recall = \frac{\text{number of relevant files} \bigcap \text{retrieved files}}{\text{number of relevant files}}.$$

In our context, the relevant files are the files in the malicious evaluation set. The retrieved files are those that NiCad identifies as clones of the signature set. Thus recall measures the fraction of the malicious evaluation set that was identified by NiCad.

Precision is the fraction of relevant documents retrieved by a query. It measures how many irrelevant documents are retrieved in error. It is defined as[22]:

$$precision = \frac{\text{number of relevant files} \bigcap \text{retrieved files}}{\text{number of retrieved files}}.$$

In our context, precision measures the fraction of identified malicious code that is in the malicious evaluation set. The precision is less than 100% when any of the benign files are identified as clones of the signature set.

$F$-measure, or accuracy, of a query is the harmonic mean of its precision and recall. It is a weighted average between the precision and the recall. It is defined as[22]:

$$F\text{-}measure = 2 \times \frac{recall \times precision}{recall + precision}.$$

## 4 Case Study Design

In this section, we present our experiment environmental setup, dataset collection, and data preprocessing phase.

### 4.1 Setup

To achieve this result, various resources are needed. We build our working environment on a Linux platform. We install all the following tools we used in this research on the Linux platform:

- Dex2jar: a tool that can convert .dex files into .class format;
- JD-CORE: a Java decompiler;
- NiCad[7]: a scalable, flexible code clone detection system based on TXL;
- TXL[21]: software analysis and source transformation programming language.

[⑧] http://jd.benow.ca, Mar. 2015.

### 4.2 Data Collection

In terms of dataset, we collect two groups of APK files: benign APK files and malicious APK files. The benign group APK files are from a third party market AppChina[9]. The malicious group APK files are from Android Malware Genome Project[10].

There are two reasons we choose AppChina as our benign group source. First, it is easier to obtain APK files from AppChina than to get those from Google Play. AppChina offers a client side assistant application, which runs on PC platform and assists to download APK files as many as you can. Second, we can assume the APK files from AppChina are clean. AppChina has a review mechanism, which will check each uploaded app from developers who want to publish their apps on AppChina before the apps are released.

We download 484 apps in total from 15 different categories of AppChina, such as browser, camera, communication, finance, news, social and so on. Only the top apps in each category are downloaded.

This malware group contains 1 260 Android malware samples in 49 malware families. The malware data samples are collected from August 2010 to October 2011. It has a very good coverage of existing Android malware. However, we could not fully use all the samples in our evaluation. Our approach is based on clone detection technique and we need more than one sample to form a clone class. Some malware families have only one sample, which is hard to form clone class, and also we separate the malware samples into two subsets: malicious extractions set and evaluation set. Thus, we need to choose the malware data samples that belong to a malware family which has multiple samples. Finally, total 1 170 APK files are included in our evaluation and they are from 19 malware families. We take a portion of each malware family as the malicious code extraction set and the rest are kept as the evaluation set. Table 1 shows the detail of separation of each malware family.

### 4.3 Pre-Processing

We use a pre-processing phase to process the APK files, so that we can take the advantage of clone detection technique. The first step of this phase is to generate the source code from the APK files. The APK file is in .zip file format. We extract classes.dex and then transform it to JAVA source code. The processing steps for each APK file are as following:

1) extraction of classes.dex file from the APK file;
2) using dex2jar to transform .dex file to .jar file;
3) using JD-CORE to decompile the .jar file to a Java source file.

**Table 1.** Number of Apps of the Two Subsets of the Malware Sample Set: Malicious Extraction Set and Evaluation Set

| Malware Family | Malicious Extraction Set | Evaluation Set | Total |
|---|---|---|---|
| ADRD | 10 | 12 | 22 |
| AnserverBot | 10 | 177 | 187 |
| BaseBridge | 10 | 112 | 122 |
| DroidDream | 8 | 8 | 16 |
| DroidDreamLight | 10 | 36 | 46 |
| DroidKungFu1 | 10 | 24 | 34 |
| DroidKungFu2 | 10 | 20 | 30 |
| DroidKungFu3 | 10 | 299 | 309 |
| DroidKungFu4 | 10 | 86 | 96 |
| Geinimi | 10 | 59 | 69 |
| GoldDream | 10 | 37 | 47 |
| jSMSHider | 8 | 8 | 16 |
| KMin | 10 | 42 | 52 |
| Pjapps | 10 | 48 | 58 |
| Plankton | 5 | 6 | 11 |
| SndApps | 10 | 5 | 10 |
| YZHC | 10 | 12 | 22 |
| zHash | 5 | 6 | 11 |
| Zsone | 6 | 6 | 12 |
| Total | 167 | 1 003 | 1 170 |

We write a script to automatically process all the APK files at once according to the above steps. The source code gathered from the pre-processing phase is also categorized into benign group and malicious group, and the malicious source code is categorized by malware families.

The decompiled Java source files do not completely conform with the standard Java grammar. The following figures show some examples of errors in the decompiled Java files, such as empty labels, using an keyword "finally" as a variable, using an empty type cast for an assignment, and dot number($a.2()$) as function. Fig.4 shows some examples of the abnormal decompiled Java source code.

```
label235:
localObject = finally;
long l1 = ()(1000.0F * paramFloat);
public static final f a = new a.2();
```

Fig.4. Examples of abnormal decompiled Java statements.

[9] http://www.appchina.com, Mar. 2015.
[10] http://www.malgenomeproject.org/, Mar. 2015.

Since NiCad by default uses the standard Java grammar, a modification of the grammar is necessary so that clone detector can parse the decompiled Java files. Specifically, we have added the ability to handle empty labels, variables with keyword names and empty type casts, and numbers as method names. Fig.5 shows the examples of the modification to allow NiCad to parse a label without a target.

```
define label_statement
  [label_name] ': [statement]
  | [label_name] ':   %added for the empty
    label.
end define

define assignment_expression
  [conditional_expression]
  |[unary_expression] [assignment_operator] [
    assignment_expression]
  |'finally %added for the keyword finally
end define
```

Fig.5. Examples of the modification of NiCad Java grammar in TXL.

During the pre-processing phase, 473 apps were successfully decompiled from a total of 484 downloaded apps. We successfully decompiled 1 180 863 Java files in total. Among the apps that were successfully decompiled, there are 375 apps containing at least one Java file with only null in the Java file, which represents 0.43% of the total decompiled Java files. Without any modification of Java grammar, NiCad can parse 89.3% of the total decompiled Java files. After the grammar modification, this percentage of successfully decompiled files increased to 99.82%.

## 5 Case Study Results

This section presents the results of our three research questions. For each question, we present its motivation, the analysis approach, and a discussion of our finding.

### 5.1 RQ 1: Can We Extract Malicious Code from Malware Apps to Generate a Malware Signature Set by Using Clone Detection Technique?

*Motivation.* Several researchers have explored the possibility of applying clone detection to detect malware[23-25]. Karademir *et al.*[23] used clone detection to identify JavaScript malware in Adobe Acrobat (PDF) files. Both [24] and [25] identify the code clone fragments at binary level. None of them detect the malware at Java source code level. If we can extract the malicious code contained in the malware, it not only will help to identify the malware, but also can help to remove the malware.

*Approach.* First, we need to obtain the malicious code from the source code to form a malicious signature as a malware pattern. We separate our data into three groups: benign set, malicious code extraction set, and testing set. In RQ1, we mainly focus on the malicious code extraction set, which still keeps the same directory structure as the testing set. Both of them are categorized into 19 malware families.

To generate the malware signature set, we apply the clone detection technique to the malicious code extraction dataset for each malware family separately. The NiCad clone detector is used in this step. In each malware family folder, there are several sub-folders which contain the decompiled Java source code of each APK sample, and they should contain the identical or similar malicious code fragments that belong to one malware family. Thus, NiCad can easily cluster the identical code into one class. The clone detection report of NiCad presents the clone classes within the malicious data extraction set. Based on the clone class information, we can extract the code of one member of each clone class to form a signature set for 19 different malware families and the code extracted from each malware family is saved as a new Java file called malewarefamilyname.java. Following these steps, we are able to obtain the malicious code from the sample set.

*Findings. Malicious code can form a clone class.* We examine the result of NiCad to each malware family in the extraction set to see if a set of malicious code fragments can form a clone class. Table 2 shows the preliminary result of this signature generation phase at 100% similarity level. Column 3 presents the number of clone classes generated from malicious classes. We only keep those clone classes across all samples within one malware family. In our malware code extraction experiment, we conduct sensitivity analysis. We run clone detection at several different similarity thresholds: 70%, 80%, 90% and 100% and three types of clones. Most of malware signature code is similar or identical. Therefore, threshold does not affect a lot. In terms of the malware code extraction, the clone detection results for different similarity thresholds and clone types of these various settings do not have significant differences. Since the malware code extraction is performed within one single malware family

**Table 2.** Experimental Results on the Malicious Extraction Dataset

| Malware Family | Number of APK | Number of Clone Classes | Similarity (%) |
|---|---|---|---|
| ADRD | 10 | 3 | 100 |
| AnserverBot | 10 | 8 | 100 |
| BaseBridge | 10 | 8 | 100 |
| DroidDream | 8 | 8 | 100 |
| DroidDreamLight | 10 | 1 | 100 |
| DroidKungFu1 | 10 | 25 | 100 |
| DroidKungFu2 | 10 | 72/18 | 100 |
| DroidKungFu3 | 10 | 2 | 100 |
| DroidKungFu4 | 10 | 3 | 100 |
| Geinimi | 10 | 3 | 100 |
| GoldDream | 10 | 11 | 100 |
| jSMSHider | 8 | 33 | 100 |
| KMin | 10 | 56 | 100 |
| Pjapps | 10 | 2 | 100 |
| Plankton | 5 | 2 | 100 |
| SndApps | 10 | 4 | 100 |
| YZHC | 10 | 4 | 100 |
| zHash | 5 | 180 | 100 |
| Zsone | 6 | 129 | 100 |

and the decompiled Java code of applications should contain the identical malware code, we only show the result in Table 2 at 100% similarity. Android app development often uses the third party or open source libraries. We take this into account when we generate the malware code signature from the decompiled code. The decompiled apps have a clear folder structure, where each directory name indicates the function of the code, for example, *ads*, *util*, and *sdk*. We skip the code if the code is within a library folder such as *sdk*. Though apps within one malware family contain same malicious code, these apps are not the same apps. The chance of containing same library is low. Thus it is difficult to form a clone class across all apps within the same malware family.

Giving an example, the first row of Table 1 is the malware family ADRD. We use 10 sample apps as the malicious extraction set. Accordingly, there are 10 subfolders under folder ADRD. We assign a sequence number for each subfolder, and in this way, we can easily identify the source file of each clone class. Fig.6 shows the partial result of the phase 1 clone detection. We only show two clone classes: class "1" and class "27" in this example. Obviously, class "27" is the malicious code clone class, which contains the ten identical code fragments from each different APK sample source code. Although class "1" is a clone class, which does not cover all the sub-samples, we do not take it as the malicious code clone class.

The clone report gives the location information of the identical code, such as file name, startline, and endline. Thus, we can extract each malicious code fragment of one member of each malicious clone class to a new java file called ADRD.java (see Fig.7).

The above example is extracted from com.xxx.yyy.UdateHelper.java from the ADRD malware family. ADRD is a Trojan that can open several system services. It can also upload infected cell phone's information (IMEI, IMSI, and version) to the control

```
<clones>
  <systeminfo processor="nicad3" system="_" granularity="unctions" threshold="0%" minlines="10" maxlines="2500"/>
  <cloneinfo npcs="4176" npairs="948"/>
  <runinfo ncompares="23612" cputime="77054"/>
  <classinfo nclasses="164"/>
- <class classid="1" nclones="3" nlines="108" similarity="100">
    <source file="/home/Training/ADRD/7/com/ophone/MiniPlayer/IPlayback.java" startline="106" endline="216" pcid="1003"/>
    <source file="/home/Training/ADRD/9/com/ophone/MiniPlayer/IPlayback.java" startline="106" endline="216" pcid="1815"/>
    <source file="/home/Training/ADRD/5/com/ophone/MiniPlayer/IPlayback.java" startline="106" endline="216" pcid="2938"/>
  </class>

<class classid="27" nclones="10" nlines="35" similarity="100">
  <source file="/home/Training/ADRD/8/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="4072"/>
  <source file="/home/Training/ADRD/6/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="1408"/>
  <source file="/home/Training/ADRD/4/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="144"/>
  <source file="/home/Training/ADRD/10/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="21"/>
  <source file="/home/Training/ADRD/3/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="3466"/>
  <source file="/home/Training/ADRD/1/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="3343"/>
  <source file="/home/Training/ADRD/7/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="737"/>
  <source file="/home/Training/ADRD/9/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="1545"/>
  <source file="/home/Training/ADRD/5/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="2672"/>
  <source file="/home/Training/ADRD/2/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="2216"/>
</class>
```

Fig.6. Example of phase 1 clone detection result of NiCad.

server and then receive its commands. In addition, it can download an installation file (.apk) to a specified directory of the SD card. Infected cell phones will generate significant network traffic and cause users extra expenses. The example code shows the code set, a new host to "10.0.0.172", can create a new folder, and save the download APK as "myupdate.apk". Based on the clone class information, we can extract the malicious code.

```
private boolean GetO(String paramString)
{
...
HttpHost localHttpHost = new HttpHost("
  10.0.0.172", 80, "http");
localDefaultHttpClient.getParams().
  setParameter("http.route.default-proxy",
  localHttpHost);
...
InputStream localInputStream =
  localHttpResponse.getEntity().getContent()
  ;
File localFile = new File(String.valueOf(
  savefilepath) + "myupdate.apk");
FileOutputStream localFileOutputStream = new
   FileOutputStream(localFile);
...
}
private void newFolder(String paramString)
{
try
{
File localFile = new File(paramString.
  toString());
if (!localFile.exists())
localFile.mkdir();
return;
}
catch (Exception localException)
{
System.out.println("新建目录操作出错");
}
}
```

Fig.7. ADRD.java.

*The identical or similar malicious code may not cross the entire malicious extraction set within one malware family.* This is a very interesting finding, as we assume that the malicious code should cross the whole extraction set within one malware family at beginning. However, we could not find any clone class within the DroidKungFu2 family across the ten extraction set instead of some clone classes from six of ten and other clone classes from the rest of four extractions set. In another word, the signature set is formed by two kinds of clone class: one kind is the clone class containing six pieces of similar or identical code, and the other kind is the clone class containing four pieces of similar or identical code. Thus, we extract the code from both different malicious sample clone classes. In Table 2, we show the numbers from different sample clone classes. For DroidKungFu2, 72 clone classes can be clustered from six sample sets and 18 clone classes can be formed from the rest four sample sets.

### 5.2 RQ 2: Can We Use the Malware Signature Set to Detect the Malware Apps in the Rest of the Malicious Set?

*Motivation.* Clone detection is to identify similar or identical code. Hence, we can use clone detection as a pattern matching engine to find the similar malware signature pattern in the evaluation set.

*Approach.* NiCad has two modes of clone detection: standard mode clone and incremental mode clone. The standard mode gives NiCad a single source folder to examine and all source files inside this folder are examined for clones, which is the way used in RQ1. The second incremental mode compares two separate folders of source code to find code clone pairs between the two systems; no clones are detected within the single folders in this mode. When testing in incremental mode, NiCad is run to compare a "malware" and a "testing/-malwarefamliy" folder. The malware folder contains the 19 malware java files of known malware and the "testing/malwarefamliy" folder represents the evaluation set of each malware family's decompiled java source files.

Precision, recall, and $F$-measure values are used to evaluate the clone detection means of malware detecting.

*Findings. NiCad can cluster the malware file and the evaluation set into clone class. In other word, our clone detection technique can detect successfully the malware.* This result proves the ability of clone detection technique in finding malware in Android platform. Fig.8 shows the partial incremental mode clone detection report for ADRD malware family. In this example, 13 files are clustered into one clone class, and one

```
−<class classid="1" nclones="13" nlines="35" similarity="100">
    <source file="../malware/ADRD.java" startline="1" endline="45" pcid="44"/>
    <source file="../Testing/ADRD/40156a176bb4554853f767bb6647fd0ac1925eac/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="364"/>
    <source file="../Testing/ADRD/f2bed732c841a2fff018f9b9c971bbeb994565f0/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="10075"/>
    <source file="../Testing/ADRD/744c0c7091630fc5a999277b95dc0b1cf3b68153/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="6860"/>
    <source file="../Testing/ADRD/e173b320dd39b7c0991ccde5a48a9af532ad9715/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="6540"/>
    <source file="../Testing/ADRD/dc46fefc4cb3d0fc0fcaa4fba76e5d5c7538da2f/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="4117"/>
    <source file="../Testing/ADRD/a35937acc1a17eeff9a2015c6bbafddaaac44897/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="1500"/>
    <source file="../Testing/ADRD/f4fc04c1e1566c80875160236641cd5b84f7da57/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="10413"/>
    <source file="../Testing/ADRD/ffe62967b75aab56710110b26baa69acd47a81dd/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="9595"/>
    <source file="../Testing/ADRD/b999d381d96cc3b40edce7048543f8a7a3f0e79e/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="2727"/>
    <source file="../Testing/ADRD/8784ee14bd5f4e1ef31073cc42bde7fa6671da43/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="2989"/>
    <source file="../Testing/ADRD/e28fedd06e1805fa7eaf0a1d26a45106746cecfd/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="105"/>
    <source file="../Testing/ADRD/82552e838199f9a2d4a537588fb77ecbc44a3ea8/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="5245"/>
</class>
```

Fig.8. Phase 2 incremental mode clone detection result of NiCad.

file is from the malware signature set, which is "malware/ADRD.java". From Table 1, we know there are 12 malware app samples in the evaluation set, and the result of 12 malware files plus one malware signature file matches the clone detection result in the example. Thus, we can use one malware family signature to detect all the same malware family apps in this example. It demonstrates that the clone detection technique can be as a pattern matching engine to detect the malware. The results of incremental mode clone detection between the signature set and the evaluation set are shown in Table 3.

**Table 3.** Experimental Results on the Malicious Evaluation Dataset

| Malware Family | Number of Total Malicious Apps | Number of Detected Malware | Similarity (%) |
|---|---|---|---|
| ADRD | 12 | 12 | 100 |
| AnserverBot | 177 | 175 | 100 |
| BaseBridge | 112 | 77 | 100 |
| DroidDream | 8 | 7 | 100 |
| DroidDreamLight | 36 | 9 | 100 |
| DroidKungFu1 | 24 | 23 | 100 |
| DroidKungFu2 | 20 | 20 | 100 |
| DroidKungFu3 | 299 | 298 | 100 |
| DroidKungFu4 | 86 | 78 | 100 |
| Geinimi | 59 | 59 | 100 |
| GoldDream | 37 | 37 | 100 |
| jSMSHider | 8 | 8 | 100 |
| KMin | 42 | 42 | 100 |
| Pjapps | 48 | 39 | 100 |
| Plankton | 6 | 5 | 100 |
| SndApps | 5 | 5 | 100 |
| YZHC | 12 | 12 | 100 |
| zHash | 6 | 6 | 100 |
| Zsone | 6 | 6 | 100 |
| Total | 1 003 | 918 | |

*Clone detection technique can achieve a very high accuracy in finding malware.* Our experiment is mainly evaluated through two parameters, recall and precision, based on the two evaluation sets (malicious evaluation set and benign evaluation set).

The result of the evaluation sets is shown in Table 4. These results are calculated against all 1 003 malicious apps and 473 benign apps. We set the similarity threshold at 100% for type 1 and type 2, and set it to 70% for type 3. As we know, type 1 clone is exact clone, and type 2 is renamed clone, so we set threshold at 100% that can yield a better precision. Type 3 clone has more alteration on code, so we set threshold at 70% to allow a better recall. A significant portion of the files in the malicious testing set (91%) are detected only with type 1 clone detection. Type 2 clone detection improves the detection a little bit, and type 3 does not improve the result of clone detection if we compare the type 3 result with the type 2 result. In terms of accuracy, the overall best detection was found to be type 2 clone detection at the accuracy of 96.88%. Table 4 is the result of applying incremental clone detection to the evaluation set. Since type 1 is too strict, we could not identify more malware apps. Since type 3 is too loose, we lost the precision.

### 5.3 RQ 3: Can We Find the Variants of One Malware Family in the Entire Malicious Apps?

*Motivation.* Malware also evolves over time. However, the variant is from its original malware family, which means it may contain the similar code with the original malware. Hence, we can use one known malware family to detect its variant. We try to use known malicious code to identify the unknown malware.

*Approach.* In our data sample, DroidKungFu malware has several variants. They are DroidKungFu1, DroidKungFu2, DroidKungFu3, and DroidKungFu4. We use NiCad in the standard mode to detect if there exist clone classes among the extracted malicious code of DroidKungFu malware families. Then we use NiCad in incremental mode to detect the testing set to examine the cross detection, and use known malware to detect their variants.

**Table 4**. Experimental Results on the Evaluation Dataset

| Parameter | Number of Malicious Clones | Number of Benign Clones | Recall (%) | Precision (%) | Accuracy ($F$-Measure) |
|---|---|---|---|---|---|
| Type 1 (exact) | 918 | 3 | 91.52 | 99.67 | 95.42 |
| Type 2 (rename) | 948 | 6 | 94.51 | 99.37 | 96.88 |
| Type 3 (near-miss) | 948 | 384 | 94.51 | 71.17 | 81.19 |
| Total files | 1 003 | 473 | | | |

*Findings. The malicious code of variants changed a lot over time.* When we execute clone detection on the DroidKunFu malware family, where the extracted malicious code files include DroidKunFu1.java, DroidKunFu2.java, DroidKunFu3.java, and DroidKunFu4.java, the clone detection report indicates that only DroidKunFu1 and DroidKunFu2.java contain the identical or similar code and the number is limited. From Table 2, we know DroidKunFu1 has 25 pieces of code which are identical and DroidKunFu2 has 90 pieces of code in total, but NiCad can identify that they have four pieces of identical code. On the other hand, NiCad cannot identify any similar or identical code among DroidKunFu1, DroidKunFu2, DroidKunFu3, and DroidKunFu4. The variants of the Droidkunfu family do not share too much code between each other. They only keep some same basic functions in each variant. Fig.9 shows the snippet of DroidKungFu1 malicious signature. In this example, DroidKungFu1 and DroidKungFu2 keep the downlaodFile function and the onCreate function.

*It is possible to detect the unknown malware using signature set within the same malware family variants.* We use the previous variant malware signature set to detect the next version variant. For example, we execute incremental mode clone detection among DroidKungFu1 malicious signature and DroidKungFu2, DroidKungFu3, DroidKungFu4 evaluation sets. Next, DroidKungFu2 malicious signature set is used to do incremental mode clone detection. Table 5 shows the variants of incremental mode clone detection results. We can use DroidKungFu1 malicious signature to detect the most of DroidKungFu2 and DroidKungFu4 malware apps in the evaluation set and all the DroidKungFu3 malware apps within the evolutions set.

```
public static String[] downloadFile(Context
  paramContext, String paramString)
{
...
localHttpURLConnection = (HttpURLConnection)
  localURL1.openConnection();
...
str1 = localFile1.getName();
str2 = getPath(paramContext, "download");
localFile2 = new File(str2);
if (!localFile2.exists())
localFile2.mkdir();
if (!str2.contains("sdcard"))
...
}

public void onCreate(Bundle paramBundle)
{
super.onCreate(paramBundle);
LinearLayout localLinearLayout1 = new
  LinearLayout(this);
localLinearLayout1.setOrientation(1);
...
this.message.setTextSize(20.0F);
this.message.setLayoutParams(new
  LinearLayout.LayoutParams(-1, -2));
localLinearLayout1.addView(this.message);
LinearLayout localLinearLayout2 = new
  LinearLayout(this);
...
}
```

Fig.9. Snippet of DroidKungFu1 malicious code.

*The extracted malicious code does not cover the entire malicious code existing in the malware sample set.* From the result of previous finding, it seems that

**Table 5**. Experimental Results on the Variants Clone Detection

| Signature Set | DroidKungFu2 | DroidKungFu3 | DroidKungFu4 |
|---|---|---|---|
| DroidKungFu1 | 13/43% | 299/97% | 79/82% |
| DroidKungFu2 | | 299/97% | 79/82% |
| DroidKungFu3 | | | 0/0% |

the DroidKungFu2 malicious signature set has a better coverage than the other signature sets. DroidKungFu2 signature set does not have any common code with DroidKungFu3 or DroidKungFu4 signature set, but using DroidKungFu2 signature still can detect DroidKungFu3 and DroidKungFu4 malware. Thus, we examine the result further, and find that the clone classes between DroidKungFu2 signature set and DroidKungFu3 or DroidKungFu4 are totally different from the clone classes formed by DroidKungFu3 signature set or DroidKungFu4 signature set. Thus, the extracted malicious code for DroidKungFu3 or DroidKungFu4 is not the entire malicious code. We mentioned before that DroidKungFu2 malicious signature set is formed by two different groups of the same extraction sample set: one group contains six sample apps and the other group contains the rest of four sample apps. When extracting the malicious code, we need to take into account the clone classes that are not across entire sample set. The root cause of this is the quality of decompiled code. We found some files only contain a word "null", which means something went wrong when the decompiler tried to decompile this file.

## 6 Threats to validity

In this section, we discuss the threats to validity of our study, following common guidelines for empirical studies[26].

*Construct Validity Threats.* The threats concern the relation between theory and observations. One major issue of our method is that we only take Java code into account and the code quality highly relies on the quality of decompiler. Some malware apps contain enciphered payloads and they are not regular code. Our method is impossible to detect them. The code quality factor may prevent our method from finding all malicious code within one malware family. Another issue is that not all APK files can be decompiled. We downloaded 484 benign apps in total from AppChina, and 11 apps could not even be unzipped. To avoid or alleviate such issue, we could use some intermediate representation of Java byte code as the target code of clone detection in the future. Jimple[27] and Smali[28] are intermediate representations. We can obtain the intermediate code using exiting tools such as Dexpler[29]. Then, we add the corresponding grammars to NiCad, so that we are able to parse and apply clone detection to the code.

*Threats to Internal Validity.* They concern factors that can affect our results. Our clone detection method is a signature-based approach, which has a known limitation that it can only detect instances of known malware families. Though our method can detect variants in malware, the variants are limited to the threshold used for clone comparison. The malicious code extraction in our method is based on clone class information. To form a clone class, we need two or more pieces of identical or similar code fragments. If there is only one sample file, it cannot form a clone class. In our case, we could not extract the malware signature if there is only one malware app sample within one malware family. Thus, we have to eliminate some samples from the original malware sample set. We cannot guarantee the benign apps are 100% clean. Even Google Play store still has malware in it[30].

*Threats to External Validity.* These threats concern the possibility to generalize our results. Our malware data sample set is not the most up-to-date, and it only contains the malware sample from August 2010 to October 2011. Thus, we could not detect the latest malware. The benign data set is only from AppChina. We should test more benign apps against Google Play store and more other third markets. We cannot detect zero-day malware. Our method requires that we identify malicious code first.

## 7 Conclusions

In this paper, we applied a clone detection technique, a static analysis approach for detecting malware in Android mobile apps ecosystem. Malware that belongs to one family shares a common set of characteristic code, which can be clustered through the NiCad clone detector. We applied clone detect technique in both standard mode and incremental mode in our approach. The research aim of determining the feasibility of clone detection techniques in detecting Android malware was achieved by the clone signature on NiCad. Our experiments indicated that our approach can detect malware with high accuracy of 96.88%. Our method can effectively and reliably pinpoint malicious applications that belong to certain malware families.

## References

[1] Zhou Y, Jiang X. Dissecting Android malware: Characterization and evolution. In *Proc. the 2012 IEEE Symposium on Security and Privacy*, May 2012, pp.95-109.

[2] Zhou W, Zhou Y, Jiang X *et al.* Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proc. the 2nd CODASPY*, Feb. 2012, pp.317-326.

[3] Crussell J, Gibler C, Chen H. Attack of the clones: Detecting cloned applications on Android markets. In *Lecture Notes in Computer Science 7459*, Foresti S, Yung M, Martinelli F (eds.), Springer, 2012, pp.37-54.

[4] Bruschi D, Martignoni L, Monga M. Using code normalization for fighting self-mutating malware. In *Proc. Int. Symp. Secure Software Engineering*, Mar. 2006.

[5] Walenstein A, Lakhotia A. The software similarity problem in malware analysis. In *Proc. Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software*, July 2006.

[6] Roy C, Cordy J, Koschke R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 2009, 74(7): 470-495.

[7] Cordy J R, Roy C K. The NiCad clone detector. In *Proc. the 19th ICPC*, June 2011, pp.219-220.

[8] Griffin K, Schneider S, Hu X *et al*. Automatic generation of string signatures for malware detection. In *Proc. the 12th RAID*, Sept. 2009, pp.101-120.

[9] Christodorescu M, Jha S, Seshia S A *et al*. Semantics-aware malware detection. In *Proc. the 2005 IEEE Symposium on Security and Privacy*, May 2005, pp.32-46.

[10] Hanna S, Huang L, Wu E *et al*. JuxtApp: A scalable system for detecting code reuse among Android applications. In *Lecture Notes in Computer Science 7591*, Flegel U, Markatos E, Robertson W (eds.), Springer Berlin Heidelberg, 2013, pp.62-81.

[11] Enck W, Gilbert P, Chun B *et al*. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. the 9th USENIX Conf. Operating Systems Design and Implementation*, Oct. 2010, pp.1-6.

[12] Arzt S, Rasthofer S, Fritz C *et al*. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM SIGPLAN Notice*, 2014, 49(6): 259-269.

[13] Burguera I, Zurutuza U, Nadjm-Tehrani S. Crowdroid: Behavior-based malware detection system for Android. In *Proc. the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, Oct. 2011, pp.15-26.

[14] Christodorescu M, Jha S. Static analysis of executables to detect malicious patterns. In *Proc. the 12th Conference on USENIX Security Symposium*, Volume 12, Aug. 2003.

[15] Wu D, Mao C, Wei T *et al*. DroidMat: AnDroid malware detection through manifest and API calls tracing. In *Proc. the 7th Asia Joint Conference on Information Security* (*Asia JCIS*), Aug. 2012, pp.62-69.

[16] Crussell J, Gibler C, Chen H. AnDarwin: Scalable detection of semantically similar Android applications. In *Lecture Notes in Computer Science 8134*, Crampton J, Jajodia S, Mayes K (eds.), Springer Berlin Heidelberg, 2013, pp.182-199.

[17] Andoni A, Indyk P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proc. the 47th Symp. Foundations of Computer Science*, Oct. 2006, pp.459-468.

[18] Chen K Z, Johnson N M, D'Silva V *et al*. Contextual policy enforcement in Android applications with permission event graphs. In *Proc. the 20th NDSS*, Feb. 2013.

[19] Feng Y, Anand S, Dillig I *et al*. Apposcopy: Semantics-based detection of Android malware through static analysis. In *Proc. the 22nd ACM SIGSOFT Int. Symp. Foundations of Soft. Eng.*, Nov. 2014, pp.576-587.

[20] Baxter I D, Yahin A, Moura L *et al*. Clone detection using abstract syntax trees. In *Proc. International Conference on Software Maintenance*, Nov. 1998, pp.368-377.

[21] Cordy J. The TXL source transformation language. *Sci. Comput. Program.*, 2006, 61(3): 190-210.

[22] van Rijsbergen C J. Information Retrieval (2nd edition). Butterworth-Heinemann, Newton, MA, USA, 1979.

[23] Karademir S, Dean T, Leblanc S. Using clone detection to find malware in Acrobat files. In *Proc. Conf. the Center for Advanced Studies on Collaborative Research*, Nov. 2013, pp.70-80.

[24] Farhadi M R. Assembly code clone detection for malware binaries [M.A. Thesis]. Concordia University, April 2013. http://spectrum.libray.concorida.ca/977131, Nov.2013.

[25] Farhadi M R, Fung B C M, Charland P *et al*. BinClone: Detecting code clones in malware. In *Proc. the 8th Int. Conf. Software Security and Reliability*, June 30-July 2, 2014, pp.78-87.

[26] Yin R K. Case Study Research: Design and Methods. Sage Publications, 2014.

[27] Vallee-Rai R, Hendren L J. Jimple: Simplifying Java bytecode for analyses and transformations. Sable Technical Report 1998-4. Sable Research Group, McGill University, 1998.

[28] Gruver B. Smali: An assembler/disassembler for Android's dex format@ONLINE. http://code:google:com/p/smali/, July 2015.

[29] Bartel A, Klein J, Traon Y L, Monperrus M *et al*. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *Proc. ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, June 2012, pp.27-38.

[30] Gilbert D. Malware posing as official Google Play app found in....official Google Play Store. http://www.ibtimes.co.uk/malware-posing-official-google-play-app-found-official-google-play-store-1453409, July 2015.

**Jian Chen** received his M.S. degree in computer science from the Queen's University, Kingston, in 2014. He has worked as a software developer for many years. He is pursuing his Ph.D. degree at Queen's University.

**Manar H. Alalfi** is an adjunct assistant professor in the School of Computing at Queen's University, Kingston, and an assistant professor in the Software Engineering Department at Alfaisal University, Riyadh. Dr. Alalfi obtained her Ph.D. degree from Queen's University. She is specialized in software engineering and its synergy with diverse research areas including: model driven engineering (MDE) for Web applications security analysis, MDE for automotive systems, scientific software engineering, and mining software repositories.

**Thomas R. Dean** is an associate professor in the Department of Electrical and Computer Engineering at Queen's University and an adjunct associate professor at the Royal Military College of Kingston, Kingston. His background includes research in air traffic control systems, language formalization and five and a half years as a Sr. Research Scientist at Legasys Corporation where he worked on advanced software transformation and evolution techniques in an industrial setting. His current research interests are software transformation, web site evolution, and the security of network applications.

**Ying Zou** is a Canada Research Chair in software evolution. She is an associate professor in the Department of Electrical and Computer Engineering and cross-appointed to the School of Computing at Queen's University in Canada. She is a visiting scientist of IBM Centers for Advanced Studies, IBM Canada. Her research interests include software engineering, software reengineering, software reverse engineering, software maintenance, and service-oriented architecture.