

# Detecting Android Malware Using Clone Detection

Jian Chen<sup>1</sup>, Manar H. Alalfi<sup>2</sup>, Thomas R. Dean<sup>1</sup>, Ying Zou<sup>1</sup>

<sup>1</sup>*Department of Electrical and Computer Engineering, Queen's University, Kingston, Canada*

<sup>2</sup>*School of Computing, Queen's University, Kingston, Canada*

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

Received submit date

**Abstract** Android is currently the most popular smartphone operating system. However, Android also has the largest share of global mobile malware and the significant public attention has been brought to the security issues of Android. In this paper, we investigate the use of a near-miss clone detector to identify known Android malware. We collect a set of Android applications known to contain malware and a set of clean applications. We obtain the Java source code from 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 decompiled more than 1,000 malicious apps in 19 malware families. Our results show that a small portion subset extraction set produced 95% detection 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 An-

droid platform is a widely used open source operating system for mobile devices, and it accounts for over 84.7% percent of the global

smart phone market in the second quarter 2014 according to International Data Corp. (IDC). Android is supported by the large number and wide variety of feature-rich applications. For example, there are already more than 1,400,000 apps in Google Play(December 2014) and there has been more than 50 billion downloads[20] 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 systems, such as viruses, worms, Trojan horses, and spyware.

The Android is an easy target for attackers due to the open policy that anyone can publish Apps both in 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[24]. These applications are also known as repackaged applications, because they look and work as existing genuine applications, but they hide

*J. Comput. Sci. & Technol., Mon.. Year, ,*

inside new code that misbehaves in the background. Android accounted for 97% of all mobile malware in 2014. So Android malware detection has become increasingly important.

Researchers have explored different ways to identify these threats. Prior studies[23, 13] showed that there are many cloned applications on mobile markets. It is straightforward to reverse engineer an Android application and repackage it with additional malicious functionality. 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[7, 19]. Various tools for detecting clones within and between source files have been developed by researchers with varying degrees of efficacy[18]. The NiCad[12] clone de-

tection 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 address the following research questions:

**RQ1)** *Can we use clone detection techniques to generate a signature set of malicious code extracted from malicious apps?*

Clone detection result presents the clone class information, which are the clustering of similar or identical code. The same malware family should contain the same malicious code. Thus, the common code can be identified and extracted using clone detection.

**RQ2)** *Can we use the malware signature set to find similar malicious files in the rest of the malicious collection?*

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 testing set.

**RQ3)** *Can we find the variants of one*

*malware family?*

Malware also evolves over time. The changes are incremental, so the new versions should contain similar code to the original. Hence, we can use known malware to detect variants in the same family of malware. We use known malicious code to identify the unknown malware.

In this paper, we make the following important contributions:

1. Experiment evaluating clone detection to classify Android malware.
2. Investigation evaluating the use of clone detection as a pattern matching engine for Android malware detection.

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, 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 families. In general, these patterns are sequences of bytes of instructions[15]. 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 detection. However, our signatures are much higher-level compare to templatised 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[16] is a code-reuse detection scheme based on feature hashing. The unlabelled classes:dex files of apps are grouped based upon some predefined criteria to reduce the comparison overhead. K-grams of various opcode sequence patterns within each basic block of the program are considered as features. For example, they choose 5-gram as a moving window of size 5, which moves within each basic block to map and flag the features into a  $m$  bit vector.

Then the bit vectors are further combined into a feature metric to fingerprint each app. Juxtapp currently uses various predefined opcode sequences as features. This detection scheme is able to effectively detect various code reuse cases, including piracy and code repackaging, malware existence, and vulnerable code. Fuzzy Hashing based detection DroidMOSS[23] leverages specialized hashing technique, called fuzzy hashing. Instead of computing a hash over the entire program instruction set, a hashing value is computed for each local unit of opcode sequence of the classes. It uses a reset point to split long opcode sequences into small units and then concatenate all the hash values into a whole. In this way, it can localize the modification caused by repackaging. Also, DroidMOSS focuses on instructions' opcode part in order to be resilient against "operand string literal" based obfuscation. DroidMOSS can efficiently identify those pieces that were not touched by the repackager and works well when code manipulation was only performed at a few interesting points such as hard coded URLs.

Both dynamic and static taint analyses have been proposed for tracking information flow in mobile applications. TaintDroid[14] is an example of dynamic taint analysis that tracks threat information flow by instrumenting the Dalvik virtual machine. FlowDroid[5]

is a highly precise static taint analysis for Android applications.

DNADroid[13] 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 dex2jar[1] to convert Dalvik byte codes to java byte codes so that they can utilize WALA[3] 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.

Jiang et. al [24] collect more than 1,200 malware samples and aim to systematize or characterize existing Android malware. Juxtap[16] extracted the DEX file, and analyzed it for code similarity analysis among Android applications. Crowdroid[8] applied dynamic analysis to analyze application behaviours for detecting Android malware. Static analysis[10] is based on source code or binaries inspection looking at suspicious patterns. DroidMat[21] presented a static feature-based mechanism to provide a static analyst paradigm for detecting the Android malware.

In this paper, we demonstrate a clone detection approach to detect malware in the An-

droid platform. Clone detection technique has been investigated to detect malicious software. Walenstein et. al[19] shows that it is possible to find the evidence where parts of one software system match parts of another by comparing one malicious software family to another. Bruschi et al.[7] demonstrate a method to detect self-mutating malware (a particular form of code obfuscation) with clone detection techniques.

### 3 Approach

Android malware detection techniques use either static, dynamic, or a 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 app thus to achieve a high recall and precision.

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

APK files[4] 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 detection and cross-clone detection. The NiCad clone detector is used in the clone detection phase. It is used to identify



Fig. 1. General Process Procedure

clone classes for these two phases.

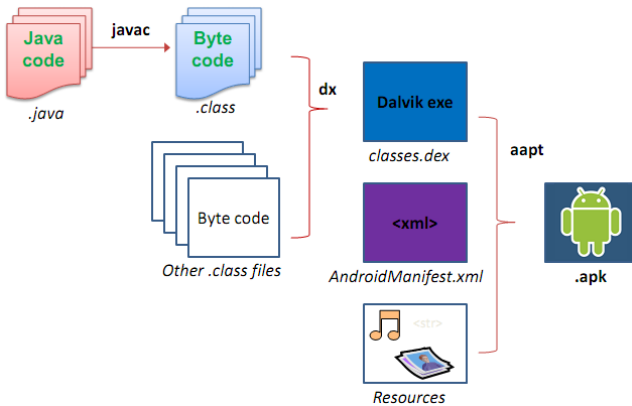


Fig. 2. Android app building process. When we de-compiled a apk file, it is from right to left.

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 applications (apps). Android apps are written in Java and distributed as **apk**[4] 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.

Figure2 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 Javac 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.

The manifest file, and other XML files needed for app execution are encoded in binary

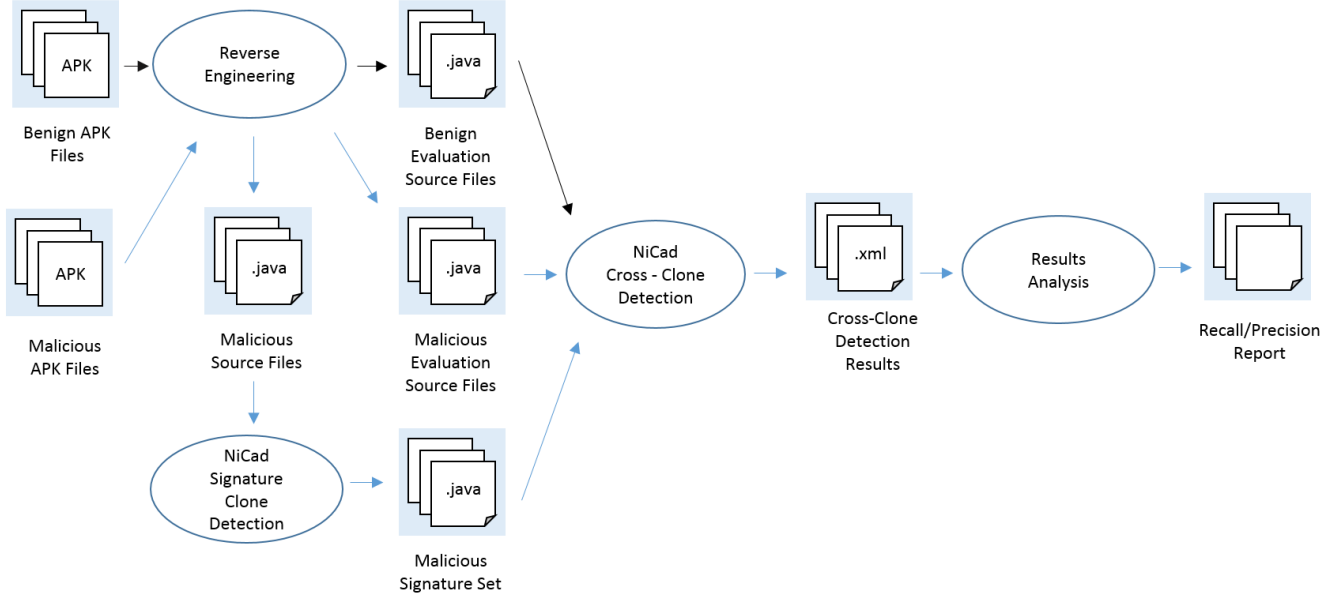


Fig. 3. Clone signature process steps and malware cross-clone detection steps of our approach

form. This XML 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[6]. 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 that differ in number of lines up to the specified difference threshold. 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 number of lines only up to the specified difference threshold. Both formats contain 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). It can be used either to find clones within a system, or cross-clones between two different systems.

NiCad is based on TXL[11], 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 parse the Java source code based on a TXL Java grammar.

### 3.3 Overview of approach

Figure 3 shows the detailed clone signature detection process steps and malware cross-clone detection 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:

#### 3.3.1 File Selection

We select two sets of Android applications, one containing 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 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[2] 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, thus, the generated Java is not identical to the original Java source code. But since all of the malware was optimized by the same converter, the source code recovered by the reverse engineering process is similar.

#### 3.3.3 Clone Detection

Two clone detection phases are used: signature detection and cross-clone detection. The first phase, NiCad is used to find the clone



classes within the malicious training set. That is the sets of similar malware. Different malware will be clustered into different classes. 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. The second phase, NiCad is used in cross-clone detection mode to find clones of the members of the signature set in the malicious and benign evaluation sets. The NiCad gives us the clone report, which we can do the further investigation of the malware analysis.

### 3.3.4 Results Analysis

Finally, we analyse 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 [17]:

$$recall = \frac{|\text{relevant files} \cap \text{retrieved files}|}{|\text{relevant files}|}$$

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 frac-

tion 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 [17]:

$$precision = \frac{|\text{relevant files} \cap \text{retrieved files}|}{|\text{retrieved files}|}$$

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

The 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 [17]:

$$F = 2 * \frac{recall * precision}{recall + precision}$$

## 4 Case Study

In this section, we present our experiment environment set up, dataset collection, and data pre-processing phase.

### 4.1 Experiment Environment

To achieve this result, various resources are needed. First, we need a *Safe Environment*: a Linux platform environment, which provides a safe environment for testing mobile

environments, including devices, applications, and supporting infrastructure. We install all the following tools we used in this research on the Linux platform.

- Dex2jar[1]: is a tool that can convert .dex files into .class format.
- JD-CORE[2]: is a Java decompiler.
- NiCad[12]: a scalable, flexible code clone detection system based on TXL.
- TXL[11]: software analysis and source transformation programming language.

## 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*. The malicious group APK files are from Android Malware Genome Project.

There are two reasons we choose AppChina as our benign group source. First, It is easier to obtain APK files from AppChina than 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

*J. Comput. Sci. & Technol., Mon.. Year, ,*

review mechanism, which will check each uploaded apps from developers who want to publish their apps on AppChina before the apps are released.

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

This malware group contains 1260 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, also we separate the malware samples into two subset: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 1170 APK files are included in our evaluation and they are from 19 malware families. We took a portion of each malware family as the malicious code extraction set and the rest are kept as the evaluation

**Table 1.** Malware sample set divided into two sub 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
<b>Total</b>	167	1003	1170

set. Table 1 shows the detail of separation of each malware family.

### 4.3 Reverse Engineering

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* then transform it to JAVA source code. The processing step for each APK file 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.

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

The decompiled Java source files do not 100% conform with the standard Java Grammar. The following listing shows some examples of errors in the decompiled Java files, such

*J. Comput. Sci. & Technol., Mon.. Year, ,*

as empty labels, using the keyword(***finally***) as a variable, using and empty type cast for an assignment, and dot number(*a.2()*) as function.

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

Listing 1. Example of abnormal decompiled Java statements

Since NiCad by default uses the standard grammar, a modification of NiCad Java grammar was necessary so that clone detector can parse the decompiled Java files to detect the similar code.

## 5 Case Study Results

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

**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 apply clone detection to detect malware. However, none of them detect the malware at 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 data set 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 are the decompiled java source code of each APK sample, they should contain the identical or similar malicious code belong to one malware family. Thus, the 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 each malware family is saved as a new java file called *malwarefamilyname.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 the malicious code can form a clone class. Table 2 shows the preliminary result of this signature detection phase. Those numbers are the malicious code clone classes. We only keep those clone classes that across all samples within one malware family.

Giving an example, the first row of table 1 is the malware family *ADRD*. We use ten sample apps as the malicious extraction set. Accordingly, there are ten sub folders under folder *ADRD*. We assign a sequence number for each sub folder, in this way, we can easily identify the source file of each clone class. Figure4 shows the partial result of the phase one 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 fragment 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*, *endline*. Thus, we can extract each malicious code fragment of one member of each malicious clone class to a new java file

**Table 2.** Experiment results on the malicious extraction data set

Malware Family	APK	Clone Class	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%

```

<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/jian/ELEC876/Training/ADRD/7/com/ophone/MiniPlayer/IPlayback.java" startline="106" endline="216" pcid="1003"/>
  <source file="/home/jian/ELEC876/Training/ADRD/9/com/ophone/MiniPlayer/IPlayback.java" startline="106" endline="216" pcid="1815"/>
  <source file="/home/jian/ELEC876/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/jian/ELEC876/Training/ADRD/8/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="4072"/>
  <source file="/home/jian/ELEC876/Training/ADRD/6/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="1408"/>
  <source file="/home/jian/ELEC876/Training/ADRD/4/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="144"/>
  <source file="/home/jian/ELEC876/Training/ADRD/10/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="21"/>
  <source file="/home/jian/ELEC876/Training/ADRD/3/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="3466"/>
  <source file="/home/jian/ELEC876/Training/ADRD/1/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="3343"/>
  <source file="/home/jian/ELEC876/Training/ADRD/7/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="737"/>
  <source file="/home/jian/ELEC876/Training/ADRD/9/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="1545"/>
  <source file="/home/jian/ELEC876/Training/ADRD/5/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="2672"/>
  <source file="/home/jian/ELEC876/Training/ADRD/2/com/xxx/yyy/UpdateHelper.java" startline="34" endline="78" pcid="2216"/>
</class>

```

Fig. 4. The phase one clone detection result of NiCad.

called *ADRD.java*.

```

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");
}

```

code/ADRD.java

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 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.

**The identical or similar malicious code may not across the entire malicious extraction set within one malware family.** This is very interesting finding, as we assume that the malicious code should across 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. Thus, we extracted the code from both different malicious sample clone class. 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.

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

*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 and cross clone. The stan-

*J. Comput. Sci. & Technol., Mon.. Year, ,*

dard mode gives NiCad a single source folder to examine all source files inside this folder are examined for clones, which is the way used in RQ1. The second cross clone 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 cross clone mode, NiCad is run to compare a "malware" and a "testing/*malwarefamily*" folder. The malware folder contains the 19 malware java files of known malware and the "testing/*malwarefamily*" folder represents the evaluation set of each malware family 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. Figure 5 shows the partial cross clone detection report for ADRD malware family. In this example, 13 files are clustered into one clone class, one file is from the malware signature set, which is "malware/ADRD.java".

From table 1, we know there are 12 mal-



**Table 3.** Experiment results on the evaluation data set

Malware Family	Evaluation APK	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%
<b>Total</b>	<b>1003</b>	<b>918</b>	

```

- <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\\dc46f9c4cb3d0fc0fcaa4fba76e5d5c7538da2f\\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\\e28fedd06e1805fa7eaf0a1d26a45106746cecf\\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. 5. The phase two cross clone detection result of NiCad.

ware apps samples in the evaluation set, and plus one malware signature file that is the cross clone result. Thus, we can use one malware family signature to detect all the same malware family apps in this example. It demonstrates the clone detection technique can be as a pattern matching engine to detect the malware. The results of cross clone detection between the signature set are shown in Table 4.

**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 1003 malicious apps and 473 benign apps. We set the similarity threshold at 100% for type I and type II, and set it to 70% for type III. A significant portion of the files in the malicious testing set (91%) are detected only with type I clone detection. Type II clone detection im-

prove the detection a little bit, and type III did not improve detection compare to Type II. In terms of accuracy, the overall best detection was found to be type II clone detection at accuracy of 96.88%.

*RQ3) Can we find the variants of one malware family?*

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, *Droid-KungFu* malware has several variants, they are *DroidKungFu1*, *DroidKungFu2*, *Droid-KungFu3*, and *DroidKungFu4*. We use NiCad at the standard mode to detect if there exist clone classes among the extracted malicious code of *DroidKungFu* malware families. Then we use NiCad at cross clone mode to detect the testing set to examine the cross detection, use

**Table 4.** Experiment results on the evaluation data set

Parameters	Malicious Clones	Benign Clones	Recall	Precision	Accuracy F-Measure
type-I(exact)	918	3	91.52%	99.67%	95.42
type-II(rename)	948	6	94.51%	99.37%	96.88
type-III(near-miss)	948	384	94.51%	71.17%	81.19
<b>Total Files</b>	<b>1003</b>	<b>473</b>			

know malware to detect the "unknown" malware.

*Findings* **The malicious code of variants changed a lot over time.** When we execute clone detection on the DroidKunFu malware family extracted malicious code files including DroidKunFu1.java, DroidKunFu3.java, DroidKunFu3.java, and DroidKunFu4.java, the clone detection report indicates that only DroidKunFu1 and DroidKunFu2.java contain the identical of similar code and the number are limited. From table 2, we know DroidKunFu1 has 25 pieces of code are identical and DroidKunFu2 has 90 pieces of code in total, but NiCad can identify they have 4 pieces of identical code. On the other hand, NiCad can not identify any similar or identical code among DroidKunFu1, DroidKunFu2, DroidKunFu3, and DroidKunFu4. Each variant does not share too much code each other. They only keep some same basic functions in each variant. In the following example, DroidKungFu1 and

DroidKungFu2 keep the downlaodFile function and onCreate function.

```

public static String [] downloadFile(Context
    paramContext, String paramString)
{
    ...
    localURLConnection = (
        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);

```

```
...
}
```

code/DroidKungFu1.java

**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 cross clone detection among DroidKungFu1 malicious signature and DroidKungFu2,DroidKungFu3,DroidKungFu4 evaluation sets. Next, DroidKungFu2 malicious signature set is used to do cross clone detection. Table 5 shows the variants cross clone detection results. We can use DroidKungFu1 malicious signature detect the most of DroidKungFu2 and DroidKungFu4 malware apps in the evaluation set and all the DroidKungFu3 malware apps within the evolutions set.

**The extracted malicious code do not cover the entire malicious code exist in the malware sample set.** From the result of previous finding, it seems the DroidKungFu2 malicious signature set has a better coverage than 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,

*J. Comput. Sci. & Technol., Mon.. Year, ,*

and found 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 are not the entire malicious code. We mentioned before, DroidKungFu2 malicious signature set is formed by two different group of the same extraction sample set, one group contains six sample apps and the other group contains the rest of four sample apps. When extract 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", it 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[22].

*Construct validity 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 highlight rely on the quality of decompiler. Some malware contain enciphered payloads and they

**Table 5.** Experiment results on the variants clone detection

Signature set	DroidKungFu2	DroidKungFu3	DroidKungFu4
DroidKungFu1	13	299	79
DroidKungFu2		299	79
DroidKungFu3			0

are not regular code. Our method is impossible to detect them. The code quality factor makes our method not able to find all malicious code within on malware family. Another issue is that not all APK files can be decompiled. We total downloaded 484 benign apps form *AppChina*, 11 apps could not even be unzipped.

*Threats to internal validity* concern factors that can affect our results. Our clone detection method is a signature-based approach, which has a known limitation is that it can only detect instances of known malware families. Though our method can detect malware variant, it only limits to its own variants. The malicious code extraction in our method is based on clone class information, if only one sample file, it can not form a clone class. Thus, we have to eliminate some samples from the original malware sample set. We can not guarantee the benign apps are 100% clean. Even Google Play store still has malware in it.

*Threats to external validity* concern the possibility to generalize our results. Our malware data sample set is not the most up to date,

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. Compare other dynamic detection method, we can not detect zero-day malware. Our method requires we extract the malicious code first.

## 7 Conclusion

In this paper, we apply a clone detection technique, a static analysis approach for detecting malware in Android mobile apps ecosystem. Malware that belong to one family share a common set of characteristic code, which they can be clustered through NiCad clone detector. We apply clone detect technique in both standard mode and cross clone mode in our approach. The research aim of determining the feasibility of clone detection techniques in detecting script-based malware was achieved by the clone signature on NiCad. Our experiments indicate that our approach can detect malware

with high accuracy 96.88%. Our method can effectively and reliably pinpoint malicious applications that belong to certain malware families.

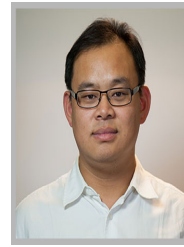
## References

- [1] dex2jar. <https://code.google.com/p/dex2jar/>.
- [2] Jd project. <http://jd.benow.ca>.
- [3] Watson libraries for analysis. [http://wala.sourceforge.net/wiki/index.php/Main\\_Page](http://wala.sourceforge.net/wiki/index.php/Main_Page).
- [4] Android. A. p. k. <http://developer.android.com/google/play/expansion-files.html>, 2013.
- [5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Outeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.
- [6] I.D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377, 1998.
- [7] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *In Proceedings of International Symposium on Secure Software Engineering. IEEE*, 2006.
- [8] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowddroid: Behavior-based malware detection system for android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM ’11, pages 15–26, New York, NY, USA, 2011. ACM.
- [9] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46, May 2005.
- [10] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM’03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

- [11] James R. Cordy. The txl source transformation language. *Sci. Comput. Program.*, 61(3):190–210, August 2006.
- [12] James R. Cordy and Chanchal K. Roy. The NiCad clone detector. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, ICPC '11*, pages 219–220, Washington, DC, USA, 2011. IEEE Computer Society.
- [13] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *ESORICS*, volume 7459 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2012.
- [14] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10*, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.
- [15] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-Cker Chiueh. Automatic generation of string signatures for malware detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection, RAID '09*, pages 101–120, Berlin, Heidelberg, 2009. Springer-Verlag.
- [16] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtap: A scalable system for detecting code reuse among android applications. In Ulrich Flegel, Evangelos Markatos, and William Robertson, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 7591 of *Lecture Notes in Computer Science*, pages 62–81. Springer Berlin Heidelberg, 2013.
- [17] C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2nd edition, 1979.
- [18] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470 – 495, 2009. Special Issue on Program Comprehension (ICPC 2008).
- [19] Andrew Walenstein and Arun Lakhotia. The software similarity problem in malware analysis. In *In Proceedings Dagstuhl Seminar 06301: Duplication, Re-*

*dundancy, and Similarity in Software*, 10 pp., Dagstuhl, 2006.

- [20] Wikipedia. Google play@ONLINE. [http://en.wikipedia.org/wiki/Google\\_Play](http://en.wikipedia.org/wiki/Google_Play), 2014.
- [21] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69, Aug 2012.
- [22] Robert K Yin. *Case study research: Design and methods*. Sage publications, 2014.
- [23] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.
- [24] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, 2012.



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



**Manar H. Alalfi** Dr. Alalfi is an Adjunct Assistant Professor in the School of Computing at Queen's University. Dr. Alalfi 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** Dr. Thomas 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.



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** Dr. 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.