2015

Android application evolution and malware detection

Shi, Wenjie

http://hdl.handle.net/2144/16308
Boston University
BOSTON UNIVERSITY
METROPOLITAN COLLEGE

Thesis

ANDROID APPLICATION EVOLUTION AND
MALWARE DETECTION

by

WENJIE SHI

B.S., Huazhong University of Science and Technology, 2013

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science

2015
Approved by

First Reader

Yuting Zhang, PhD
Assistant Professor of Computer Science

Second Reader

John Day, MSEE
Lecturer of Computer Science
Acknowledgments

I gratefully acknowledge the help of my supervisor Yuting Zhang, Assistant Professor of Computer Science, for her support and guidance during the whole research. Without her patient instruction and insightful criticism, the completion of this thesis would not have been possible.

I would like to thank Lou Chitkushev, Associate Professor and Associate Dean for Academic Affairs, and Tanya Zlateva, Associate Professor of Computer Science and Interim Dean of Metropolitan College & Extended Education, for providing me with a Graduate Assistant scholarship and actively participating my master’s thesis defense.

I would like to express my gratitude to my second reader John Day, Lecture in Computer Science, for his invaluable comments and suggestions. I’m also greatly indebted to the teachers at the Department of Computer Science: Suresh Kalathur, Jae Young Lee, Zoran Djordejevic, Madani Naidjate, who have offered me valuable courses during my study.

Last but not least, I owe much to my classmate Tandhy Simanjuntak, Graduate Assistant in Computer Science, who gave me his help and time in discussing with me and helping me work out my problems during the research. And I would like to thank the previous students, Nebiyu A. Feleke, Felex Rohrer, James Seibe and Craig Chaffee, who initially investigate the tools we used in the research.
Android has dominated the mobile market for a few years now. Meanwhile, Android has seen a sharper increase in malware. It is a matter of utmost urgency to find a better way to detect Android malware.

In this thesis, we use static code analysis to extract the android application security features and build classification models to detect Android malware. Our permissions-based classification model can achieve 96.5% accuracy, 97.2% TPR and 95.5% TNR with lower overhead. By using multiple security metrics, the detection rate increases to 99.3% accuracy, 99.5% TPR and 99% TNR.

Moreover, we investigate Android application security evolution. The data shows that more than half applications have security vulnerabilities or dangerous behaviors. Based on this result, we argue that there can be higher chance to impose update attack, where, the malware is contained in the updated version of a benign application. Our multiple-metrics based classification model is adapted to detect the update attack and can achieve similar or even better performance based on our initial results.
Contents

1 Introduction .............................................. 1

2 Background .............................................. 5
   2.1 Android Architecture ................................. 5
   2.2 Application Installation and Update Process .......... 6
      2.2.1 Application Installation ........................... 6
      2.2.2 Application Update ................................. 7
   2.3 Android Security Mechanisms ......................... 7
      2.3.1 Sandbox ......................................... 8
      2.3.2 Inter-Component Communication .................... 8
      2.3.3 Application Signing ............................... 8
      2.3.4 Permission Model ................................ 9
   2.4 Android Malware ..................................... 10
   2.5 Data Mining Techniques ............................... 11

3 Security Metrics Selection ............................... 14
   3.1 Vulnerabilities ..................................... 14
      3.1.1 Over-privileged Permissions ....................... 14
      3.1.2 Open Components ................................ 15
      3.1.3 Re-delegation ..................................... 15
   3.2 Dangerous Behaviors ................................. 16
      3.2.1 Hidden Files .................................... 16
      3.2.2 Dangerous Domains ............................... 17
3.2.3 Root Exploits ........................................ 17
3.2.4 Code Loading ........................................ 17

3.3 Sensitive Features ...................................... 18
3.3.1 Cost-Sensitive ....................................... 18
3.3.2 Personal Information ................................ 18
3.3.3 Sensitive Data Input Devices ....................... 19
3.3.4 Device MetaData ..................................... 19

3.4 Application Attributes ................................ 19
3.4.1 Application Size .................................... 20
3.4.2 Application Classes ................................ 20
3.4.3 Application Components ............................ 20
3.4.4 Permissions ........................................ 20

4 Cross-Versions Analysis Tool 21
4.1 Collecting Data ........................................ 22
4.2 Extractor ................................................ 22
4.3 Analysis ................................................ 25

5 Application Evolution 28
5.1 Vulnerabilities Analysis ................................. 31
5.2 Dangerous Behaviors Analysis ......................... 32
5.3 Application Attributes Analysis ....................... 34

6 Permissions-based Malware Detection 36
6.1 Approach Overview .................................... 36
6.2 Dataset ................................................ 37
6.3 Feature Vectors ........................................ 37
6.4 Classifiers Performance ............................... 40
6.5 Comparison ............................................ 44
7 Multiple Metrics Malware Detection

7.1 Approach Overview ............................................ 46
7.2 Dataset ....................................................... 47
7.3 Feature Vectors ................................................. 47
7.4 Classifiers Performance ...................................... 48
7.5 Comparison .................................................... 50

8 Update Attack Detection

8.1 Approach Overview ............................................ 52
8.2 Dataset ....................................................... 53
8.3 Feature Vectors ................................................. 53
8.4 Classifiers Performance ...................................... 54
8.5 Comparison .................................................... 58

9 Related work

9.1 Android Security ............................................... 60
9.2 Malware Detection Using Data Mining ...................... 60
9.3 Update Attack .................................................. 62

10 Future work

11 Conclusion

References

Vita
List of Tables

2.1 Detection results from four representative mobile anti-virus software (Zhou and Jiang, 2012) ........................................... 10

2.2 Confusion matrix .................................................................................. 12

5.1 Vulnerabilities evolution analysis ............................................................... 31

5.2 Dangerous behaviors evolution analysis .................................................. 33

5.3 Application metadata evolution analysis ................................................... 34

6.1 20 requested permissions with the highest percentage difference in malware and benign applications .............................................. 38

6.2 20 used permissions with the highest percentage difference in malware and benign applications ...................................................... 39

9.1 Performance comparison among different research groups ....................... 61
# List of Figures

2-1 Android system architecture (Android Developers, ) .......................... 5

3-1 Re-delegation ................................................................. 16

4-1 Cross-versions analysis tool framework ................................. 21

5-1 Application evolution original metrics .................................. 29

5-2 Variation in application evolution ...................................... 30

6-1 Accuracy of using the requested permissions ......................... 40

6-2 TPR of using the requested permissions .............................. 41

6-3 TNR of using the requested permissions .............................. 41

6-4 Accuracy of using the used permissions ............................. 43

6-5 TPR of using the used permissions ................................. 43

6-6 TNR of using the used permissions ................................. 44

6-7 Performance comparison between the requested permissions and the
used permissions ............................................................... 45

7-1 Accuracy of multiple metrics malware detection .................... 48

7-2 TPR of multiple metrics malware detection ......................... 49

7-3 TNR of multiple metrics malware detection ......................... 49

7-4 Performance comparison between using multiple metrics and the re-
quested permissions ............................................................. 51

8-1 Accuracy of update attack detection (the first experiment) ........ 54
8.2 TPR of update attack detection (the first experiment) ........... 55
8.3 TNR of update attack detection (the first experiment) ........... 55
8.4 Accuracy of update attack detection (the second experiment) . . . . 56
8.5 TPR of update attack detection (the second experiment) ........... 57
8.6 TNR of update attack detection (the second experiment) ........... 57
8.7 Performance comparison between update attack detection and using multiple metrics .................. 58
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>Accuracy</td>
</tr>
<tr>
<td>APK</td>
<td>Android Package file</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>Apps</td>
<td>Applications</td>
</tr>
<tr>
<td>DVM</td>
<td>Dalvik Virtual Machine</td>
</tr>
<tr>
<td>FN</td>
<td>False Negative</td>
</tr>
<tr>
<td>FP</td>
<td>False Positive</td>
</tr>
<tr>
<td>FPR</td>
<td>False Positive Rate</td>
</tr>
<tr>
<td>ICC</td>
<td>Inter-Component Communication</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>SMS</td>
<td>Short Messaging Service</td>
</tr>
<tr>
<td>TN</td>
<td>True Negative</td>
</tr>
<tr>
<td>TP</td>
<td>True Positive</td>
</tr>
<tr>
<td>TPR</td>
<td>True Positive Rate</td>
</tr>
<tr>
<td>UID</td>
<td>User Identifier</td>
</tr>
<tr>
<td>WIFI</td>
<td>Wireless Fidelity</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Android is a mobile operating system that is based on the Linux kernel. Its open source nature attracts the interest of many companies and developers. However, this open nature also contributes to many security threats. Last November, The Financial Times (Thomas, 2014) released a report that more than 40% of UK businesses were hit by mobile security breaches in the last year. The mobile security is a heated subject currently.

To provide greater functionality, Android developers publish new versions of the apps to keep their product alive. If the new version has better security than the old one? This is the question we want to investigate.

Moreover, in the mobile market, Android shares 87% of the global smartphone market in 2013. However, Android’s share of global mobile malware is an even larger number: 97% (Kelly, 2014). These malware are found in not only the official market, Google Play, but also in many third-party markets. Malware is the application which is harmful or potentially harmful to users or their devices. Currently, the signature technique of malware detection is not effective and has difficulty detecting zero-day malware currently, because it is so changeable.

Based on these motivations, many research groups focus on Android security evolution and malware detection. To show the security evolution, the researchers use some special metrics to see the evolution, such as, permissions (Wei et al., 2012). In malware detection, one of the research fields is using the data mining techniques to
detect the malware. Using data mining techniques can have zero-day malware detection, since an unknown application can be classified as a suspicious application or a benign application by using the built model. Recently, (Aung and Zaw, 2013) uses all the requested permissions as the feature vectors to build the classification model. The best performance is 91.6% accuracy, 91.6% TPR and 91.6% TNR. (Aafer et al., 2013) uses the called APIs as the feature vectors to build the model with the best performance of 96.5% accuracy, 97.2% True Positive Rate (TPR) and 95.5% True Negative Rate (TNR).

To explain the Android application and improve the malware detection, we use the crawler to download thousands of applications from the official market and the third-party market in a period time. In the application set, we find 292 applications from Google Play and 886 applications from the third-party market have multiple versions. We use these samples to investigate the Android application security evolution, a number of security metrics are defined based on the literature, we group them into four categories, vulnerabilities, malicious behaviors, sensitive APIs and application metadata. We find not all applications solve the potential security problems exposed in the old version. On the contrary, many applications have more vulnerabilities or the malicious behaviors in the updated version.

Then we use the classification algorithms to detect the malware. We compare the performance of using the requested permissions with the used permissions to do the classification. Different from (Aung and Zaw, 2013), we rank the permissions by percentage difference between malware and benign applications. Then we use four classifiers to find the best model. The similar best performance is achieved by using the top 40 requested permissions and the top 25 requested permissions of using the IBK classifier. The best performance of using the requested permissions is 96.5% accuracy, 97.2% TPR and 95.5% TNR. One of the benefits of using the requested
permissions approach, it is really fast.

Then we use more security metrics to do the malware detection and check if the performance can be improved. The multiple metrics include the vulnerabilities, the dangerous behaviors, the sensitive APIs besides the requested permissions. The best performance is 99.3% accuracy, 99.5% TPR and 99% TNR with IBK classifier. This model is not only applied in the Google Play market, but also in the third-party market.

Since the result of application evolution is not optimistic, we take a look at the update attack detection. Update attack is a new type of attack where the malicious code is contained in the update components of a legitimate application. As most applications are set to automatically update, this new attack is harder to detect and will induce more users to install the malicious code. We use machine learning classifiers to detect this attack based on the difference between application versions in various security metrics. And we also try to see if we use the old versions as the training dataset to build the classification model, it can detect the malware from the new versions more accurately. The answer is YES. When we use this method, the TNR reaches to ”1” in the Google Play market, which means our model classifies all the malware from the test dataset correctly.

In summary, our contributions include:

- We use multiple methods to improve the performance of malware classification. When we use multiple metrics to do the classification in the Google Play market, our best model can reach 99.3% accuracy, 99.5% TPR and 99% TNR, which is much higher than others work. What’s more, the applications sources are from both the official market and the third-party market. To the best of our knowledge, this’s the first time that permissions, APIs, vulnerabilities and dangerous behaviors have been used together to detect Android malware.
• We have implemented an Android cross-version application analysis tool and using this tool to investigate the application evolution in the Google Play and the third-party market. We find that the security issues have not gotten enough attention by developers. Consequently, the results are not optimistic. More than half of applications have one or more vulnerabilities or dangerous behaviors and many of them remain the same problems in the updated version.

• We propose two methods to detect update attacks, the malware detection rate increases to 100% and 99.7%, which proves that our models can classify the malware correctly.

The rest of this paper is organized as follows: Chapter 2 presents the Android architecture, installation and update process, security mechanisms, malware and some data mining techniques. Chapter 3 provides all metrics which are used to present an Android application. Then, chapter 4 shows the application evolution analysis of security. After that, chapter 5 illustrates the results of application evolution. Chapter 6 shows the performance of using the requested permissions and using the used permissions to do the classification. Chapter 7 provides the classification performance of using multiple metrics, followed by the detection of update attack in chapter 8. Chapter 9 discusses some related work. Finally, chapter 10 explains the future work and chapter 11 concludes this paper.
Chapter 2

Background

This chapter introduces the Android architecture, installation and update process, the Android security mechanisms, malware and some data mining techniques which are widely used in the malware detection.

2.1 Android Architecture

The Android operating system has four layers with five parts as shown in Figure 2·1. Each layer has its own functionality.

Figure 2·1: Android system architecture (Android Developers, )
Android has the Linux kernel at the bottom of the layers. This layer provides the basic function, such as IPC (inter-process communication), device management, memory management and so on.

Native libraries are in the second layer from the bottom. It provides a set of libraries written in C or C++. For example, SQLite database is a kind of native library, which is used to store the application data.

Android runtime is also in the second layer. Android uses the DVM (Dalvik Virtual Machine) sandbox to support its application sandbox. It’s optimized for low power and small memory, which is desirable for mobile devices. This layer also provides a set of core Java libraries.

The application framework is the third layer. It provides many higher-level services to the application layer.

Application layer is the top layer in Android architecture. Some pre-installed applications are installed on every device. Users can also install additional applications in this layer. The applications can come from the official market or the third-party markets.

2.2 Application Installation and Update Process

The Android application installation and update process mainly consist of three steps. After an APK file is downloaded and ready, the first step is to decide whether this application is a new application or an updated version. The second step is to assign the user id (UID) and the last step is to assign the permissions.

2.2.1 Application Installation

Android Operating System (OS) should verify that the APK file has not been modified by others before installation (Barrera et al., 2012). After this, the OS should check the manifest if the package name is different from other package names which have
been already installed, or if this package name is new, this installation is considered as an initial installation.

In the UID assignment step, sharing UID is checked in the manifest. If the sharing UID exists, Android checks other applications' shareUserId. If they match to each other, this application is assigned with the existing UID. If no applications match or no sharing UID in the manifest, a new UID is assigned to this application.

In the permission assignment step, if the UID is new, this UID will have all permissions requested in the manifest if the users approve. If the UID is shared, this application will not only have its own requested permissions, but also the permissions of other applications with the same UID.

2.2.2 Application Update

After validating APK file, if the package name is the same as the name of installed applications on the device, this application is considered as an updated version.

In the UID assignment step, the new version has the same UID as the old version.

In the permission assignment step, if the new version doesn’t ask for new permissions, no permissions would be shown on the user’s screen. Otherwise the user needs to grant the new permissions to the new version. After the permission assignment step, the old installed version will be removed and the new updated version will be installed. Most applications are set to automatically update.

2.3 Android Security Mechanisms

This section provides an overview of the Android security mechanisms. Android security focuses primarily on protecting user data, system resources and application isolation (Android Developers, ). To achieve these goals, Android provides the following features, sandbox, secure ICC (inter-component Communication), application signing and permission model.
2.3.1 Sandbox

Application sandbox is a means to isolate the applications from each other in the Android system by assigning a UID and a set of permissions.

When the application is installed on the device, it runs in its own sandbox and other applications cannot access or interfere. An application can only access its own files, unless other applications explicitly assign the access permissions to this application. For example, if the applications are created by the same developers, the developers can make these applications share the same UID, then these applications will run in the same sandbox and share the resources in that sandbox.

2.3.2 Inter-Component Communication

Android application consists of components. There are four kinds of components, activities, services, broadcasts and providers. Android platform provides a secure ICC that is similar to IPC to the Unix system.

ICC is provided by the binder mechanism which is in the middleware layer of Android. The binder is a remote procedure call that is from a custom Linux driver (Android Developers, ). ICC is achieved by intents. An intent is a message that shows the target with some data optionally. It can be used in explicit communication if it identifies the name of the receiver, or used in the implicit communication that let the receiver see if it can access this intent or not.

2.3.3 Application Signing

Application signing is used to ensure the application security. It creates a certification between developers and their applications.

Before placing an application into its sandbox, the application signing creates a relationship between the UID and the application. The applications couldn’t be run on the Android without signing. With the same UID, that is, running in the
same sandbox, the applications can share the permissions and communicate with each other.

By using application signing, the application update process can be simplified. Since different versions of the same application have the same certificate, the package manager can verify this certificate. Then, the old version is replaced, the new version can have the permissions already granted to the old version. What’s more, the application signing can also ensure that an application cannot communicate with another unless using the ICC. But if the author is the same, the author can use the same application signing to enable the direct communication among his/her applications.

2.3.4 Permission Model

The application is isolated when running in the sandbox. When it wants to access some sensitive features, such as camera, location, telephony, network. Android provides a permission model to achieve this goal.

Permissions mechanism is used to make some restrictions when the applications want to access the sensitive APIs of the operating system.

An application can declare which permissions it needs in the manifest. Before the application is installed on the device, the system will ask the users if they grant the permissions to this application. If the users agree to grant all requested permissions to the application, the installation continues, otherwise, the installation cancels. Unlike iOS, the user cannot choose which permissions they want to grant and which permissions they want to deny. Moreover, the application can get the permissions through the application signing.

The permissions have four levels, normal permissions, dangerous permissions, signature permissions and signatureOrSystem permissions. Normal permissions can be granted automatically; dangerous permissions are inferred to those granted by the users; signature permissions are granted within the same sandbox; signatureOrSys-
tem permissions are granted to pre-installed applications or the applications installed by the root.

2.4 Android Malware

While the Android OS is increasing its market share at an amazing speed, the Android malware also increase many times. The reasons may be the Android security framework and its open source nature.

A mobile malware survey (Felt et al., 2011a) shows that the incentives for writing malware include selling user information, stealing user credentials, making premium-rate calls and SMS and so on.

Now the most popular method to detect malware is based on the application’s signature, which is difficult to detect zero-day malware. In Table 2.1 shows the malware detection results from four representative mobile anti-virus software (Table 2.1), AVG Antivirus Free v2.9 (AVG), Lookout Security & Antivirus v6.9 (Lookout), Norton Mobile Security Lite v2.5.0.379 (Norton), and TrendMicro Mobile Security Personal Edition v2.0.0.1294 (Trend Micro).

<table>
<thead>
<tr>
<th>Detected Samples (out of 1260)</th>
<th>AVG</th>
<th>Lookout</th>
<th>Norton</th>
<th>Trend Micro</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>689</td>
<td>1003</td>
<td>254</td>
<td>966</td>
</tr>
<tr>
<td>%</td>
<td>54.7%</td>
<td>79.6%</td>
<td>20.2%</td>
<td>76.7%</td>
</tr>
</tbody>
</table>

**Table 2.1:** Detection results from four representative mobile anti-virus software (Zhou and Jiang, 2012)

In Table 2.1, the malware dataset has 1260 unique malware samples, which is also used in our research as malware dataset.
2.5 Data Mining Techniques

In the malware detection area, data mining is a frequently used technique. Especially, the classification algorithms are useful to classify an unknown application.

Classification is a supervised learning (Han et al., 2006). It has two steps. First, it builds a model based on the training dataset which has known class labels. Then, it uses the model to classify the new data with unknown class labels. In our research, a class label that equals "1" means the benign application while a class label that equals "2" means the malware.

In this research, we use four different classifiers in Weka (Hall et al., 2009), IBK, J48, Logistic and JRip, respectively. These four classifiers are used to in malware detection by many researchers.

**IBK (Aha and Kibler, 1991)** uses KNN (K-Nearest Neighbor) algorithm, is a lazy learning classifier built after the training dataset has already been input. It selects K neighbors as a group to classify. Value K is selected by experience. In our research, we use default value "1" for K.

**J48 (Quinlan, 1993)** uses the decision tree. It consists of a decision graph and possible results. It’s readable and easy to describe. What’s more, it’s efficient since the calculation times of each prediction are less than the depth of the decision.

**Logistic (le Cessie and van Houwelingen, 1992)** is a logistic regression model which can find the relationship between the dichotomy results and its factors.

**JRip (Cohen, 1995)** is a propositional rule learner which repeats two stages, a grow stage and a prune stage. In the grow stage, it will add the rule’s antecedents until the rule is perfect. In the prune stage, it will prune each rule to some degree.

After using the classifier, two methods are generally used to test the predicted class label. The first one is 10 fold cross-validation. The original data set is partitioned into 90% training set and 10% test set. This process repeats 10 times. The second
method is that we randomly select 30% original dataset as the test set, and the rest 70% dataset are used for training.

To evaluate the performance of the built model, TPR, TNR and overall accuracy (ACC) are used. The confusion matrix is used to help understand these measurements. It’s used to show the relationship between the actual class label and the predicted label using the built classification model in Table 2.2.

<table>
<thead>
<tr>
<th>Predicted class /Actual class</th>
<th>Positive</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>True Positives (TP)</td>
<td>False Negatives (FN)</td>
</tr>
<tr>
<td>Negative</td>
<td>False Positives (FP)</td>
<td>True Negatives (TN)</td>
</tr>
</tbody>
</table>

**Table 2.2:** Confusion matrix

- **TP**: the number of the benign applications that are correctly identified.
- **FP**: the number of the benign applications that are identified as malware.
- **TN**: the number of malware that are correctly identified.
- **FN**: the number of malware that are identified as the benign application.
- **TPR**: the proportion of the benign applications that are correctly identified. It’s also called sensitivity and recall.

\[
TPR = \frac{TP}{P} = \frac{TP}{TP + FN}
\]

- **TNR**: the proportion of malware that are correctly identified. It’s also called specificity.

\[
TNR = \frac{TN}{N} = \frac{TN}{TN + FP}
\]
• ACC: the proportion of the applications that are correctly identified:

\[
ACC = \frac{TP + TN}{TP + TN + FP + FN}
\]
Chapter 3

Security Metrics Selection

Based on a collection of security-related paper ((Chin et al., 2011),(Bugiel et al., 2011), (Dietz et al., 2011), (Felt et al., 2011b)... and our previous research, we choose four categories of metrics, the vulnerabilities, the dangerous behaviors, the sensitive features and the application attributes to analyze the security of an Android application. The malware developers can exploit the vulnerabilities exposed by the victim applications. The dangerous behaviors can be done by the dangerous applications. The sensitive features are used frequently in malware. The application attributes can show the application variation trend.

3.1 Vulnerabilities

In this section we survey the Android security vulnerabilities which are identified by many research groups. And we quantify these vulnerabilities using metrics. These vulnerabilities may be used by other malware developers.

3.1.1 Over-privileged Permissions

An application is Over-privileged when it requests more permissions than it uses. These extra permissions are potentially dangerous. For example, the malware may ask for enough permissions in the initial benign version, and when it updates to the new malicious version, it doesn’t need to ask for more permissions to do the bad behavior, which may relax the vigilance of users.
The Android operating system encourages the developers to follow the least-privileged principle, but it’s not compulsive since the insufficient API documentations. As a result, the developers themselves don’t pay much attention to it.

This metric records the count of extra permissions an application requested.

3.1.2 Open Components

Open components are public components which can be accessed by other components or even other applications. Open components can cause intent spoofing, eavesdropping, denial of service, hijacking attack (Chin et al., 2011).

Since the default value of ”exported” in a component is ”TRUE”, if a component is private, the developer should set the ”exported” as ”FALSE” explicitly. If the component needs to communicate with other components with the ”exported” as ”TRUE”, it’s better to set the intent target explicitly. Otherwise, Android operating system will decide which component should receive this intent.

This metric measures the number of open components in an application.

3.1.3 Re-delegation

Permission re-delegation is also called confused deputy attack. As in Android security mechanism, an application can get the dangerous permissions by asking for users to grant them. However, permission re-delegation circumvents this rule.

Re-delegation can be exploited through open components in the victim applications. It can also be used by an attacker directly. For example, they can develop a benign application and a malicious application which share the same UID. The malicious application can get all the permissions which the users granted to the benign application.

For example, in Figure 3.1, if Eve wants to call a particular method, such as, sending an SMS, the system will reject her since she doesn’t have permission. However, if
Eve sends an intent to Alice who has the sending SMS permission, then the system will approve Alice’s request since Alice has permission to call the method. In this way, Eve can send an SMS without the permission.

This metric records the number of protected methods that accessed by open components.

### 3.2 Dangerous Behaviors

Dangerous behaviors may lead to terrible consequence for the users. These dangerous behaviors are caused by the applications themselves.

#### 3.2.1 Hidden Files

An application may contain the dangerous behaviors in hidden files.

They can be detected by checking the postfix of files in the APK file structure. The dangerous postfix contains ”bat”, ”bll” and so on.

This metric measures the number of suspicious files in an application.
3.2.2 Dangerous Domains

The dangerous domains are identified malicious sites. The application containing dangerous domains can push ads to the users’ devices or get the important data from the device. Currently, there are more and more dangerous domains.

We collect 22,922 dangerous domains from the work of previous students and check whether the APK file contains these dangerous domains.

This metric records the number of malicious sites in an application.

3.2.3 Root Exploits

A root exploit is an application is attempt to gain the root privilege so that it can go around the Android security check. For example, the malicious application which has root exploit can install applications on the user’s device without users’ participation, even take control the whole device. This behavior is really extremely dangerous, many malware contain root exploits.

This metric records the count of root exploit times of an application.

3.2.4 Code Loading

There are four types of code loading, native code loading, reflection, dynamic code loading and crypto code loading. In malware datasets, the dynamic code loading is used the most frequently.

Native Codes Native codes refer to the developers use of native languages, such as C, C++ to write the Android application. It can increase the speed of the application. Though it is packaged into an APK file and run in the virtual machine on the device, it can also change the encryption scheme which is difficult for Android to check.

This metric is used to record the usage times the native codes in an application.
**Reflection**  Reflection is a method in Java, this method can be invoked dynamically. This metric records the invoked times of reflection method in an application.

**Dynamic Codes**  The dynamic codes can be run dynamically in an application. If the codes are loaded from a malicious server, the consequence is serious. This metric is defined as the number of times that dynamic codes are invoked in an application.

**Crypto Codes**  Crypto is a package related to application encryption and decryption in Android. This metric records the times of crypto codes invoked in an application.

### 3.3 Sensitive Features

There are many sensitive APIs in Android, we choose 40 APIs and divide them into four categories, cost-sensitive, personal information, sensitive input devices and device metadata. We use "1" to represent the API is used and "0" is not used.

#### 3.3.1 Cost-Sensitive

A cost-sensitive API is a function that may increase the users’ cost. For example, if an application abuses the SMS function, it may use the APIs, "sendTextMessage()", "sendMultipartTextMessage()", "sendMultimediaMessage()" under the class "android.telephony.SmsManager".

#### 3.3.2 Personal Information

It’s normal for an application to ask for users to share some information, even personal information. However the malware may use these functions to look up the user’s location, read the user’s phone and so on.
For example, to detect location leakage, the APIs, "GsmCellLocation/getLac()" and "GsmCellLocation/getCid()" are detected. To detect telephony leakage, the methods "getSubscriberId()", "getDeviceId()", "getLine1Number()", "getSimSerialNumber()", "getNetworkOperator()" and "getCellLocation()" under the class "TelephonyManager" may be used.

3.3.3 Sensitive Data Input Devices

Android allows applications to use the network, the zip files and so on. In order to check the network, we record the APIs, "getNetworkInfo()", "setWifiEnabled()", "getWifiState()" and so on. We also check the methods, "read()", "close()" and some other methods under the class "java.util.zip.ZipInputStream". These APIs are used frequently in malware.

3.3.4 Device MetaData

In the device settings part, we check the APIs, "getRunningServices()", "getMemoryInfo()" and "restartPackage()" in the class "android.app.ActivityManager". These APIs can get the running services or check the application’s memory usage or kill other processes. These methods are frequently used by malware. We also check some application related APIs, such as "android.content.ContentResolver" which provides the methods to access to content providers.

3.4 Application Attributes

This section we select the attributes of the application, including the size, classes, components, permissions. And we measure these attributes using metrics.
3.4.1 Application Size

Application size is a metric that records the size of the APK file. The unit in this research is kilobytes.

3.4.2 Application Classes

Application class count is a metric that counts the number of classes in an APK file.

3.4.3 Application Components

Application component count is a metric that counts the number of components used in an APK file.

3.4.4 Permissions

Permission contains two kinds of metrics.

Requested permission is a metric that counts the number of permissions requested which are declaimed in the manifest file of an application.

Used permission is a metric that counts the number of permissions used in an application, usually through API calls.
Chapter 4

Cross-Versions Analysis Tool

Cross-versions analysis tool is implemented to analyze the security of Android applications and compare the differences in multiple versions of an application. This tool is based on previous students' works (Nebiyu Feleke, 2014), we modify and extend the tool to fulfill the requirements of our research. The framework of the tool for our project is in Figure 4-1.

![Cross-versions analysis tool framework](image)

**Figure 4-1:** Cross-versions analysis tool framework
4.1 Collecting Data

To collect enough applications to analyze, we use two crawlers to download the applications from the Google Play market and from the third-party market called anzhi. We collect malware from the Genome Project (Zhou and Jiang, 2012).

The crawler module is used to download the applications from the market automatically. The first crawler is powered by (Alexandre, ) which is an open source library. It’s a Java based tool and used to download the applications from the official market. We use this crawler to download thousands of applications.

The second crawler is powered by (Sun, ). It’s an extensible crawler for downloading Android applications in third-party markets. It’s a Python based tool that it uses the downloaded url addresses of the applications to download the applications into the repository. We choose a Chinese third-party market, called anzhi, to download the applications. We choose anzhi as the third-party market since it provides multiple versions of each application, it helps us to analyze the application evolution. However, this crawler can only download the latest version of each application, so we modified the crawler to download multiple versions of each application. We download more than 5000 applications using this crawler, among them, nearly 900 applications have multiple versions.

We contacted the authors’ of (Zhou and Jiang, 2012) and were granted the permissions to access the malware database. We downloaded around 1500 malware from their database.

4.2 Extractor

The extractor module extracts the security parameters of each application’s APK file. It includes a collection of scripts.

The extractor module is based on the Androguard tool (Desnos, 2011). The
Androguard tool is a python powered tool that reverse engineers APK files from DEX/ODEX/APK/AXML/ARSC format into full Python objects. Its open source nature enables developers to customize the static analysis.

Before extracting the security parameters, the applications from the Google Play and from the third-party market are renamed to include both the package name and its version. For example, ”xxx.apk” will be called ”xxx-1.0.apk” after renaming, ”xxx” is its package name and ”1.0” is its version.

Based on Androguard, we developed a collection of scripts explained below:

1. **Over-privileged Script**: checks the requested permissions which are declaimed in manifest files and gets the used permissions in the Dalvik code. It lists the permissions which are in the requested permissions list but not in the used permissions list, then counts them. This script stores the application name and the number of permissions that are unused in an application in the .csv file.

2. **Open Component Script**: reads the component information from the manifest files. If the component doesn’t set the ”exported” value as ”FALSE” explicitly, the open component count adds ”1”. This script stores the application name and the count of open components in the .csv file.

3. **Re-delegation Script**: gets the implicit open components from the manifest files and checks in the Dalvik code if any component is making a call to the protected methods. (Nebiyu Feleke, 2014) If it does, the number of re-delegation of this application increases ”1”. This script stores the application name and the count of re-delegations of each application in the .csv file.

4. **Hidden File Script**: checks the files from the manifest files, if the file has the suffix, such as ”bat”, ”chm” and so on, the count of hidden files adds ”1”.
This script stores the application name and the count of hidden files of each application in the .csv file.

5. **Malicious Domain Script:** checks the strings from the Dalvik code. We get a malicious domains list from previous work, if any string in the malicious domains list, the malicious domain count increases "1". This script stores the application name and the number of malicious domains in an application in the .csv file.

6. **Root Exploit Script:** checks the strings in the Dalvik code, if a string contains "test-keys", "release-keys" or "su", or it starts with "su", "/system/" or "busy-box", this string is considered as containing the root exploit. This script stores the application name and the number of root exploit strings in an application in the .csv file.

7. **Code Loading Script:** searches the methods to check the native code loading, dynamic code loading, reflection, crypto code loading in Dalvik code. To detect the dynamic code loading, it searches the class "dalvik/system/DexClassLoader;", and returns this class usage time. To detect the reflection, the script searches the class "java/lang/reflect/Method;" and also returns the usage times. To detect the crypto code loading, the script searches the class "javax/crypto/.", and record the usage times. This script stores the application name and the counts of native code loading, dynamic code loading, reflection and crypto code loading in the .csv file.

8. **Sensitive API Script:** checks the Dalvik code to see whether the application uses the 40 sensitive APIs defined previously. If the application uses one of the sensitive APIs, the corresponding sensitive API's value sets "1", if not, the value sets "0". This script stores the application name and the sensitive APIs
usage in the .csv file.

9. **Size Script**: returns the application name and the size of each APK file and stores in the .csv file.

10. **Component Script**: reads the component defined in the manifest files. This script returns the application name and the number of components of an application in .csv file.

11. **Class Script**: searches the class information in the Dalvik code and returns the application name and the count of class in an application in .csv file.

12. **Permission Script**: checks the requested permissions which declared in manifest files and gets the used permissions in the Dalvik code. This script returns the application name, the number of the requested permissions, the used permissions, the list of the requested permissions and the list of the used permissions and stores as the .csv file.

13. **Requested Permission Script**: checks the manifest file to see whether the application requests the top 40 permissions. These permissions are extracted from the permissions-based malware detection module. If the application uses one of the top permissions, the corresponding permission’s value set to be ”1”, else, the value set to be ”0”. This script shows the application name and the top requested permissions usage and stores in the .csv file.

4.3 Analysis

The analysis module is used to analyze the application evolution and finish the preparatory work for malware detection and update attack detection.
**Application Evolution Module**  To analyze the application evolution, we choose the applications which have multiple versions. This module collects a number of metrics, the vulnerabilities, the dangerous behaviors and the application attributes. After that, the latest version subtracts the corresponding value of the previous version. This module gets the result and records the change of each application which has multiple versions and stores the results in the .csv file.

**Permissions-based Malware Detection Module**  This module consists of the following steps:

1. We use the permission script to get all the requested permissions in the malware dataset. These requested permissions are used as a vector list.

2. For an application, if it requests the permissions listed in the vector list, the corresponding value is set to ”1”, else to ”0”. For example, if the application uses the first and the third permissions, it can be indicated as ”1,0,1,0,0,0,0,...”

3. If the application is from Google Play, we assume that it is the legitimate application, and we use ”1” to represent a benign application. This value will append the previous number list. If the application is from the malware dataset, the appended number will be ”2”. If the application which we take as an example previously is from Google Play, now it can be indicated as ”1,0,1,0,0,0,...,1”; if it’s from malware dataset, it can be shown as ”1,0,1,0,0,0,...,2”.

4. The .csv file format converts to the .arff file that can be read in Weka. The last column is the class label.

The same steps are followed for the used permissions.

In the second step, we can also get the percentage difference of each permission in malware and benign applications. Then we rank the percentage difference to get a
ranked permission list. The top 40 permissions which are used as feature vectors in
the following modules are selected here.

**Multiple Metrics Malware Detection Module**  This module collects the data
of the vulnerabilities, the dangerous behaviors, the sensitive APIs and the top 40
permissions of each application. If the application is from Google Play or the third-
party market, they are considered as legitimate applications, and the class label will
be "1". If the application is from malware dataset, the class label will be "2".

**Update Attack Detection Module**  This module collects the metrics of vulnera-
bilities, the dangerous behaviors, the sensitive APIs, the top 40 requested permissions
and the application attributes. However, the metrics used in this module are the dif-
ference value between the new version and the old version of an application. The
class labels are the same as those defined in the multiple metrics malware detection
module.
Chapter 5

Application Evolution

To analyze the application evolution, the application evolution module analyzes 292 applications with two versions from Google Play and 886 applications with two versions from the third-party market. Each application was checked for the vulnerabilities, the dangerous behaviors and the application metadata.

To show the evolution more clearly, we show each metric in the initial version in Figure 5.1 and the change in the new version in Figure 5.2. These two figures are using the mean value of each metrics to do the comparison.

From Figure 5.1, we can see that the applications from the Google Play or the third-party market have the vulnerabilities and the dangerous behaviors. Some metrics, such as hidden files or root exploits, illustrate the applications from Google Play are less dangerous than those from the third-party market. However, we cannot easily conclude that the applications from the third-party are worse than those from Google Play. For example, the applications from Google Play are more frequently using the dynamic codes, a very potentially dangerous behavior, than the applications from the third-party market.

From Figure 5.2, we can see not all metrics are less in the updated version. Re-delegation and native code-loading are less not only in the official market, but also in the third-party market. The values of the rest metrics are increased or decreased slightly. However, this figure can only give an initial impression since the application evolution can be more precisely described when compared with the old version.
Figure 5-1: Application evolution original metrics
Figure 5.2: Variation in application evolution
In the following sections, we analyze each metric in details. From Table 5.1, Table 5.2 and Table 5.3, we list the change across the versions. The ”decreased” column means that the percentage of applications delete or decrease the corresponding metric. The ”increased” column shows that the percentage of applications have more count of that metric. The ”same” means that both versions have this metric and this metric stayed the same in the updated version. The ”zero” shows the percentage of both version without such metrics.

### 5.1 Vulnerabilities Analysis

From Table 5.1, we can see that these vulnerabilities frequently happen in most applications, regardless if they are from the official market or the third-party market.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Market</th>
<th>Decreased</th>
<th>Increased</th>
<th>Same</th>
<th>Zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>Over-privileged</td>
<td>Google Play</td>
<td>5%</td>
<td>11%</td>
<td>73%</td>
<td>11%</td>
</tr>
<tr>
<td></td>
<td>Third-party Market</td>
<td>17%</td>
<td>38%</td>
<td>45%</td>
<td>0%</td>
</tr>
<tr>
<td>Open components</td>
<td>Google Play</td>
<td>4%</td>
<td>13%</td>
<td>83%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>Third-party Market</td>
<td>13%</td>
<td>37%</td>
<td>44%</td>
<td>6%</td>
</tr>
<tr>
<td>Re-delegation</td>
<td>Google Play</td>
<td>51%</td>
<td>9%</td>
<td>14%</td>
<td>26%</td>
</tr>
<tr>
<td></td>
<td>Third-party Market</td>
<td>30%</td>
<td>12%</td>
<td>38%</td>
<td>20%</td>
</tr>
</tbody>
</table>

**Table 5.1: Vulnerabilities evolution analysis**

Over-privileged permissions metric is also not optimistic. 11% of applications from the Google Play and 38% of applications from the third-party market have more extra permissions in the new version. While 73% of applications from the Google Play and 45% of applications from the third-party market remain the same in the new versions of applications. The developers may have benign purpose, such as more convenience, they ask more permissions than they use. However, it’s not suitable
since the malicious developers may use this method to harm the users. What’s more, Android also encourages the developers to follow the least-privileged permissions rule.

Open components are less optimistic than re-delegation. 83% of applications from the Google Play and 44% of applications from the third-party suffers from this vulnerability and have no change in the updated version. Open components may be not brought to the attention, since the default value of the component is ”TRUE”, most developers don’t get used to modify it to ”FALSE” manually.

More than half applications from the official market decrease the value of re-delegation in the updated version, it can be inferred that re-delegation has already been detected and the developers have tried to fix this vulnerability in the updated version. However, 30% of the applications from the third-party market have less re-delegation and 38% of which suffer from the re-delegation don’t change in the new version. The improvement in the third-party is less obvious than that in the Google Play. What’s more, we cannot ignore the problem of re-delegation, since only 26% of applications from the Google Play and 20% of applications from the third-party market are without re-delegation in the whole update process.

5.2 Dangerous Behaviors Analysis

From Table 5.2, most applications don’t have dangerous behaviors, since the dangerous behaviors are different from the vulnerabilities. The dangerous behaviors are frequently contained in the malware, it can be more severe than the vulnerabilities. We can also see that the deleted rate and the added rate in the third-party market are higher than those in the official market, which means that the third-party market is less stable than the official market.

Hidden files are used more frequently in the applications from the third-party market when compared to Google Play. Only few applications have less hidden files
<table>
<thead>
<tr>
<th>Dangerous behaviors</th>
<th>Market</th>
<th>Decreased</th>
<th>Increased</th>
<th>Same</th>
<th>Zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hidden files</td>
<td>Google Play</td>
<td>1%</td>
<td>2%</td>
<td>13%</td>
<td>84%</td>
</tr>
<tr>
<td></td>
<td>Third-party Market</td>
<td>4%</td>
<td>7%</td>
<td>19%</td>
<td>70%</td>
</tr>
<tr>
<td>Dangerous domains</td>
<td>Google Play</td>
<td>2%</td>
<td>1%</td>
<td>7%</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>Third-party Market</td>
<td>1%</td>
<td>0%</td>
<td>2%</td>
<td>97%</td>
</tr>
<tr>
<td>Root exploits</td>
<td>Google Play</td>
<td>3%</td>
<td>4%</td>
<td>27%</td>
<td>66%</td>
</tr>
<tr>
<td></td>
<td>Third-party Market</td>
<td>8%</td>
<td>18%</td>
<td>28%</td>
<td>46%</td>
</tr>
<tr>
<td>Native codes</td>
<td>Google Play</td>
<td>5%</td>
<td>7%</td>
<td>28%</td>
<td>61%</td>
</tr>
<tr>
<td></td>
<td>Third-party Market</td>
<td>16%</td>
<td>32%</td>
<td>34%</td>
<td>18%</td>
</tr>
<tr>
<td>Reflection</td>
<td>Google Play</td>
<td>12%</td>
<td>30%</td>
<td>53%</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Third-party Market</td>
<td>19%</td>
<td>41%</td>
<td>33%</td>
<td>7%</td>
</tr>
<tr>
<td>Dynamic codes</td>
<td>Google Play</td>
<td>2%</td>
<td>13%</td>
<td>35%</td>
<td>50%</td>
</tr>
<tr>
<td></td>
<td>Third-party Market</td>
<td>7%</td>
<td>12%</td>
<td>22%</td>
<td>59%</td>
</tr>
<tr>
<td>Crypto codes</td>
<td>Google Play</td>
<td>3%</td>
<td>13%</td>
<td>64%</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>Third-party Market</td>
<td>14%</td>
<td>33%</td>
<td>37%</td>
<td>16%</td>
</tr>
</tbody>
</table>

**Table 5.2:** Dangerous behaviors evolution analysis

in their updated versions. However, 7% of applications from the third-party increase the value of this dangerous behavior in the updated version, which is really worse.

Dangerous domains are rare in the both markets. Most applications don’t change the dangerous domains in the updated version. The reason is that, the dangerous domains are the identified malicious sites. The official market or the third-party market may delete the applications which contained these identified malicious domains from the markets once they detect them.

Root exploits are a dangerous behavior which is applied in most malware. The applications from the third-party market increase or decrease the root exploit counts
more frequently than the applications from Google Play. For example, 18% of applications from the third-party market have more root exploits while only 4% of applications from Google Play increase the count of root exploit.

Code loading is more serious in the third-party market than in the Google Play. Though the percentage of decreasing the count of code loading in the third-party market is higher than that in the Google Play, the percentage of increasing is also higher in the third-party market.

### 5.3 Application Attributes Analysis

<table>
<thead>
<tr>
<th>Metadata</th>
<th>Market</th>
<th>Decreased</th>
<th>Increased</th>
<th>Same</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requested perms</td>
<td>Google Play</td>
<td>5%</td>
<td>13%</td>
<td>81%</td>
</tr>
<tr>
<td></td>
<td>Third-party</td>
<td>13%</td>
<td>45%</td>
<td>42%</td>
</tr>
<tr>
<td>Used perms</td>
<td>Google Play</td>
<td>9%</td>
<td>21%</td>
<td>70%</td>
</tr>
<tr>
<td></td>
<td>Third-party</td>
<td>14%</td>
<td>38%</td>
<td>48%</td>
</tr>
<tr>
<td>Classes</td>
<td>Google Play</td>
<td>16%</td>
<td>56%</td>
<td>28%</td>
</tr>
<tr>
<td></td>
<td>Third-party</td>
<td>19%</td>
<td>64%</td>
<td>17%</td>
</tr>
<tr>
<td>Components</td>
<td>Google Play</td>
<td>9%</td>
<td>26%</td>
<td>65%</td>
</tr>
<tr>
<td></td>
<td>Third-party</td>
<td>17%</td>
<td>54%</td>
<td>29%</td>
</tr>
<tr>
<td>Size</td>
<td>Google Play</td>
<td>19%</td>
<td>56%</td>
<td>25%</td>
</tr>
<tr>
<td></td>
<td>Third-party</td>
<td>25%</td>
<td>65%</td>
<td>10%</td>
</tr>
</tbody>
</table>

**Table 5.3:** Application metadata evolution analysis

From Table 5.3, we can also see that the percentage of the metrics changed are higher in the third-party market.

In terms of permissions, the requested permissions and the used permissions have more in the new versions of applications from the third-party market. Most applica-
tions from the official market don’t change the permissions in the updated version.

The classes count, the component count and the size of the most applications tend to have more in the updated version. This may be the result of the functionality development of applications. And these three metrics are related to each other.
Chapter 6

Permissions-based Malware Detection

Android’s permission model has its advantages and shortcomings. Most malware need to request and use some permissions to achieve their malicious goals. As a result, permissions are useful features in malware detection.

We propose to use machine learning classifiers to detect the malware based on the application permissions. We compare the performance of the requested permissions and the used permissions by using four classifiers. The result is that the best performances of using the requested permissions and using the used permissions are similar.

6.1 Approach Overview

Our goal is to build a model which can distinguish the malware from the benign applications efficiently based on Android permissions. We find that the requested permissions are not always the same as the used permissions. We cannot reach a conclusion arbitrarily as to whether the best results are achieved from looking at the requested permissions or the used permissions. As a result, we compare the performance of using the requested permissions with the used permissions.

For each requested permissions collected from malware dataset, we calculate the percentage difference between malware and the legitimate application dataset, and order them based on this value. The percentage is calculated as the number of applications requested this permission over the total number of applications in the
6.2 Dataset

Our data set consists of malware dataset and benign application dataset. The malware dataset has 1247 malware samples from Genome Project (Zhou and Jiang, 2012). And the benign applications consist of 1514 applications downloaded from five categories in Google Play. 528 apps are from the business category, 272 apps are from the finance category, 265 apps are from the health and fitness category, 33 apps come from the shopping category and 416 apps are from the application widgets category.

6.3 Feature Vectors

The requested permissions are extracted from each application’s manifest file. The used permissions are extracted from .dex file.

102 different permissions are requested by the malware in our malware dataset. Instead of choosing the top permissions requested in malware, we rank the permissions by the percentage difference between malware and benign applications, because many permissions are requested frequently not only in the malware but also in the benign apps. For example, 97.67% malware request INTERNET permission while 94.98% benign apps also request this permission. Our aim is to use these permissions to classify the malware and benign ones and we want to find the difference between them, ranking difference value is better than only ranking the top frequently requested permissions in the malware. Table 6.1 shows the top 20 requested permissions with the highest percentage difference in malware and benign samples.

There are only 33 permissions used in malware. Table 6.2 depicts the top 20 difference value of used permissions in malware and benign samples.
<table>
<thead>
<tr>
<th>Permission</th>
<th>Malware rate</th>
<th>Goodware rate</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_SMS</td>
<td>63.35%</td>
<td>5.88%</td>
<td>57.47%</td>
</tr>
<tr>
<td>WRITE_SMS</td>
<td>52.77%</td>
<td>3.90%</td>
<td>48.87%</td>
</tr>
<tr>
<td>READ_PHONE_STATE</td>
<td>93.50%</td>
<td>47.03%</td>
<td>46.47%</td>
</tr>
<tr>
<td>SEND_SMS</td>
<td>42.82%</td>
<td>6.74%</td>
<td>36.08%</td>
</tr>
<tr>
<td>RECEIVE_SMS</td>
<td>38.97%</td>
<td>6.61%</td>
<td>32.36%</td>
</tr>
<tr>
<td>GET_ACCOUNTS</td>
<td>4.09%</td>
<td>35.67%</td>
<td>31.58%</td>
</tr>
<tr>
<td>CAMERA</td>
<td>1.52%</td>
<td>31.24%</td>
<td>29.72%</td>
</tr>
<tr>
<td>RECEIVE_BOOT_COMPLETED</td>
<td>55.17%</td>
<td>27.15%</td>
<td>28.02%</td>
</tr>
<tr>
<td>ACCESS_WIFI_STATE</td>
<td>64.47%</td>
<td>36.46%</td>
<td>28.01%</td>
</tr>
<tr>
<td>WRITE_APN_SETTINGS</td>
<td>27.99%</td>
<td>1.06%</td>
<td>26.93%</td>
</tr>
<tr>
<td>RESTART_PACKAGES</td>
<td>26.70%</td>
<td>3.96%</td>
<td>22.74%</td>
</tr>
<tr>
<td>CHANGE_WIFI_STATE</td>
<td>31.92%</td>
<td>9.71%</td>
<td>22.21%</td>
</tr>
<tr>
<td>WRITE_CONTACTS</td>
<td>29.99%</td>
<td>9.05%</td>
<td>20.94%</td>
</tr>
<tr>
<td>INSTALL_PACKAGES</td>
<td>20.05%</td>
<td>0.73%</td>
<td>19.32%</td>
</tr>
<tr>
<td>CALL_PHONE</td>
<td>34.00%</td>
<td>14.99%</td>
<td>19.01%</td>
</tr>
<tr>
<td>BILLING</td>
<td>0.08%</td>
<td>17.31%</td>
<td>17.23%</td>
</tr>
<tr>
<td>READ_CONTACTS</td>
<td>36.65%</td>
<td>21.00%</td>
<td>15.65%</td>
</tr>
<tr>
<td>WAKE_LOCK</td>
<td>34.00%</td>
<td>49.08%</td>
<td>15.08%</td>
</tr>
<tr>
<td>DISABLE_KEYGUARD</td>
<td>19.81%</td>
<td>4.95%</td>
<td>14.86%</td>
</tr>
<tr>
<td>READ_LOGS</td>
<td>19.25%</td>
<td>7.73%</td>
<td>11.52%</td>
</tr>
</tbody>
</table>

**Table 6.1:** 20 requested permissions with the highest percentage difference in malware and benign applications
<table>
<thead>
<tr>
<th>Permission</th>
<th>Malware rate</th>
<th>Goodware rate</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_CONTACTS</td>
<td>18.85%</td>
<td>71.00%</td>
<td>52.15%</td>
</tr>
<tr>
<td>FACTORY_TEST</td>
<td>25.18%</td>
<td>69.88%</td>
<td>44.70%</td>
</tr>
<tr>
<td>WAKE_LOCK</td>
<td>31.11%</td>
<td>74.17%</td>
<td>43.06%</td>
</tr>
<tr>
<td>USE_CREDENTIALS</td>
<td>0.08%</td>
<td>35.14%</td>
<td>35.06%</td>
</tr>
<tr>
<td>READ_PHONE_STATE</td>
<td>94.87%</td>
<td>62.02%</td>
<td>32.85%</td>
</tr>
<tr>
<td>SEND_SMS</td>
<td>34.16%</td>
<td>5.35%</td>
<td>28.81%</td>
</tr>
<tr>
<td>CAMERA</td>
<td>1.36%</td>
<td>29.66%</td>
<td>28.30%</td>
</tr>
<tr>
<td>GET_ACCOUNTS</td>
<td>4.25%</td>
<td>29.85%</td>
<td>25.60%</td>
</tr>
<tr>
<td>VIBRATE</td>
<td>71.21%</td>
<td>92.60%</td>
<td>21.39%</td>
</tr>
<tr>
<td>ACCESS_COARSE_LOCATION</td>
<td>29.27%</td>
<td>8.45%</td>
<td>20.82%</td>
</tr>
<tr>
<td>ACCESS_FINE_LOCATION</td>
<td>44.35%</td>
<td>64.80%</td>
<td>20.45%</td>
</tr>
<tr>
<td>RECORD_AUDIO</td>
<td>2.65%</td>
<td>20.21%</td>
<td>17.56%</td>
</tr>
<tr>
<td>BLUETOOTH</td>
<td>0.56%</td>
<td>17.64%</td>
<td>17.08%</td>
</tr>
<tr>
<td>READ_LOGS</td>
<td>50.84%</td>
<td>34.54%</td>
<td>16.30%</td>
</tr>
<tr>
<td>ACCESS_NETWORK_STATE</td>
<td>81.64%</td>
<td>93.33%</td>
<td>11.69%</td>
</tr>
<tr>
<td>CHANGE_COMPONENT_ENABLED_STATE</td>
<td>7.22%</td>
<td>11.29%</td>
<td>10.57%</td>
</tr>
<tr>
<td>CHANGE_WIFI_STATE</td>
<td>0.08%</td>
<td>8.98%</td>
<td>8.90%</td>
</tr>
<tr>
<td>RESTART_PACKAGES</td>
<td>9.94%</td>
<td>3.37%</td>
<td>6.57%</td>
</tr>
<tr>
<td>MODIFY_AUDIO_SETTINGS</td>
<td>0.48%</td>
<td>6.87%</td>
<td>6.39%</td>
</tr>
<tr>
<td>WRITE_SETTINGS</td>
<td>2.65%</td>
<td>8.78%</td>
<td>6.13%</td>
</tr>
</tbody>
</table>

**Table 6.2:** 20 used permissions with the highest percentage difference in malware and benign applications
6.4 Classifiers Performance

We compare the classifiers performances using the requested permissions with the used permissions in this section.

We use four classifiers to compare the performance. They are IBK, J48, Logistic and JRip. We use 10 times cross-validation method to do the test. The performance is evaluated by the accuracy, TPR and TNR.

We found 102 requested permissions in the malware dataset. Each time we add 10 more permissions as the feature vectors to build the model. The requested permissions performance with the highest percentage difference is illustrated in Figure 6·1, Figure 6·2 and Figure 6·3.

![Accuracy of using the requested permissions](image)

**Figure 6·1:** Accuracy of using the requested permissions

Figure 6·1 shows that classifier IBK always has higher accuracy than other classifiers. The highest accuracy is achieved at 96.5% when almost 40 or more than 80 permissions are used.
Figure 6.2: TPR of using the requested permissions

Figure 6.3: TNR of using the requested permissions
Figure 6·2 shows that the classifiers have a stable TPR when top 30 permissions are selected and the TPR is more than 95% after that. The IBK has the highest TPR which is 97.6% when top 30 permissions are used as feature vectors. Figure 6·3 shows that the IBK also has the highest TNR, which is 95.5% when the top 40 permissions are selected.

In this case, it’s obvious that the classifier IBK has a better performance than other classifiers. And since more feature vectors are selected, the time of building model will take longer time, there’s a tradeoff between accuracy, TPR and TNR. From Figure 6·1, Figure 6·2 and Figure 6·3, we could conclude that top 40 permissions are the best option. Although top 80, 90 and 100 permissions have the similar performance with top 40 permissions, using 40 features vectors are less time-consuming.

We found only 34 used permissions in the malware dataset. Each time we add 5 more permissions as the feature vectors to build the classification model. The used permissions performance with the highest percentage difference is shown in Figure 6·4, Figure 6·5 and Figure 6·6.

Figure 6·4 illustrates that the classifier IBK has the best performance in terms of accuracy. When the top 25 permissions are used as feature vectors, the accuracy of IBK is 96.5%, which is the highest one. When more feature vectors are selected, the accuracy, however, decreases.

Figure 6·5 shows that the classifier Logistic has the best sensitivity at first, however, classifier IBK gets the highest sensitivity (97.2%) at top 25 permissions. In Figure 6·6, classifier IBK gets the highest specificity (95.7%) when the top 25 permissions are used.

There’s no doubt that classifier IBK with the top 25 permissions are the first choice when using the used permissions to build the classification model.
Figure 6-4: Accuracy of using the used permissions

Figure 6-5: TPR of using the used permissions
6.5 Comparison

In this section, we compare the requested permissions with the used permissions when they have the best performance. The same point is that they are both using classifier IBK, the difference is that the requested permissions use the top 40 while the used permissions use the top 25.

Figure 6-6 illustrates the best performance of using the requested permissions and the used permissions. From the figure, we can see that they have the same TPR. The used permissions have higher TNR, but lower accuracy. Although higher TNR means that this model can classify malware more precisely, we argue that the requested permission is the better choice for two reasons. First, the TNR of these two models is approximate, the difference is only 0.2%. Second, the requested permissions are extracted from manifest files, which is much faster than extracting the used permissions from the .dex files. Furthermore, if an application only uses the documented API,
about 18%-26% of the non-system permissions can be hidden (Au et al., 2012), but the used permissions are extracted from the API calls in Androguard (Desnos, 2011), this is also a reason why only 34 kinds of permissions are found in malware. As a result, if we want to use the used permissions, the APIs would be a better option.

**Figure 6.7:** Performance comparison between the requested permissions and the used permissions
Chapter 7

Multiple Metrics Malware Detection

Chapter 6 shows the performance of only using permissions, the results still have space to improve, as a result, in this chapter, we use multiple metrics to detect malware. The metrics we defined in Chapter 3 are used as the feature vectors. After this, we build the models to classify the malware and legitimate applications by using four different classification algorithms. For each model, we compare the performance among the Google Play applications with malware, the third-party applications with malware and their combination with malware.

7.1 Approach Overview

Our goal is to build a model which has a better performance to classify the malicious applications from the benign applications. As a result, we come up with the idea that using multiple security metrics to do the classification instead of only using permissions.

Each application is represented by a list of numbers. Each number is the count of the corresponding security metric or whether that metric is used in the application. Then each application is marked by a class label, the legitimate applications are labeled as ”1” and the malware are labeled as ”2”. Then we use these metrics as feature vectors to do the classification. Four classifiers (J48, Logistic Regression, IBK and JRip) are chosen to build the models. At last, we evaluate the performance of each classifier by accuracy, TPR and TNR.
7.2 Dataset

The dataset consists of malware samples, applications from the Google Play and the applications from the third-party market.

In order to have a better comparison with the permissions-based malware detection, we use the same applications (1514) from the Google Play and the same malware samples (1247). We import the third-party market in multiple metrics malware detection part, and 3822 applications are selected.

We assume that the applications from the Google Play and the third-party market are legitimate applications, their class labels are ”1”. The malware samples are all malicious, their class labels are ”2”.

7.3 Feature Vectors

The vulnerabilities, the dangerous behaviors, the sensitive API and the top requested permissions are selected as the feature vectors. We exclude the application metadata in the multiple metrics malware detection, since the size of malware is much smaller than that of normal applications, as a result, the component count and the class count in malware are also much less than those in normal applications. It would skew the results of the classification and would lead to bias.

In the selected metrics, there are 3 feature vectors belonging to the vulnerabilities, they are re-delegation, open components and over-privileged permissions. There are 7 feature vectors belonging to the dangerous behaviors, they are hidden files, root exploits, dangerous domains, native code loading, dynamic code loading, reflection and crypto code loading. There are 40 sensitive APIs related to cost, personal information, data input devices or device metadata. And there are 40 requested permissions are selected. These 40 permissions are chosen in the permissions-based malware detection part. The best performance of using requested permissions is when the top
requested 40 permissions are chosen. They are ranked by the percentage difference between malware and legitimate applications.

### 7.4 Classifiers Performance

Four classifiers are used to build the models. They are J48, Logistic Regression, IBK and JRip, respectively. For each model, the performance is evaluated by accuracy, TPR and TNR.

We have three experiments, the first one we compare the Google Play applications with the malware. The second one we compare the third-party applications with the malware. The third one we compare the applications from Google Play and the third-party market with the malware, which are shown in Figure 7·1, Figure 7·2 and Figure 7·3.

![Figure 7·1: Accuracy of multiple metrics malware detection](image)

In Figure 7·1, IBK has the best accuracy no matter in which markets. And in every market, the accuracy is almost the same by using the IBK classifier, which
Figure 7.2: TPR of multiple metrics malware detection

Figure 7.3: TNR of multiple metrics malware detection
is 99.3% (in the both markets, the accuracy is 99.4%). What’s more, the lowest accuracy is 97.9% in the third-party market by using JRip, however, this accuracy is still higher than the best accuracy of only using permissions as the feature vectors.

In Figure 7·2, IBK classifier has no doubt to have the best TPR, the value is 99.6%. In Figure 7·3, IBK classifier also has the best TNR, the value is about 99%, which means this model can distinguish the malware from unknown application dataset exactly.

### 7.5 Comparison

In this section, we compare the multiple metrics with the requested permissions. In Figure 7·4, the applications are from the Google Play and malware dataset, while in the requested permissions, the result is the best performance that uses the top 40 permissions. We can easily conclude that using multiple metrics can achieve a much better performance than using the permissions only, no matter in terms of accuracy, TPR or TNR. Especially, the TNR increases most, which means that using multiple metrics can classify the malware more precisely.

There are three reasons which make using multiple metrics having a better performance, first, while we use multiple metrics, we contain the top 40 requested permissions, in other words, we select more feature vectors to build the classification models. Second, from the used permissions malware detection, we know that less used permissions can have the similar performance with that using requested permissions. However, given the lack of API documentation, we use the API instead of using the used permissions. Third, the dangerous behaviors help to classify the malware, since malware has more dangerous behaviors than the normal applications.
**Figure 7.4:** Performance comparison between using multiple metrics and the requested permissions
Chapter 8

Update Attack Detection

When we investigate the Android application evolution, security may not be enhanced in the new version of an application. This phenomenon leads us to pay attention on update attack. Update attack is a new type of attack where the malicious code is contained in the update components of a legitimate application. It is easy to carry out because users don’t tend to question the legitimacy of updates for already-installed software.

In this chapter, we implement two test experiments to see the update attack detection performance using machine learning techniques.

8.1 Approach Overview

To detect update attack, we conduct the experiments from two directions.

First, we use the old versions of the applications and 70% malware as the training dataset. We use the new versions of the applications and 30% malware as the test dataset. The feature vectors are the same as those in multiple metrics malware detection.

For second experiment, we use the difference values of each metric as the feature vectors. This experiment tends to detect update attacks, since if the malware is contained in the new version of an application, the difference values of metrics will be similar to the original values of metrics in malware.

Both experiments have four classifiers to do the classification. Accuracy, TPR
and TNR are used to evaluate the performance of each classifier.

8.2 Dataset

We collect 292 applications with two versions of the Google Play and 886 applications with two versions from the third-party market. In the malware dataset, we have 1247 malware samples.

In the first experiment, The applications of the old version and 70% malware consist of the training dataset. The applications of the new version and the rest malware are used as the test dataset.

In the second experiment, 70% applications from the Google Play or the third-party market and 70% malware are used as the training dataset. The test dataset is consisted of the rest applications and the rest malware.

8.3 Feature Vectors

In the first experiment, the feature vectors are totally the same as those in the multiple metrics malware detection. The feature vectors include the vulnerabilities, the dangerous behaviors, the sensitive APIs and 40 requested permissions with the highest percentage difference in malware and benign applications.

However, in the second experiment, the feature vectors include the vulnerabilities, the dangerous behaviors, the sensitive APIs, the requested permissions in malware and the application metadata. We use the difference value between the two versions of an application to represent each metric in the vulnerabilities and the dangerous behaviors categories. Since we use "0" and "1" to represent the sensitive APIs and permissions in previous experiments, in order to show the change, we use "1" to represent if the metric in the old versions is "0" and the metric in the new version is "1", otherwise, the metric is "0". In this experiment, we add the application metadata.
as feature vectors, since the different values between versions is comparable to the metadata in malware. For example, if malware is inserted into the new version of an application, the difference value of size is almost the same of that malware size.

8.4 Classifiers Performance

We use four classifiers to compare the performance, J48, Logistic Regression, IBK and JRip. We also compare the performance of different sources, the Google Play, the third-party market and their combination. For each model, accuracy, TPR and TNR are used to evaluate the classification performance.

Figure 8.1, Figure 8.2 and Figure 8.3 show the first update attack detection experiment result.

![Figure 8.1: Accuracy of update attack detection (the first experiment)](image)

In Figure 8.1, the IBK classifier has the best accuracy, which are 99.1% and 99.6% in the Google Play market and the third-party market.

In Figure 8.2, the IBK classifier also has an average best TPR. However, we can
Figure 8.2: TPR of update attack detection (the first experiment)

Figure 8.3: TNR of update attack detection (the first experiment)
find the performance in the third-party market is better than in the Google Play. The sample number may cause this because the application number of the third-party is 4 times larger than the application number from the Google Play. Figure 8-3, the TNR reaches "1" when the classifier is the IBK and the market is Google Play. This implies that this model can classify all the malware from the unknown applications dataset.

Figure 8-4, Figure 8-5 and Figure 8-6 show the second update attack detection experiment result.

**Figure 8-4:** Accuracy of update attack detection (the second experiment)

Figure 8-4 illustrates the applications from Google Play always have a better accuracy than the applications from third-party market (99.3% and 98.8%, respectively). The IBK classifier always has the best accuracy than other classifiers.

In Figure 8-5, the applications from the third-party market have a better TPR than the Google Play when the classifier is the IBK. The reason may be the same as in the first experiment. In Figure 8-6, the highest TNR is 1 when the classifier is
Figure 8.5: TPR of update attack detection (the second experiment)

Figure 8.6: TNR of update attack detection (the second experiment)
the JRip and the market is Google Play, however, the average best TNR is using the IBK classifier, which is 99.7% and 98.9% while the markets are Google Play and the third-party market.

8.5 Comparison

In this section, we compare performance of the two experiments to detect update attack with using multiple metrics to detect malware. In Figure 8·7, the performances are using the Google Play applications and malware with the classifier IBK. The accuracy and TPR of the two experiments are similar which using multiple metrics is better. However, the TNR of the two experiments are better than using multiple metrics, particularly, the TNR of the first experiment reaches to ”1”.

![Figure 8·7: Performance comparison between update attack detection and using multiple metrics](image)

Though the accuracy and TPR of using the multiple metrics are better than the update attack detection, we argue that the update attack detection improves the performance of using the multiple metrics for the following reasons. It’s the imbalanced
benign samples. More than 1000 Google Play applications train the model of using multiple metrics while around 200 Google Play applications train the model of two experiments. If a benign application is tested as the malware, the accuracy and TPR will decrease. As a result, if the samples are small, it’s difficult to achieve a higher TPR, which will effect the accuracy. Moreover, we assume that the applications from the official market and the third-party market are benign. However, we don’t know these applications are really benign or not. So in this model, it’s hard to conclude it’s true that the benign application is misclassified as the malware. As a result, the TPR and accuracy are not fair to evaluate the performance. Nevertheless, the TNR can evaluate the performance more reasonable since the malware are identified by the research groups and the malware samples of these three are the same. It proves that the update attack detection can have a better performance to classify the malware than using the multiple metrics to detect malware.
Chapter 9

Related work

9.1 Android Security

The Android security issue is always on the spotlight, many research groups try to find and solve the security problems. Comdroid (Chin et al., 2011) talks about if the intent doesn’t identify the recipient or the sender, it may be intercepted or suffer from spoofing attacks. XmanDroid (Bugiel et al., 2011), Quire (Dietz et al., 2011) and IPCInspection (Felt et al., 2011b) use different methods to solve re-delegation. Kirin (Enck et al., 2009) finds several dangerous permissions combinations. (Davi et al., 2011) try to overcome the over-privileged permissions. These papers identify many vulnerabilities, we define some of them as the metrics used in our research.

A few papers discuss the Android security evolution. The paper (Wei et al., 2012) studies the evolution of the Android ecosystem to understand whether the permission model is allowing the platform and its applications to become more secure. While in our paper, we use multiple categories metrics to define the security evolution, include the vulnerabilities, the malicious behaviors and the application metadata.

9.2 Malware Detection Using Data Mining

Data mining techniques are widely used in malware detection. While (Sami et al., 2010), (Wang et al., 2003) and (Kolter and Maloof, 2004) apply data mining to detect the malware in Windows operating system, we use data mining techniques in
the Android OS.

In the paper (Aung and Zaw, 2013), the authors list all the requested permissions as the feature vectors to detect the Android malware. They compare the performances of three classifiers. While we also use classification algorithms to classify the malware, however, we not only use the requested permissions, but also use the used permissions. Moreover, we ranked the permissions by the percentage difference between malware and benign applications. Finally, we find using 40 requested permissions with the highest percentage difference can get the best performance.

In the paper (Aafer et al., 2013), the authors use the APIs as the feature vectors to do the classifications in the Android market. Different from their work, we not only use the APIs and the requested permissions but also use the application vulnerabilities and its malicious behaviors as the feature vectors. In our research, we select 90 feature vectors while they choose 169 feature vectors, which means that our model takes less time to build.

In the paper (Yerima et al., 2013), they use Bayesian classification to detect the Android malware detection. They choose the permissions and the API as the feature vectors. While we use four different classifiers and compare their performances.

The performance comparison is shown in the Table 9.1. The first two results are from others research. The rest two are from our research.

<table>
<thead>
<tr>
<th>Research Group</th>
<th>Accuracy</th>
<th>TPR</th>
<th>TNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Aung and Zaw, 2013)</td>
<td>91.58%</td>
<td>0.916</td>
<td>0.916</td>
</tr>
<tr>
<td>(Aafer et al., 2013)</td>
<td>99%</td>
<td>0.999</td>
<td>0.978</td>
</tr>
<tr>
<td>(Yerima et al., 2013)</td>
<td>92.1%</td>
<td>0.906</td>
<td>0.937</td>
</tr>
<tr>
<td>Permission-based</td>
<td>96.5%</td>
<td>0.972</td>
<td>0.955</td>
</tr>
<tr>
<td>Multiple metrics</td>
<td>99.3%</td>
<td>0.995</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 9.1: Performance comparison among different research groups
9.3 Update Attack

Zhou (Zhou and Jiang, 2012) says that more recent Android malware families are adopting update attacks and drive-by downloads to infect users. Update attack detection is more tricky than common malware detection. Now the most popular way to detect malware is to use the signature, which is not susceptible to update attack detection.

In the paper (Tenenboim-Chekina et al., 2013), the authors use network behavioral analysis to identify the update attack. However, in our research, we use data mining techniques to detect the update attack. We have two ideas about detecting the update attack. For a single application, we get the difference between the versions, then use this difference to compare with the malware. If we have a bunch of old versions of the applications and malware, we use the new versions of the applications as the test dataset to detect the update attack.
Chapter 10

Future work

To analyze the application evolution, currently, we only use the number to represent each metric in vulnerabilities and malicious behaviors and use subtraction to get the result. In the future, we can get the details of each metric, then compare this metric in different versions to see the modification.

Since we get the best performance in multiple metrics malware detection. We can find more metrics related to vulnerabilities and malicious behaviors to increase the performance of the classification model.

We tried to get a basic idea of update attacks in this research. In the future, it’s planned to pay more attention to the modified parts between versions of an application. Then the metrics extract from the modified parts instead of from the whole APK files.

In the malware detection part, the accuracy, TPR and TNR are already high enough. However, we need to pay more attention for the false classifications and find out the deep reasons.
Chapter 11

Conclusion

To analyze the application evolution, we find that the security of applications doesn’t increase with the development of the applications. Even as the applications expose more security issues than the previous versions, regardless of whether the applications are from the Google Play or from the third-party market.

In permissions-based malware detection, the permissions which are used more frequently in malware are listed. The performances of using the requested permissions or the used permissions are similar. However, using the requested permissions will be faster and can be used precisely.

In multiple metrics malware detection, the performance is much better than only using permissions. Furthermore, the model can be used to predict whether the unknown applications are benign applications or malware by using the vulnerabilities, the malicious behaviors, the sensitive APIs and the permissions requested.

Finally, in the update attack detection, we present the classification models by using the difference values of metrics between versions of an application. The classification models are effective to detect the update attack detection, especially the ability of detecting applications which contain malware in the new version.
References


Kelly, G. (2014). Report: 97% of mobile malware is on android. this is the easy way you stay safe. [Online; accessed Mar 24, 2014].


Vita

Wenjie Shi was born in China in 1991. She enrolled in the Software Engineering Bachelor’s program at Huazhong University of Science and Technology, which she successfully completed in 2013.

After graduation, she entered Boston University’s Metropolitan College in Fall of 2013. She will receive her Master’s degree in Computer Science in May 2015.

Email Address:
wjshi@bu.edu