Looking Back! Using Early Versions of Android Apps as Attack Vectors

Yue Zhang[ID], Jian Weng[ID], *Member, IEEE*, Jiasi Weng[ID], Lin Hou, Anjia Yang[ID], *Member, IEEE*, Ming Li[ID], Yang Xiang[ID], *Senior Member, IEEE*, and Robert H. Deng[ID], *Fellow, IEEE*

Abstract—Android platform is gaining explosive popularity. This leads developers to invest resources to maintain the upward trajectory of the demand. Unfortunately, as the profit potential grows higher, the chances of these Apps getting attacked also get higher. Therefore, developers improved the security of their Apps, which limits attackers ability to compromise upgraded versions of the Apps. However, developers cannot enhance the security of earlier versions that have been released on the Play Store. The earlier versions of the App can be subject to reverse engineering and other attacks. In this paper, we find that attackers can use these earlier versions as attack vectors, which threatens well protected upgraded versions. We show how to attack the upgraded versions of some popular Apps, including Facebook, Sina Weibo and Qihoo360-Cloud-Driven by analyzing the vulnerabilities existing in their earlier versions. We design and implement a tool named DroidSkynet to analyze and find out vulnerable apps from the Play Store. Among 1,500 mainstream Apps collected from the real world, our DroidSkynet indicates the success rate of attacking an App using an earlier version is 34 percent. We also explore possible mitigation solutions to achieve a balance between utility and security of the App update process.

Index Terms—Android, early version, attack vector, reverse engineering, code protection

✦

1 INTRODUCTION

WITH the burgeoning popularity of smartphones, the Android operating system has captured an important part of the total market-share [1]. This is owing to Android's easy-to-join application development community for developers and vendors. Now, the developers extend their service to the mobile domain by creating a lot of applications (Apps). As of now, there have been more than 2.6 million Apps on GooglePlay [2], and total download count has crossed 197 billion [3].

However, the popularity of Android Apps makes them the prime attack targets [4], [5], which may severely undermine users' information security and privacy. For example, the Man-in-the-Middle attack (MITM) can be launched on a network enabled apps, which can give access to the user's information in the cloud [6] to the attacker without user's awareness. Reverse Engineering [7] allows an attacker to explore the source code and extract the security-related algorithms, which helps them design the attack vectors

easier. For example, an attacker can launch the repackage attack based on the source codes. The repackage attack can put apps under grave threats. In this attack, attackers inject a payload into an App to repack a trojanized version, making a benign App malicious.

To enhance security in Android Apps, developers and vendors have adopted a variety of approaches over recent years. To prevent the MITM attack, developers/vendors protected their Apps' network communications by using technologies such as TLS/SSL or Access Control [8], [9]. To defend their apps to be reverse engineered, developers guarded their Apps' source code with code protection methods, such as Java-Native-Interface (JNI) [10] and layout obfuscation [11].

Honestly, these countermeasures make Apps difficult to compromise. But as all these countermeasures are applied only to the "upgraded versions", the older versions of these apps remains vulnerable. For example, the famous social media App facebook has more than 200 versions [12], some of them might not be as secure as we want them to be.

The earlier versions and the upgraded versions may share similar or exactly the same functionalities. For example, despite some Game Apps (e.g., Angry birds, 2,048 etc.) may constantly get updated with new features, including addition of new roles or new scenes to make it more interesting, but the basic idea or the rule of how to play the game might not change that much. Meanwhile, to ensure development efficiency, the newer versions of the App might not be that "new" at all. They may build on the top of the earlier ones. Therefore, their source code may also be similar.

Based on this observation, we ask a different questions in this paper: instead of analyzing an upgraded App which is

- Y. Zhang, J. Weng, J. Weng, L. Hou, A. Yang, and M. Li are with the College of Informatin Science and Technology / College of Cyber security, National Joint Engineering Research Center of Network Security Detection and Protection Technology, and Guangdong Key Laboratory of Data Security and Privacy Preserving, Jinan University, Guangzhou 510632, China. E-mail: {zyueinfosec, cryptjweng, Wengjiasi, linhou19, anjiayang, limjnu} @gmail.com.
- Y. Xiang is with the School of Software and Electrical Engineering, Swinburne University of Technology, Hawthorn, VIC 3122, Australia. E-mail: yxiang@swin.edu.au.
- R.H. Deng is with the School of Information Systems, Singapore Management University, Singapore188065. E-mail: robertdeng@smu.edu.sg.

ultimately secure, what can an attacker still do by analyzing its previous vulnerable versions? Is there any vulnerabilities that exists in earlier versions that will also work on these upgrade versions? If so, how to explore it? In fact, in the case of Games, it is easy to visualize the threat caused by the leakage of the source code, even of an earlier version. If the Game company's competitor can obtain the source code and algorithm by reverse engineering an earlier version of the Game App, they may pirate a new App, which threatens the intellectual property of original Game company. In the following sections, we will show that attackers could launch more serious attacks through exploiting vulnerabilities existing inside earlier versions.

Our study inspects how such attacks could be launched. In our motivation example, two types of attack vectors are involved. They are, weak interface analysis and security related algorithm (or keys) analysis. In weak interface analysis, we explore some insecure interfaces that exists in both earlier version and upgraded version. We demonstrate this type of attack based on 360 Cloud Drive. By analyzing the network request, we found an insecure Restful API [13] existing in earlier version on which we could launch the MITM and delete the files on the Cloud Drive, which could severely threaten security of the newer versions. In the security related algorithm (or keys) analysis, we get the HMAC key and algorithm that are used in App authentication on the poorly protected Apps. Using this, an attacker may build a trojanized version.

To better understand this problem, we introduce our tool DroidSkynet, which can determine whether an App is suffering from the threats brought by earlier versions. Our tool accepts multiple versions of a given App, and if possible extracts the source code from them. The runnable analysis determines if an earlier version is still running properly, while the similarity analysis returns whether an earlier App is similar to its corresponding upgraded one. If both are yes, vulnerability analysis returns the possibility that an early version can be used as the attack vector.

The main contributions of our work are as follows:

1) We demonstrate that the poorly protected early versions of Apps could be used as attack vectors to attack the corresponding upgraded versions. Three attack instances are presented including Facebook, Sina Weibo, and Qihoo 360 Cloud Driven.
2) We design a detection strategy and implement a detection tool DroidSkynet based on this strategy, which analyses the risk brought by the earlier versions.
3) We run DroidSkynet on 1,500 Android Apps collected from the Internet, then find that most of the earlier versions of the apps have poor protection, and the success rate of attacking an App using an earlier version is 34 percent. To address this problem, we propose defense remedies to mitigate the threats imposed by the earlier versions.

2 BACKGROUND AND THREAT MODEL

In this section, we describe the necessary concepts and terminologies related to Android App protection, and we outline the threat model that our work is based on.

2.1 Reverse Engineering and Program Code Protection

The following concepts of reverse engineering and program code protection on the Android platform are important in the context of our work.

2.1.1 Android Reverse Engineering

Android reverse engineering is a process to extract and reconstruct the original Java source code out of Android Apps [14]. The information about the functionalities and implementation details of an App can be obtained by Android reverse engineering, which needs some specific tools such as APKTool [15], dex2jar [16] and AXMLPrinter2.jar [7] to help obtain source codes and resource files from an App or a binary executable file.

2.1.2 Android Program Code Protection

The opposite of Android reverse engineering is the Android program code protection. Android program protection is an efficient technique which prevents attackers from parsing source codes. It protects the source code by making it difficult to extract the source codes and resource files.

There are many methods to implement the Android program code protection. For example, Java native interface is a standard used code protection method. It provides interfaces between Java codes and native codes [10] and allows an Android App to execute C/C++ codes in the Java codes layer. By JNI technologies, developers package the core business logic in the C/C++ library and then call it in the Java code layer. JNI makes the reverse engineering difficult since C/C++ code is much harder to be reverse engineered than the Java codes. As another famous code protection method, layout obfuscation [11] can effectively obstruct the reverse engineering process. In detail, source codes are replaced by a series of meaningless characters, which makes source codes unreadable for human but readable for machines.

2.2 Threat Model

Without loss of generality, we assume the following threat model. To launch attacks, an attacker needs to find an early version which could be utilized by attackers to carry a malicious payload. The attacks extract specific code lines or other useful information from the poor protected early versions, which then can be used as building blocks of other attacks.

In our paper, we mainly focus on two types of vulnerabilities, which correspond to two kinds of attacks. (i) Weak interfaces that are existing both in earlier versions and upgraded versions. By using the weak interfaces, the attacker can explore the design fault of Apps and launch attacks. Particularly, we focus on MITM attacks. (ii) Revealing Security Related Algorithms or Keys from early versions. As an attacker, he can use these security-related algorithms or keys to analyze and break the authenticate protocol between an App and its server. He also can obtain an essential part of source codes to build trojanized versions.

3 MOTIVATING EXAMPLES

In this section, we introduce attack instances on three widely used Apps, in which the attackers can compromise the upgraded version of these Apps through early ones.

3.1 Weak Interfaces

Upgraded releases use encryption to protect network traffic from replay attack and eavesdropping attack, while its earlier versions may be vulnerable to various attacks. Moreover, a weak Restful APIs [13] used in the early versions may also be available, which may poses threats to the updated versions. In Web security, if a poorly designed interface has been explored, a hacker can compromise the entire website utilizing this interface.

QIhoo360 [17] has been devoted to providing high-quality security services for Chinese Internet users. Facing threats coming from trojan horse, virus and rogue software, QIhoo360 constructs a security ecosystem to identify and eliminate malicious behavior. And with the development of the QIhoo360, its service has covered almost all aspects of security. For example, 360 Cloud Drive which provides a secure storage service to a user, is one of the successful products.

Although 360 Cloud Drive enforces the security of the upgraded versions, the early ones are still exposed to threats. We now demonstrate the workflow of our attack. The goal of our attack is to find an insecure Restful API existing in an early version and still works, then use it as an attack vector to launch a MITM attack. In our experiment, we use "Fiddler", a client-side proxy-based tool, to analyze the network traffic and replay user's operation sessions. The tool with an out-of-the-box functionality of replaying a specific request, helping us to process our experiment smoothly. We apply the tool to many versions of 360 Cloud Drive and trigger each operation manually, including downloading files, deleting files, renaming files. With the help of such an experimental comparison, an insecure Restful API is exposed. This Restful API is designed to delete unnecessary files in the user's cloud storage initially. It can be found in the version 1.2.2 but has been removed from the version 7.1.0. By using this insecure Restful API, the attacker can delete files. Individually, when a user attempts to delete one file, we act as a man in the middle and block the request. After we obtain a request, we tamper it by using the insecure Restful API, which can lead the user to delete another file. More seriously, by editing and replaying the package, attackers are trivial to carry out such attacks, even without the user's awareness.

3.2 Revealing the Security Related Algorithms or Keys

One motivation for using the Code Protection is to avoid the security-related algorithms or keys being leaked. Once attackers have obtained such algorithms or keys, attackers may use them to launch various attacks. Although the earlier versions may be different from the newer ones, attackers still can extract some design idea of the protocols from the earlier ones. No companies would like to restart their Apps from zero every-time they release a new version. If the attackers are lucky enough, they may even get a key from the earlier versions, and this key is still being used in the upgraded versions.

User authentication is significantly crucial to Apps, while we found that there are vulnerabilities of authentication code fragments in our targeted Apps. Generally, the codes implementing user authentication use salted password hashing to enhance security. Salted password hashing is a preferable way to prevent passwords guessing attacks [18]. The implementation process of salted password hashing is described as follows: when a user inputs its password and presses the login button, a salt is retrieved from hard-coded source code file and appended with user's input password to create a new hashed password used to protect the original input password. The salt acts as the HMAC key [19], which avoids the modification of the hashed password. Besides, salted password hashing is also a way to authenticate a client's identity. For example, suppose that this mechanism is not guaranteed; an attacker could extract all features of authentication just through the network flow. The extracted features then can be used to forge a fake client. Our goal is to obtain the secret salt hard-coded in the early version, which could be used to launch attacks above.

Sina Weibo is a Twitter-like platform for information share and exchange. It is one of the most popular social networks in the world, which makes its users enjoy the convenience heartily brought by the Internet. Worldwide, Weibo has 222 million monthly active users, while the daily active users is up to 100 million [20].

Through our observation, we found that when a client of Sina Weibo App attempts to connect its server, the server will receive authentication parameters from the client. There are three authentication parameters required, including username, password and a string of letters and numbers. Through our analysis, we found that the third parameter plays an essential role in authentication. Without it or replacing it with another one during the authentication, the login request will be refused by server. For a specific user, this parameter will not change as long as the user does not change his password. We want to find out how this parameter can be generated.

For these upgraded versions of Weibo, the reverse engineering process becomes powerless due to the usage of heavy code protection. It motivates us to examine its early versions. We choose early versions which are running properly. Running properly indicates that the authentication mechanism used in the upgraded versions is the same as the one in the early versions.

After manual efforts, we found the early version of Sina Weibo (V2.4.0_1) has poor code protection. Note that, the login request will be sent only after user inputs the identity credentials and presses the *login button*. Thus, we take the login button-pressed action as a breakthrough point. To capture such an action, we need to find the relationship between the source code and user interaction. To this end, we first search the layout files to find out where the intended *login button* is. The layout file offers a human-readable structure for describing UI screens. Each layout file corresponds to a *Activity*. The *Activity* contains a set of functions which can be executed when a particular event occurs on the UI described by the corresponding layout file. After we found the function triggered by the action of pressing the *login button*, we make this function as an entry point. Start from this entry point, we trace other functions called by this function. We split each function by its function name and return value. The function names expose their functionality, which could help us lock our target function more quickly. For example, the function "getLoginResponseContent" is to get the response from the
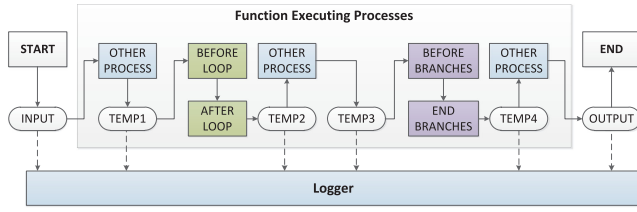
Fig. 1. The executing process of "Logger".

server. After several failed attempts, we found the algorithm used to generate the third parameter mentioned above. The algorithm takes a username, a password, and a regular salt as inputs, then feeds these inputs to an MD5 function. Listing 1 illustrates the detailed implementation of the algorithm. It can be observed that a static string named "KEY" is used as the salt of MD5 hash function. This string can be extracted by reverse engineering from the class "Constants".

Listing 1. Detailed Implementation of the Generated-Algorithm

---

```
1  private static String calculateS(String param)
   {
3    char[] arrayOfChar = MD5.hexdigest(paramString +
            Constants.KEY).toCharArray();
     return arrayOfChar[1] + arrayOfChar[5] +
            arrayOfChar[2] + arrayOfChar[10] +
            arrayOfChar[17] + arrayOfChar[9] +
            arrayOfChar[25] + arrayOfChar[27];
5  }
```

Facebook. Facebook constitutes one of the largest social networks in the world that owns a large user group. Worldwide, Facebook has about 1.79 billion monthly active users (MAUs) [21]. With so many users, the security of its client becomes particularly important, which attracts the attention of researchers.

Authentication mechanism of Facebook is similar to Weibo, which requires several parameters to build a valid URL. Failed to reverse engineer the upgraded version (the version 9.2), we attempt to reverse engineer its early release to extract the core engine. Unfortunately, the reverse engineering process is not as easy as that of Weibo, since part of the source codes of Facebook's earlier versions has also been obfuscated.

To overcome this limitation, we use App repackage technique to build a malicious App. Attackers often use the repackage technique to inject a malicious payload into a normal App. We program a log-record payload, named "Logger", and inject it into the Facebook (version 1.9.2). Logger registers the listener in the original methods of Facebook (version 1.9.2), by which it can record the values of interested, including inputs, outputs and temp values generated of a specific function.

The executing process of Logger is shown in Fig. 1. By injecting the "Logger" into several early versions (three versions have been tested, including "V1.91", "V1.6" and "V2.2"), we restructure an algorithm of interested. The algorithm used in Facebook takes a username and a password

as the dynamic inputs, a static string named "signatureKey" as the salt of MD5 hash function, and finally returns the hash value of MD5 hash algorithm.

4 ANALYSIS STRATEGY

By observing the aforementioned attack examples, we summarize the analysis and attack steps as follows:

1) *Early versions discovery.* For a given upgraded App, the attacker needs to find its early versions. Based on these, the attacker can perform his analysis.

2) *Runnable analysis.* When the attacker explores an insecure interface, the attacker needs to select early versions that run properly from all early ones he found. An earlier version running properly indicates that a weak interface is still being used.

3) *Similarity analysis.* The attacker needs to ensure the early version he finds are similar to the corresponding upgrade one. A high similarity between an earlier version and an upgrade one indicates that the developer does not change the functions of this App too much, by which an attacker may still extract useful information from the earlier version (i.e., security related functions or keys).

4) *Vulnerability analysis.* The attacker needs to further narrow down the selected early versions to only those that are vulnerable.

5) *Attack.* Attackers analyze the common functionalities between a vulnerable earlier version and an upgraded version. The vulnerabilities can be explored in this process.

Algorithm 1. Analysis the Success Rate of Attacking an App

Input: S (an Apps' different versions)
Output: $compromisableRate$ (the success rate of attacking an App)

```
1   S_cpr()=∅;
2   for i = 1; i ≤ m; do
3       S^i_sorted = sortByReleasedTime(S);
4       n = sizeof(S^i_sorted);
5       for j = 1; j ≤ n; do
6           a_new S^i_sorted.pop(n − 1);
7           a_j = S^i_sorted.pop(j);
8           if isRunnable(a_j) then
9               rra_j=getSimilarity(a_j, a_new);
10              pra_j=getVunRate(a_j);
11              cpra_j= rra_j * pra_j;
12              S_cpr().add(cpra_j);
13:
14          end
15          j=j+1;
16      end
17      i=i+1;
18  end
19  return getMax(S_cpr);
```

There have been a lot of related works exploring vulnerabilities from poorly protected Apps, and thus the last

TABLE 1
Function List

Function Name	Description
$reqNet(a)$	returns **TRUE** when an App a requires a network connection.
$getNumURL(a)$	returns the total URL numbers existing in source codes of an App a.
$getNumAvaURL(a)$	returns the available URL numbers existing in source codes of an App a.
$containSAURL(a)$	returns **TRUE** when the source code of an App contains sensitive (S) URLs which are also available (A).
$getCreateTime(a)$	returns the creation time of an App a.
$S.push(a)$	pushes an element a into a set S.
$S.pop(i)$	returns an element's index i from a set S.
$sizeof(S)$	returns the number of elements of a set S.
$getMax(S)$	returns the Max value of a set S.
$sortByReleasedTime(S)$	returns a sorted set S^s of a set of App S. in S^s, an App's index will be lower if its released time is earlier.
$getSimilarity(a, b)$	returns the similarity between App a and b.
$isRunable(a)$	returns **TRUE** when an App a is running properly.
$getVunRate(a)$	returns a value that indicates the possibility of attacking an App a.

phase is out of the scope of this paper. We concentrate on the first four phases to identify early versions that are potential to be candidates.

We give the basic algorithm as shown in Algorithm 1. Before we start to describe our algorithm, Table 1 lists some functions which are used. We will detail each phase in the following sections.

4.1 Early Versions Discovery

To identify whether an App is an early one, we should know the right time when this App was released. In general, the creation time of an App is the time users download it, which does not include the information about when the App's developer releases it. However, some downloaded page has recorded the published time. On the other hand, Through experiments, we find that APK files can be unpacked by using unpacking software and the unpacked files indicate creation time of the App.

4.2 Runnable Analysis

Recall that attackers need to know the early versions he found are running properly. Our idea based on a simple observation, which enables us to know an App is running properly or not. Apps can be classified into two categories: those use the internet connection, and those do not. For an App does not use the internet, it will always work whenever a user installs it since Android System is downward compatible. For an App uses the internet, if the server-side stops

providing the service to the App, the App would stop working. For these Apps, we also need to take the server-side into consideration. Based on this observation, we propose our algorithm in Algorithm 2 ($isRunnable(a)$).

Algorithm 2. Judge Whether an Early Version Runs Properly (isRunnable (a))

    Input: a_{early}
    Output: Boolean value: ISRUNNABLE
1   ISRUNNABLE= **FALSE**;
2   **if** $reqNet(a_{early})$ **then**
3   **if**
        $((getNumAvaURL(a_{early})/getNumURL(a_{early}) \geq$
        THRESHOLD) OR ($containSAURL(a_{early})$)) **then**
4   ISRUNNABLE= **TRUE**;
5   return ISRUNNABLE;
6   **end**
7   ISRUNNABLE= **FALSE**;
8   return ISRUNNABLE;
9   **end**
10   **else**
11   ISRUNNABLE=**TRUE**;
12   return ISRUNNABLE;
13   **end**

In Algorithm 2, we define the threshold to be 50 percent, which represents the ratio of available URLs in total URLs found in an App's source codes. We regard the App is running properly if the ratio of its available URLs in total URLs is larger than the threshold.

The function $containSAURL(a)$ will return the value *TRUE* if source codes of an App a contain sensitive URLs. Specifically, a URL is sensitive if it contains the following information:

- *User action.* During the process users requesting service from a server, user action represents the command which a user requests the server to execute. Such action falls into two classes: operation request and authentication request. Operation request means users want to operate their resource on the server, such as deleting files or viewing files. Authentication request includes the login request and registration request.
- *Action sink.* Similarly, the sink of the action is considered sensitive. We want to know which resources users want to operate.
- *Identity.* Users' identity information exists in the system, such as phone number, account, email address, and password.
- *Hash value.* As discussed earlier, password hashing with salt is a preferable way to protect the integrity of a password. A hash value of the password and username can be attached in the URL.

For example, suppose that there is a URL in such a format: http://a.com?un=a&pwd=b&key=c&op=delete&fn=1. txt. In this example, a user with a username "a" and a password "b" wants to delete (op=delete) a file named "1.txt" on his cloudDrive. The parameter key is a generated hash. In this case, deleting the file is the user action, while the file name is the Action sink. Username and password is his identity information.

4.3 Similarity Analysis

To know the similarity between an early version and a new one, we introduce SimiDroid to analyze Apps. SimiDroid [22] is a framework for comparing Android Apps. It can identify and explain the similarity and changes between different Apps. It comes with three out-of-box plugins, which allows researchers to compare with two Apps on different levels. These plugins are method-based comparison, resource-based comparison, and component-based comparison. We focus on the method-based comparison since it will give us the information about whether the developer has changed its functionality of Apps or not. The method-based comparison can extract all the method signatures and abstract representations of statements from two different Apps, then calculate the similarity between the two Apps. These signatures and descriptions are not only based on constant values (names of method or parameters) but also based on its structures. Therefore, even if the developer uses the layout obfuscation technology to protect its source code, the result will not be impacted.

4.4 Vulnerability Analysis

We should ensure that attackers can compromise an alternate early version. What makes an App vulnerable? First, the source code is not heavily protected. Otherwise, the attacker can not obtain enough information from it. Second, it does not use cryptographic algorithms to preserve its essential parameters.

4.4.1 Source Code Protection Analysis

To address the first problem, we need to know how well an App protects its source code. We first classify Android program code protection techniques into two types: Logic protection (LP) and File protection (FP). LP protects logic functionalities of source code files by reducing the readability of source code. The most frequent methods of LP is layout obfuscation. FP prevents source code files from being leaked in an App file. It is a method to prevent attackers from extracting all source code files. The most common technique of FP is JNI. Therefore, we concentrated on JNI technology in this research.

Logic Protection Handling. The vulnerability of an App is closely related to the readability of its source code. We define that when the layout obfuscation (LP) is applied, the layout obfuscation rate is one of the important factors which determines if an App can be compromised. The layout obfuscation rate R_{LP} can be calculated using the formula below: $R_{LP} = \frac{ClassNum_{LP}}{ClassNum_{LP} + ClassNum_{NOLP}}$. The $ClassNum_{LP}$ is the number of classes is obfuscated by developers in source code while $ClassNum_{NOLP}$ is the number of classes without obfuscation.

File Protection Handling. The JNI functions exposed to the upper layer is often elementary, which covers the underlying logic in its native layer. When the JNI (LP) is applied, an attacker can use the JNI-libs directly without understanding its logic. For some attackers, they do not care about whether the FP is used or not. For example, in the case of Facebook, we use the source code of the original App, including all the JNI-libs, to repackage a malicious App. During the whole process, we use the JNI-libs as building blocks of our malicious App without understanding them. However, for some other attackers, they may also care about the implementation of the application's JNI-libs. To cover these two cases, we introduce the impact factor of FP $IF_{FP}(IF_{FP} \leq 1)$. The IF_{FP} will be a part of the result, but its value can be customized. In other word, we can detect whether there is File Protection, but we will let the attackers or developers decide if it matters or not.

4.4.2 Cryptographyic Algorithms Analysis

As discussed earlier, the attacker can extract insecure interfaces from an earlier version. What makes an interface not secure? Insecure interfaces usually accept plain-text as their inputs and submit it directly without any cryptographic process. Based on this observation, we introduce taint-analysis technology to trace how the parameters pass through from one function to another. For Android applications, taint analysis enables us to build a path from a specific entry point, known as the source, to a particular exit point, known as the sink. To determine whether cryptographic algorithms are used, we taint return values of encryption APIs as the source and taint inputs of network APIs as the sink. Taint technology can identify if there are cryptographic algorithms applied to secure the interfaces in a given App, but it can not decide how many parameters submitted by this interface are. Therefore, we introduce the impact factor of IF_{crypto} to address this problem. A developer or an attacker can custom this value.

We now can decide whether an App is a vulnerable one. Specifically, we present the definition of the $VunRate$, which indicates the possibility of attacking an App a. The $VunRate$ can be described as following: $VunRate = (1 - R_{LP})((1 - IF_{FP}) * (1 - IF_{crypto}))$.

5 IMPLEMENTATION EXPERIENCE: DROIDSKYNET

Based on Algorithm 1, we implement a tool that can identify whether an App is suffering the potential security risks. We name the tool DroidSkynet.[1] With the assistance of Droid-Skynet, we are trying to ring the alarm to developers that attackers could launch an attack with the help of the early versions. Furthermore, DroidSkynet can also distinguish which kind of code protection technique is used on a given App and give us an idea of trends about App protection techniques.

5.1 Overview Design

In this section, we give an overview of "DroidSkynet" and describe the key techniques employed in our framework. Fig. 2 shows the entire workflow of "DroidSkynet".

DroidSkynet consists of five major components: fundamental information analysis, source code extractor, runnable analysis, similarity analysis, vulnerability analysis.

---

1. The word "Skynet" comes from the famous movie "Terminator". In this movie, an artificial intelligence named "Skynet" comes back from the future by using a time machine. Its mission is to kill the young leader of the resistance and halt humanity. Although our DroidSkynet cannot go back in time, it can find attack payloads exiting in the early version and use it to launch attacks for updated versions.
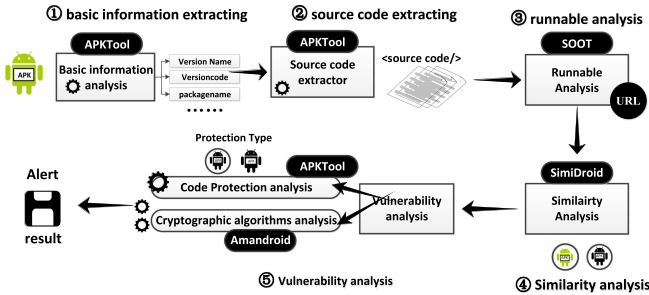
Fig. 2. Work flow of DroidSkynet.

DroidSkynet accepts multiple versions of an App and returns the possibility of attacking this App. To perform an analysis, the DroidSkynet needs to compare the latest version with each early versions one by one. Specifically, The basic information analysis component parses the "AndroidMainfest.xml" files inside the latest version and an earlier version, then extracts the basic information from them. The information includes the package name, version code, and version name. The basic information of the latest one will be cached in the memory and kept for the later usage. This part enables us to know which App and which version we are testing on, as well as the release time of each App. Source code extractor extracts source codes from an APK file. The runnable analysis determines whether the given App is a runnable one using Algorithm 2. These two components only work with the early versions. After that, if the early version of the application is runnable, we use the tool of similarity analysis to analyze the similarity between the lasted one and the earlier one. At the same time, vulnerability analysis component analyzes whether the earlier version is being protected. It returns a value that indicates the possibility of attacking the previous version of the App. Finally, after analyzing all the versions, the tool will combine the analyzed results above, and give the conclusion on whether the App is suffering from the threat from earlier versions.

5.2 Core Engine Implementation

Among all the components, basic information analysis, source code extractor does not contain too many challenging issues, while the similarity analysis is based on the Simi-Droid. Thus, in this section, we will give more details about how we implement runnable analysis and vulnerability analysis.

5.2.1 Runnable Analysis

For an App that does not use the internet, it will always work whenever a user installs it, while Apps that use the internet, we need to test whether the URLs used in the Apps are available. To identify whether an App uses the internet or not, we parse the "AndroidMainfest.xml" file of an Android App. This file records run-time permissions of an App. If an App uses the internet, it must register on the permission tree in this file. In other word, if the string "uses-permission android:name="android.permission. INTERNET"" can be found in the file "AndroidMainfest. xml", the App may use the network connection. On the other hand, since Apps often require excessive permission,

an App with the INTERNET permission does not mean that the App needs network connection undoubtedly [23]. To address this problem, we introduce the static analysis framework *Soot* to search the Internet-related APIs [24]. Soot is a Java optimization framework that can process Android source code from a given App. It can convert the Android source code to a Java instance. The functions $getApplicationClasses()$ and $getMethods()$ can extract the classes and the functions. We search each class to find whether the internet related APIs has been used, such as $getInputStream()$ and $openConnection()$.

On the other hand, DroidSkynet goes through the source code files and returns the number of available links. To identify whether these found URLs are available, DroidSkynet follows the semantics of the HTTP status codes for communicating status information in HTTP. We link each URL found in the source codes, which represents the outcome of the connection action. According to the status codes, we can determine whether the connection is available. For example, the status code 200 is a response to a successful request, while the status code 404 indicates that the server has not found anything matching the Request-URL. Actually, although some URL requests may depend on the other previous requests, the status code also is still an acceptable solution. For example, the data downloading request can only be sent when the authentication request has successfully passed through. When the authentication fails, the status code would be 401. The similar status codes include 403 and 407. In other words, if we obtain the status codes above, we can know that the request URL is in protection but still available. On the other hand, according to our previous discussion, running properly cannot be easily determined by analyzing the number of available links. The privacy-related URLs should also be taken into account. Each URL represents a potential sensitive source.

5.2.2 Vulnerability Analysis

Protection Method Analysis. Protection method analysis identifies which protection types it used. One essential module of the component protection method analysis is APKTool, which is also the critical factor in designing the source code extractor. APKTool is a tool to reverse engineer Android Apps. It decodes resources into the almost original form, such as AndroidMainfest.xml and layout files. At the same time, it parses the "classex.dex" inside the APK file, and changes the DEX file into SMALI files.

After the source code extracting, we start our analysis to identify the LP and FP. To determine whether source code protected by LP, we highlight identification methods of layout obfuscation. Our basic idea is checking whether the names of extracted source code files satisfy the principle we predefined. We take filenames as identification information since filenames can fully retain characters of source codes from original programs. After the process of reverse engineering, source code files are located in the folder "smali". Names of original classes are transformed into filenames in this folder, and package names have turned to the directory names. In our experiment, we only consider the class name as the identifying indication to check the layout obfuscation, since it rarely happens that an App obfuscates its valuable name in classes but leaves the class names as the plain-text.

Layout obfuscation often uses meaningless or non-understandable valuable names instead of meaningful ones. For example, after being obfuscated by tools, the code's valuable name is made up of lower-case letters which counted from one to three (taking "abc" as an instance). The regular expression [25] will match a word in a specific format, and the technology used to search for and substitute words satisfies the predefined rules. There are ready-made algorithms provided by the Java library to find these obfuscated words. The regular expression of such words can be expressed as "$[a-z]\{1,2,3\}\$$".

On the other hand, there are some obfuscated code lines, such as some function names, which do not satisfy such a regular expression in Apps. These obfuscated function names use some special characters to replace original variables, which makes the first regular expression failed to match. There exists a generic naming rule called camel-case naming to name a Java class [26]. Practically, most of the class names that do use the camel-case naming rule are authored by developers, not by the machine. Based on such a fact, we construct a function to identify class names that satisfy the camel-case pattern. We use this function only after the first regular expression fails to match, to improve the performance on the running time. If the program source codes match the first type of regular expression, the codes are protected by obfuscated methods which imply that it is unnecessary for us to go on the second type of regular expression match. If changing the order of this two functions and putting the second type ahead, we need to loop over each class name and check if it satisfies the camel-case naming rule and only after all the looping is done can we get the result.

For these Apps that use the FP to protect their source code, we focus on these using the JNI technology. If the JNI technology is used, we can find C/C++ libraries for the native interface called in reverse engineered source code. And those libraries often appear at the same location in the App (in the folder "lib" at the directory root) and use the ".so" ending with their filenames.

It is noteworthy that there are also many Apps using some third-party libraries for service (e.g., for LBS service) or profits (e.g., show the AD views). In this case, the App itself is not protected by the protection method, but its libraries use some program code protection. We think that an App like this should not be considered as protected since the libraries are not used to build their core logic or publicly accessible. These libraries are excluded when we determine whether an App is protected by protection methods. To achieve our goal, we collecte the names of commonly used third-party libraries in a database. All libraries found will be abandoned if it is also in the database.

Finally, we introduce the feedback component into our analysis to decrease the false rate. According to the results from vulnerability analysis, there exists a rule that the upgraded Apps released are more likely with less vulnerability rather than the early ones. For example, if we found an App that was published in 2011 is secure, but its upgraded version that released in 2018 is not secure, there may be something wrong with our analysis. In this case, we will mark these samples, and will perform a manual effort thereafter to confirm the result.
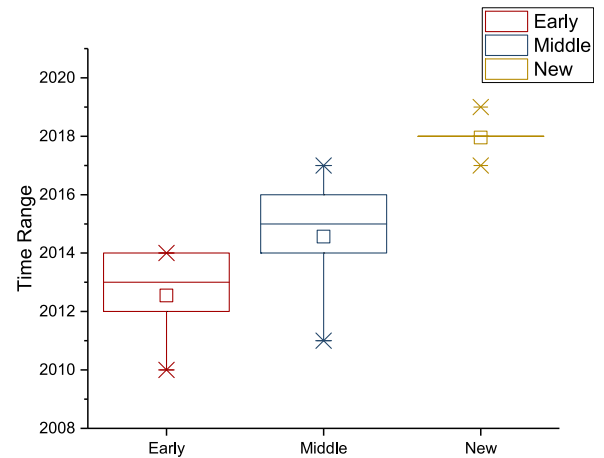

Fig. 3. Released time of each branch.

Cryptographic Algorithms Analysis. Taint-analysis enables us to determine whether cryptographic algorithms are used, we taint return values of encryption APIs as the source and taint inputs of network APIs as the sink. Our tool extends the Amandroid framework [27], which provides prerequisites for a taint analysis. We now introduce how we extend Amandroid for our purpose. Particularly, by customizing the profile "TaintSourcesAndSinks", we can trace the taint paths of interest. We care about the following sources and sinks. For the sources, we care about the functions that can process the encryption or decryption, such as *cipher.doFinal*(). The taint sinks involve network APIs and functions with the capabilities to post data to the server. For example, we taint the system network function *urlConnection.getOutputStream().write*().

6 EXPERIMENTS AND EVALUATION

6.1 Experimental Samples Collection

Principles to select appropriate Apps are required. The time span is a filter condition. However, the time partition where the branches are located cannot be easily determined by the year of an App released. For example, when some Apps are born, another App has already been released for several versions, even if they are in the same category. At the same time, for Apps released recently, samples for experiments may be not enough, which leads to the lack of intensive comparative analysis. Thus, we collect three versions of each app. The latest version, the earliest version we can found and the version that released in the middle time.

Based on this principle, we collect 1500 Apps from the top 10 categories with 150 in each category, across the three branches (50 in each branch). The 10 categories are: Games, Players, Browsers, Download-managers, Finance, Fitness, Message, Tools, Personal, Social. We collected samples from the Google Play and some third-party markets, such as UptoDown [28]. Fig. 3 shows the released time of each branch. It can be observed that most early versions were released in 2013 and middle versions were released at 2015, while the upgraded version was released at 2018.

6.2 Runnable Analysis Evaluation

We first perform the runnable analysis. To this end, Droid-Skynet needs to check an App's network usage and how

TABLE 2
The Number Distribution of the URLs in Different Categories

Categories	Early Version	Middle Version	New Version	Average
Game	154	195	222	190
Player	170	205	254	209
Browser	196	191	196	194
Download-Managers	145	141	151	145
Finance	180	184	187	183
Fitness	150	200	209	186
Message	161	378	446	328
Tools	156	149	158	154
Personal	111	115	115	113
Social	157	263	329	249
Average	158	202.1	226.7	195
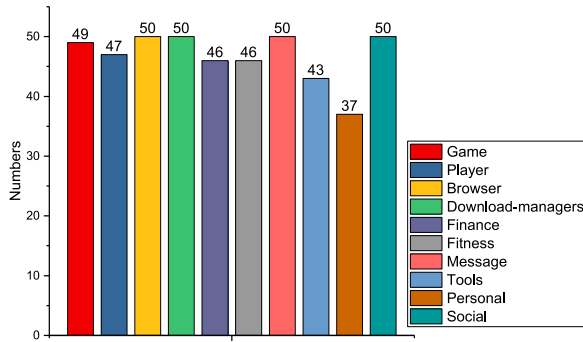
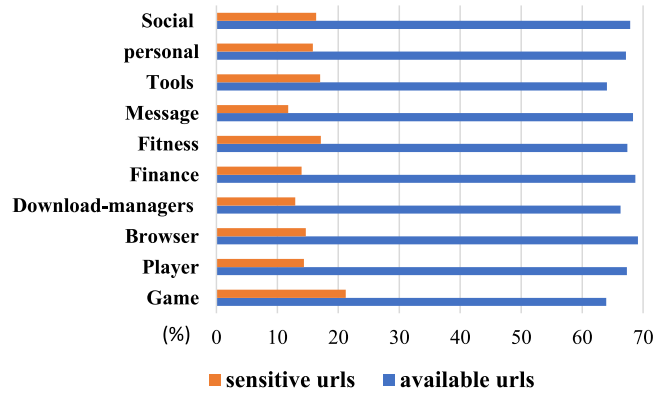Fig. 4. The distribution of network usage in each category.

Fig. 5. The average ratio of sensitive URLs and available URLs in each category (early branch).

Fig. 6. The average ratio of sensitive URLs and available URLs in each category (middle branch).

many URLs are still available. Table 2 shows the number of distribution of the URLs in different categories. It can be observed that, for most apps that use the internet, the number of URLs existing in one App is around 200 on average. The Message Apps contain more URLs than any other categories. The number of URLs existing in one App is around 330 on average. One app named "Email TypeApp" even contains 711 URLs. The numbers of URLs are gradually increasing by versions, which indicates that Apps becoming more and more complicated.

Fig. 4 provides a detailed picture of the Apps' network usage. From the figure, we can learn that most Apps need a network connection. Specifically, among 500 early versions, there are only 64 Apps that do not need the network connection. According to the previous discussion, these Apps are considered as runnable ones.

In Figs. 5 and 6, we show the average ratio of available URLs and sensitive URLs. From the figures, we can see that each category has more than 50 percent of URLs that are still available now. The middle categories have much higher radios. In the cases of Personal, Fitness Downloader-managers and Players are even exceeds 70 percent. This indicates that most functionalities of these Apps are running properly. Among all the available URLs, there are almost 30 percent considered as sensitive ones. All these URLs that may contain sensitive operations, such as log in the server and delete users' files.

6.3 Similarity Analysis Evaluation

We then perform the similarity analysis. To this end, Droid-Skynet combines all three branches, and compares the

similarity between different versions. We first check methods shared by three different versions. We considered two cases here: (i) these methods are existing in all the three versions with no change. (ii) these methods are existing in all three versions that are similar to each other. The latter one is used to describe methods shared the similar method signature and execution logic. These comparison methods are provided by SimiDroid. Fig. 7 shows the result. On average, for each app in each category, there are more than 23,000 methods existing in all the three versions without change. Compared with it, similar methods are much less. This indicates that developers tend to use the methods with no change than re-implement it. Among all the Apps, Games
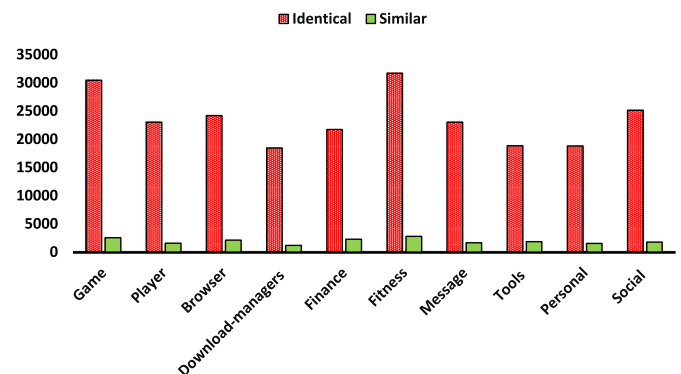

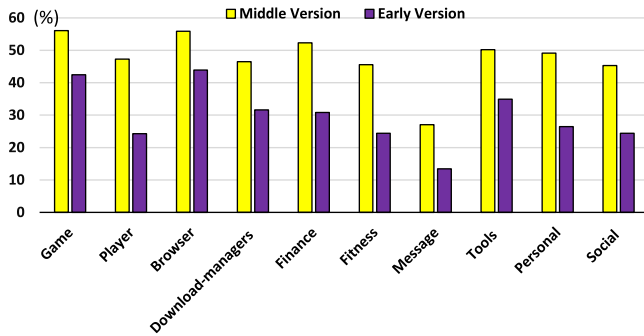Fig. 7. The distribution of methods shared by three different versions.

Fig. 8. The average similarity between early version and upgrade version and that between middle version and upgrade version.

Fig. 9. Apps without any protection.

Fig. 10. Apps with FP protection.

Fig. 11. Apps with LP protection.

have more identical and similar methods than any other categories, since Games usually are more complicated than other Apps.

We then check the similarity between the early version and the upgrade version, and that between the middle version and the upgrade version. Fig. 8 shows the result. On average, for each app in each category, the similarity between an early version and an upgrade version is around 30 percent (29.7 percent), while that between a middle version and an upgrade version is around 50 percent (47.3 percent). Among all the Apps, Games have the larger similarities between each version, when compared with other Apps. This confirms our statement in the introduction section. Games may add new roles or new scenes to make it more interesting. The basic idea of how to play may not change that much. On the other hand, Message Apps, such as Wechat and Whatsapp, are less similar to their corresponding upgraded versions. For these Apps, users have higher requirements on the communication speed and user experience, even security. Therefore, the developers of these Apps would like to improve these features to fulfill the user's requirements.

6.4 Vulnerability Analysis Evaluation

To demonstrate that DroidSkynet provides high-quality intelligence about potential attack detection, we use a manual method to evaluate it. The samples for evaluation are across the Fitness category, which includes 150 Apps in total. In our manual method, we concerned the accuracy of the protection method analysis. In our experiment, the DroidSkynet can detect all LP and FP without error rate. The reason because we only care about layout protection and JNI protection, which are relatively easy to be identified. In our future work, our tool will support more complex LP and FP features.

We first show the protection methods used in all three branches in Figs. 9, 10 and 11. It can be observed in Fig. 9 that Apps without any protections are gradually declining. As time passes, the numbers without any protection are even falling down to zero in some categories, which indicates that today's developers value the code protection more than before. Thus, compromising an upgraded App is much harder than compromising an early one. The most rapidly declining category is Finance, and we infer that these Apps involve payment functions, which makes developers have to strengthen code protection to keep attackers away. However, for others such as Games and Messages,
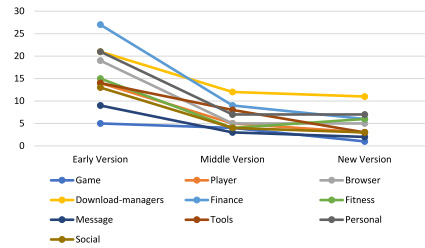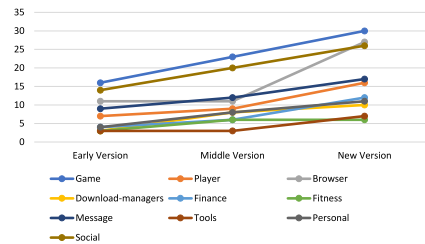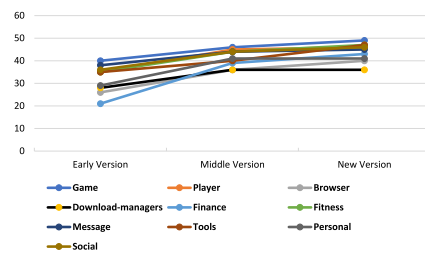
their slopes are more smooth, and they have a low position at the beginning. These indicate that their codes are well protected at the beginning of development. The reasons for those Apps in such categories often are developed by enterprises rather than individual developers. The enterprises think highly of the security of their products and possess plenty of available software development assets, which makes the creation of individual developers show no advantage in such areas. In contrast, creativity and innovation characteristics make more sense than security in Personal apps which leads the slope of such categories to decrease more slowly while their start points are higher, compared with other categories.

From Figs. 10 and 11, we can see that LP is generally a preferred protection method in the early stage. The reason for this is that, as a commonly used LP method, layout obfuscation has been applied widely in other program code protection fields. Thus, it is trivial to apply LP into Android code protection. All three figures indicate that although Apps are protected very well nowadays, it is possible to have an early version with poor protection.

Fig. 12 shows the cryptographic Protection methods used by Apps. It can be observed that Apps with cryptographic Protection are gradually increasing by versions, which indicates that compromising an upgraded App is much harder than compromising an early one. Among all Apps, the developers of Games would like to strengthen their Apps using the cryptographic methods. There are many cheating tools used
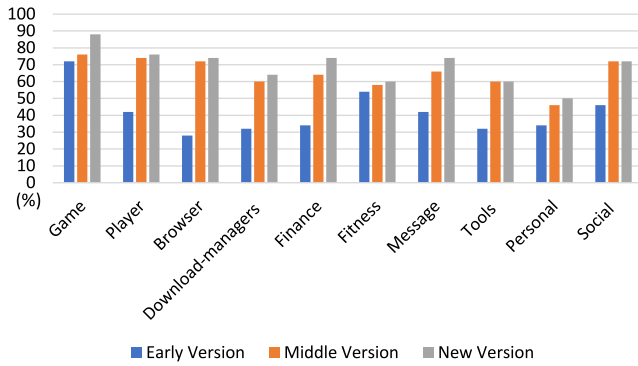
Fig. 12. The apps with cryptographyic protection.

TABLE 3
The Success Rate of Attacking an App Using an Early
Version or a Middle Version

Categories	Middle version	Early version
Game	24.6%	18.7%
Player	21.5%	10.6%
Browser	25.3%	20.3%
Download-managers	21.4%	14.2%
Finance	23.4%	14.5%
Fitness	20.3%	11.3%
Message	10.7%	5.1%
Tools	21.4%	15%
Personal	21.9%	11.5%
Social	19.9%	10.6%
Average	21%	13.1%

in mobile games and dozens of them for the popular ones, which breakS the balance of Games. Thus, involving the cryptographic methods is a way to deal with the balance issue.

Finally, based on all the above factors and Algorithm 1, we can calculate the success rate of attacking an App by using the early versions as attack vectors. In our case, The value of IF_{crypto} and IF_{FP} both are set to 0.5. Table 3 shows the result of our final result. DroidSkynet identifies that the success rates of attacking an App using an "middle" version and an early version are 21 and 13 percent, respectively. The success rate of attacking an App using an earlier version is 34 percent in total. All these affected Apps also can be exploited to passively either disclose various types of in-App data, such as hash salt, or poor APIs which may subsequently cause security problems. Although some categories in our experiment are not affected much, such as Message category, it does not mean that these Apps are secure enough. One reason for that is the detection results are relevant to the size of the sample set.

7 DISCUSSIONS

7.1 Potential Threats

We have already known that early versions of Apps can be used as attack vectors. When an early version is reverse engineered successfully, attackers could also know the following information:

1) *Core Engine.* The obvious examples are Games and Tools applications we mentioned in the previous part. Getting the program code of early versions will help attackers to rebuild a pirated App, which threatens the intellectual property of the original developers.
2) *User Interface Design.* Some software's UI has changed a little from the very beginning. Layout descriptions are texts showed in layout files located in the folder /res/layout/. By using reverse engineering, attackers could rip off the UI design of Apps.
3) *Programming Style.* The programming style is also the privacy of companies [29] which can be extracted from Apps. Few companies often change the programming style, while a good programming style cloud optimizes the efficiency of coding.
4) *Potential Vulnerabilities.* As we discussed before, access authentication and some weak interfaces can be used to launch attacks. What's more, some attackers may get elicited through the early versions, using it to crack the App, and bypassing the registration of a chargeable App.

7.2 The Root Cause of the Problem

The competition among enterprises becomes more intensive nowadays. An App without good maintainability and extendibility would not exist for a long time. Unfortunately, version upgrading is a complex process, which would cost a lot of financial cost and resources. To update the version more smoothly, developers should not only achieve functionality but also take the high cohesion and low coupling into consideration during the design process. Thus, when the vulnerability occurs, developers patch it by adjusting modules instead of rewriting the entire App, which makes the majority of old modules are reused as parts of the upgraded ones. Therefore, vulnerabilities existing in early versions still be kept in upgraded versions.

In terms of Users, old users may refuse to upgrade Apps because they cannot afford the high cost of computation and storage brought by upgraded versions providing more services. Some former users may not want to update their Apps since upgraded versions often require better devices, more computing power, and more storage space. Thus, early APIs used in old versions cannot be shut down directly, despite that they may be not secure enough.

7.3 Mitigation

It is difficult to prevent Apps suffering from such attacks since developers can not abandon users who don't want to upgrade their Apps. In our real life, some Apps would like to update their client side for every few weeks, while the server will still support early versions until several months later. This is the right balance spot between security and convenience.

Although some Apps cannot be compromised today, this does not mean that it will never be reverse engineered in the future. With the advancement of reverse engineering technology, Apps are thought to be secure at present, may be compromised some day. Thus, continuously keeping the developing process of program code protection and regularly making early APIs be shut down, are mitigation of such problems.

8 RELATED WORK

Android has become a popular platform in the smartphone area and has attracted considerable attention of researchers. The tools and the technologies we used in our work are also widely used in other works.

8.1 Obfuscation Detection

We now review the obfuscation detection technologies, since, in our paper, we designed our own obfuscation technology. Omid Mirzaei et al. [30] proposed AndrODet to detect identifier renaming, string encryption, and control flow of android applications. The online learning techniques have been applied, which enables the resource-limited environments can run it properly. Shuaike Dong et al. [31] designed efficient and lightweight detection models for identifier renaming, string encryption, Java reflection, and packing. After a large-scale investigation, they depict a holistic view of the usage of obfuscation. For example, malware authors prefer to use the string encryption. Yan Wang et al. [32] developed techniques to analyze obfuscator Identification for a given Android App. The method is based on machine learning and could determine whether the App was obfuscated and which obfuscator was used. They also explain how the obfuscator was configured. All these works are well addressed their own goals. However, their technologies cannot be applied to our scenario directly. A critical goal of our work is to evaluate how much source codes has been obfuscated in a single App, none of these tools can fulfill our requirement.

8.2 Reverse Engineering

Several researchers have considered reverse engineering as an essential tool in Android Apps analysis. In Android malware detection, reverse engineering plays an indispensable role, especially in Android static malware detection. To smoothly carry out the static analysis, analysts use reverse engineering to extract source codes in the start stage. For example, RiskRanker [33] tried to detect zero-day malware by analyzing the dangerous behaviors of Apps based on static analysis. The risky behaviors are defined based on the source codes extracted by reverse engineering. DroidAnalytics [34] proposed a solution to obtain signatures that can be used to develop an anti-virus software [35]. Work [36] focused on the UI design, which may make contributions to the repacking detection of malware. Only by reverse engineering, researches could get the layout files in Apps. There is one highlight of that reverse engineering is the base of machine learning-based malware detection [37], [38], [39], [40]. Apart from malware detection, the second area of related work includes information leakage on a mobile device. CHEX [41] and DroidSafe [42] focused on the inter-component communication in an App and utilize the reverse engineering as a building block to analyze whether Apps leak the privacy or suffer from the attack of component hi-jack. LeakMiner [43] and FlowDroid [44] considered the callback and CFG generated by the reverse engineering tools, by using the static taint analysis to target the information leaks. To detect which App leaks information, reverse engineering also provides an Android-supplied life cycle models, as the Dexteroid [45] did in their work.

8.3 Vulnerabilities and Attacks on Android Apps

Vulnerabilities and attacks on Apps are essential issues to research on Android Security. Many works emphasized on the attacks which are brought by the third-party components. For instance, LayerCake [46] tried to separate the untrusted third-party advertising components from App, which brings threats to users' privacy. Other works, value the security problems existing in Apps. Specifically, the vulnerable Apps may leak sensitive information or compromise the data integrity on Android devices. For example, AppSealer [47] and CHEX [41] focused on the component hi-jacking attacks in Android Apps, which are caused by unauthorized component interactions. SUPOR [48] used UI patterns to identify sensitive user input, which may be vulnerable to attack. Randroid [49] and Roee Hay' work [50] identified the architecture and communication patterns which pose potential threats for Apps. And Cao'work [51] focused on vulnerabilities and attacks associated with Android system services. All these attacks cared about a part or several parts of Android System or Apps but seldom took the early versions into consideration.

Chen's work [52] used different approaches but equally satisfactory results with that of our work's in analyzing the vulnerabilities and attack vectors in smartphones. We know that many iOS libraries have Android versions which are freely available, while IOS libraries are often closed source. Chen's work was based on an idea that analyzed Android libraries to understand relations between the libraries on both Android and IOS, in which way the attackers could learn behaviors of IOS libraries without getting its source code.

Finally, there are lots of works using runtime information or URLs of programs to analyze vulnerabilities in program [53], [54], [55]. Unlike these prior studies, employing those invariable features to analyze vulnerabilities on Android platform, is a kind of task that few people research on. However, Android is a widely used platform with new characteristics, which encourages us to carry on our research. For example, it is trivial to extract the source code of early Android Apps, while it is hard to process reverse engineering on upgraded versions. An executable program on Windows platform may not have such a factor since most of them are programmed by C/C++ language, which is very difficult to analyze. At the same time, Android platform appears just a few years, the short intervals between the released times of early versions and that of the upgraded versions make contributions to analyze vulnerabilities and launch attacks, which leads the core engines of them are similar.

9 CONCLUSION

In this paper, we define and study a problem, that is, early versions of an App can put its upgraded one at risk of attacks. Since early versions are trivial to be compromised by reverse engineering technology, vulnerabilities existing in early versions could be used as building blocks for launching more severe attacks on upgraded versions. In our motivation examples, we launch a MITM attack on 360 Cloud Driven and cracked the HMAC key from Facebook and Sina Weibo. Furthermore, by using reverse engineering technology, we design a static analyzer, DroidSkynet, to

detect such potential problems in a large number of Apps. DroidSkynet is evaluated with 1500 Apps collected from the real world. The experiment results show that most of these Apps are suffering from such problems. The success rate of attacking an App using an earlier version is 34 percent. We also put forward some suggestions to bridge the gap between utility and security of App update.

ACKNOWLEDGMENTS

REFERENCES

[1] M. Butler, "Android: Changing the mobile landscape," *IEEE Pervasive Comput.*, vol. 10, no. 1, pp. 4–7, 2011. [Online]. Available: http://dx.doi.org/10.1109/MPRV.2011.1

[2] Statistics, "Number of available applications in the Google Play store from december 2009 to december 2018," 2018. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, Accessed on: Feb. 15, 2019.

[3] T. S. Portal, "App download and usage statistics (2018)." [Online]. Available: http://www.businessofapps.com/data/app-statistics/, Accessed on: Dec. 24, 2018

[4] N. Asokan, V. Niemi, and K. Nyberg, "Man-in-the-middle in tunnelled authentication protocols," in *Proc. 11th Int. Workshop Secur. Protocols*, 2003, pp. 28–41.

[5] A. Yang, J. Weng, N. Cheng, J. Ni, X. Lin, and X. Shen, "DeQoS attack: Degrading quality of service in VANETs and its mitigation," *IEEE Trans. Veh. Technol.*, 2019.

[6] A. Yang, J. Xu, J. Weng, J. Zhou, and D. S. Wong, "Lightweight and privacy-preserving delegatable proofs of storage with data dynamics in cloud storage," *IEEE Trans. Cloud Comput.*, 2018.

[7] P. O. Fora, "Beginners guide to reverse engineering android apps," in *Proc. RSA Conf.*, 2014, pp. 21–22.

[8] E. Rescorla, *SSL and TLS: Designing and Building Secure Systems*, vol. 1. Reading, MA, USA: Addison-Wesley, 2001.

[9] J. Weng, J. Weng, Y. Zhang, W. Luo, and W. Lan, "BENBI: Scalable and dynamic access control on the northbound interface of SDN-based VANET," *IEEE Trans. Veh. Technol.*, vol. 68, no. 1, pp. 822–831, Jan. 2019.

[10] R. Gordon, *Essential JNI: Java Native Interface*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1998.

[11] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: Attack strategies and defense techniques," in *Proc. 4th Int. Symp. Eng. Secure Softw. Syst.*, 2012, pp. 106–120.

[12] UPtodown, "The early versions of Facebook," 2018. [Online]. Available: https://facebook.en.uptodown.com/android/old, Accessed on: Feb. 15, 2018.

[13] RESTfulAPI net, "RESTful API tutorial," 2018. [Online]. Available: https://restfulapi.net/, Accessed on: Dec. 25, 2018.

[14] T. Cannon, "Android reverse engineering," *Thomas Cannon*, 2010. [Online]. Available: https://thomascannon.net/android-reversing/, Accessed on: Aug. 5, 2019.

[15] A. Apvrille, "Android reverse engineering tools," 2012. [Online]. Available: https://ibotpeaches.github.io/Apktool/, Accessed on: Dec. 15, 2016.

[16] B. Alll and C. Tumbleson, "Dex2Jar: Tools to work with android .Dex and Java .Class files." 2010. [Online]. Available: https://www.kitploit.com/2018/10/dex2jar-tools-to-work-with-android-dex.html, Accessed on: Aug. 5, 2019.

[17] D. Evans, V. Y. Zhang, H. Chang, et al., "Analyzing competition among internet players: Qihoo 360 v. tencent," *Antitrust Chronicle*, vol. 12, p. 2, 2013, https://EconPapers.repec.org/RePEc:cpi:atchrn:12.2.2013:i=13033.

[18] T. Kwon and J. Song, "Secure agreement scheme for gxy via password authentication," *Electron. Lett.*, vol. 35, no. 11, pp. 892–893, 1999.

[19] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," *Network Working Group RFC*, 1997, pp. 1–6.

[20] eMarketer, "Weibo reaches 100 million daily users." [Online]. Available: https://www.emarketer.com/Article/Weibo-Reaches-100-Million-Daily-Users/1013449

[21] T. S. Portal, "Number of Facebook users worldwide 2008–2016." [Online]. Available: https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/

[22] L. Li, T. F. Bissyandé, and J. Klein, "SimiDroid: Identifying and explaining similarities in android apps," in *Proc. IEEE Trustcom/BigDataSE/ICESS*, 2017, pp. 136–143.

[23] S. Mansfield-Devine, "Android malware and mitigations," *Netw. Secur.*, vol. 2012, no. 11, pp. 12–20, 2012.

[24] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: Converting android Dalvik bytecode to Jimple for static analysis with Soot," in *Proc. ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, 2012, pp. 27–38.

[25] K. Thompson, "Programming techniques: Regular expression search algorithm," *Commun. ACM*, vol. 11, no. 6, pp. 419–422, 1968.

[26] G. L. Williams, "The camel case," *Law Quart. Rev.*, vol. 56, 1940, Art. no. 254.

[27] F. Wei, S. Roy, X. Ou, et al., "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1329–1341.

[28] UPtodown, "The UptoDown market," 2018. [Online]. Available: https://en.uptodown.com/android, Accessed on: Dec. 15, 2018.

[29] L. Church, J. Anderson, J. Bonneau, and F. Stajano, "Privacy stories: Confidence in privacy behaviors through end user programming," in *Proc. 5th Symp. Usable Privacy Secur.*, 2009, Art. no. 20.

[30] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 356–367.

[31] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild," *Int. Conf. Secur. and Privacy in Communication Systems*, pp. 172–192, 2018.

[32] Y. Wang and A. Rountev, "Who changed you?: Obfuscator identification for android," in *Proc. 4th Int. Conf. Mobile Softw. Eng. Syst.*, 2017, pp. 154–164.

[33] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day android malware detection," in *Proc. 10th Int. Conf. Mobile Syst. Appl. Services*, 2012, pp. 281–294.

[34] M. Zheng, M. Sun, and J. C. S. Lui, "Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware," in *Proc. 12th IEEE Int. Conf. Trust Secur. Privacy Comput. Commun./ 11th IEEE Int. Symp. Parallel Distrib. Process. Appl./ 12th IEEE Int. Conf. Ubiquitous Comput. Commun.*, 2013, pp. 163–171.

[35] A. Jain, H. Gonzalez, and N. Stakhanova, "Enriching reverse engineering through visual exploration of android binaries," in *Proc. 5th Program Protection Reverse Eng. Workshop*, 2015, pp. 9:1–9:9.

[36] C. Yang, C. Zuo, S. Guo, C. Hu, and L. Cui, "UI ripping in android: Reverse engineering of graphical user interfaces and its application," in *Proc. IEEE Conf. Collaboration Internet Comput.*, 2015, pp. 160–167.

[37] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and explainable detection of android malware in your pocket," in *Proc. 21st Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 35–40.

[38] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual API dependency graphs," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 1105–1116.

[39] R. K. Shahzad and N. Lavesson, "Veto-based malware detection," in *Proc. 7th Int. Conf. Availability Rel. Secur.*, Aug. 2012, pp. 47–54.

[40] J. Wen, K. Wu, and C. Tellambura, "A closed-form symbol error rate analysis for successive interference cancellation decoders," in *Proc. IEEE Int. Conf. Commun.*, 2017, pp. 1–6.

[41] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 229–240.

[42] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in DroidSafe," in *Proc. 22nd Annu. Netw. Distrib. Syst. Secur. Symp.*, vol.15, p. 110, 2015.

[43] Z. Yang and M. Yang, "LeakMiner: Detect information leakage on android with static taint analysis," in *Proc. 3rd World Congr. Softw. Eng.*, 2012, pp. 101–104.

[44] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2014, Art. no. 29.

[45] M. Junaid, D. Liu, and D. C. Kung, "Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models," *Comput. Secur.*, vol. 59, pp. 92–117, 2016.

[46] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *Proc. 22th USENIX Secur. Symp.*, 2013, pp. 97–112.

[47] M. Zhang and H. Yin, "AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications," in *Proc. 21st Annu. Netw. Distrib. Syst. Secur. Symp.*, pp. 45–61, 2016.

[48] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "SUPOR: Precise and scalable sensitive user input detection for android apps," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 977–992.

[49] B. R. Schmerl, J. Gennari, J. Cámara, and D. Garlan, "Raindroid: A system for run-time mitigation of android intent vulnerabilities [poster]," in *Proc. Symp. Bootcamp Sci. Secur.*, 2016, pp. 115–117.

[50] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of interapplication communication vulnerabilities in android," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 118–128.

[51] C. Cao, N. Gao, P. Liu, and J. Xiang, "Towards analyzing the input validation vulnerabilities associated with android system services," in *Proc. 31st Annu. Comput. Secur. Appl. Conf.*, 2015, pp. 361–370.

[52] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou, "Following Devil's footprints: Cross-platform analysis of potentially harmful libraries on android and iOS," in *Proc. IEEE Symp. Secur. Privacy*, May 2016, pp. 357–376.

[53] Y. Kataoka, D. Notkin, M. D. Ernst, and W. G. Griswold, "Automated support for program refactoring using invariants," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2001, Art. no. 736.

[54] D. R. Engler, D. Y. Chen, and A. Chou, "Bugs as inconsistent behavior: A general approach to inferring errors in systems code," in *Proc. 18th ACM Symp. Operating Syst. Principles*, 2001, pp. 57–72.

[55] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proc. 24th Int. Conf. Softw. Eng.*, 2002, pp. 291–301.
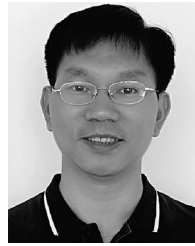
Jian Weng received the BS and MS degrees from the South China University of Technology, in 2001 and 2004, respectively, and the PhD degree from Shanghai Jiao Tong University, in 2008. He is a professor and the executive dean with the College of Information Science and Technology, Jinan University. His research areas include public key cryptography, cloud security, blockchain, etc. He has published 80 papers in international conferences and journals such as CRYPTO, EUROCRYPT, ASIACRYPT, the *IEEE Transactions on Cloud Computing*, PKC, CT-RSA, the *IEEE Transactions on Dependable and Secure Computing*, etc. He also serves as associate editor of the *IEEE Transactions on Vehicular Technology*. He is a member of the IEEE.

Jiasi Weng received the BS degree in software engineering from South China Agriculture University, in June 2016, and the graduate degree in technology of computer application from Jinan University, in September 2016. Her research interests include cryptography and information security, blockchain, and security in software defined network.

Lin Hou received the bachelor's degree from the Wuhan University of Technology, and the dual degree from the Huazhong University of Science and Technology. She is working toward the PhD degree at Jinan University. Her research mainly focuses on asymmetric cryptography and privacy.

Anjia Yang received the BS degree from Jilin University, in 2011, and the PhD degree from the City University of Hong Kong, in 2015. He is currently a postdoctoral researcher with Jinan University, Guangzhou. His research interests include blockchain security, RFID security and privacy, applied cryptography, and cloud computing. He is a member of the IEEE.

Ming Li received the BS degree in electronic information engineering from the University of South China, in 2009, and the MS degree in information processing from Northwestern Polytechnical University, in 2012. From 2016, he is working toward the PhD degree at Jinan University. His research interests include crowdsourcing, blockchain and its privacy and security.

Yue Zhang received the BS and MS degrees in information security from the Xi'an University of Posts & Telecommunications, in 2013 and 2016, respectively. From 2016, he is working toward the PhD degree at Jinan University. His research interests include bluetooth, system security, and android security.

Yang Xiang received the PhD degree in computer science from Deakin University, Australia. He is currently a full professor with the School of Software and Electrical Engineering, Swinburne University of Technology, Australia. His research interests include cyber security, which covers network and system security, data analytics, distributed systems, and networking. In particular, he is currently leading his team developing active defense systems against large-scale distributed network attacks. He is the chief investigator of several projects in network and system security, funded by the Australian Research Council (ARC). He has published more than 200 research papers in many international journals and conferences. He served as the associate editor of the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, the *Security and Communication Networks* (Wiley), and the editor of the *Journal of Network and Computer Applications*. He is the coordinator, Asia for IEEE Computer Society Technical Committee on Distributed Processing (TCDP). He is a senior member of the IEEE.

Robert H. Deng received the MS degree from the National University of Defense Technology, China, in 1981, and the BS and PhD degrees from the Illinois Institute of Technology, in 1983 and 1985, respectively. He has been a professor with the School of Information Systems, Singapore Management University, since 2004. Prior to this, he was the principal scientist and manager with the Infocomm Security Department, Institute for Infocomm Research, Singapore. His research interests include data security and privacy, multimedia security, network and system security. He was an associate editor of the *IEEE Transactions on Information Forensics and Security* from 2009 to 2012. He is currently an associate editor of the *IEEE Transactions on Dependable and Secure Computing*, and member of editorial board of the *Journal of Computer Science and Technology* (the Chinese Academy of Sciences), and the *International Journal of Information Security* (Springer), respectively. He is the chair in the Steering Committee of the ACM Symposium on Information, Computer and Communications Security (ASIACCS). He received the University Outstanding Researcher Award from the National University of Singapore in 1999 and the Lee Kuan Yew fellow for Research Excellence from the Singapore Management University in 2006. He was named Community Service Star and Showcased Senior Information Security Professional by (ISC)2 under its Asia-Pacific Information Security Leadership Achievements program in 2010. He is a fellow of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.