Journal of Information Processing Vol.27 593–602 (Sep. 2019)

[DOI: 10.2197/ipsjjip.27.593]

Regular Paper

Understanding the Origins of Weak Cryptographic Algorithms Used for Signing Android Apps

Kanae Yoshida[1] Hironori Imai[1,a)] Nana Serizawa[2] Tatsuya Mori[2] Akira Kanaoka[1,b)]

Received: November 28, 2018, Accepted: June 11, 2019

Abstract: Android applications are digitally signed using developers' signing keys. Since each key is associated with a developer, it can be used to establish trust between applications published by the author, i.e., apps signed with the same key are allowed to update themselves if package names are identical, or access each other's resources. However, if a signature is generated using a weak algorithm such as MD5, then apps signed with the corresponding key are exposed to several risks, such as hijacking apps with fake updates or granting permissions to a malicious app. In this work, we analyze several Android apps to identify the threats caused by using weak algorithms. Our study uncovered the following findings: Of the more than one million apps collected from Google Play, 223 and 52,866 were digitally signed using the weak algorithms of 512-bit RSA key and MD5, respectively. We identified the causal mechanisms for generating certificates that employ weak algorithms, and found that these mechanisms can be attributed to app-building frameworks and online app-building services. On the basis of these findings, we provide guidelines for stakeholders of the Android app distribution ecosystem.

Keywords: Android, security, digital signature, cryptographic algorithms

1. Introduction

According to the report from StatCounter, mobile Internet usage surpassed desktop usage in 2016 [1]. Security threats have become critical for mobile platforms, because mobile devices collect and keep privacy-sensitive data, which attackers may find valuable. Thus, individual mobile users must be sufficiently knowledgeable and aware of the security risks when installing a new app. As a countermeasure against malware, it is a common practice for mobile users to *not* install an app published by an untrusted party.

While security risks caused by malicious codes have been well studied and put into practical use, mobile apps have other *invisible* threats that exist outside of the code, such as developer certificates. An Android application is digitally signed with a developer's signing key, which can be verified through the corresponding developer certificate. The certificate is used to establish trust between applications published by the developer. Apps signed with the same signing key are allowed to update if their package names are identical, or access each other's resources. Therefore, if a signature is generated using a weak cryptographic algorithm such as MD5, apps signed with the corresponding signing key are exposed to several risks, such as hijacking apps with fake updates or granting permission to a malicious app.

In this study, we answer the following research question:
RQ: *What are the origins of the use of weak cryptographic algorithms for signing Android apps?*

1 Toho University, Funabashi, Chiba 274–8510, Japan
2 Waseda University, Shinjuku, Tokyo 169–8555, Japan
a) 6518002i@st.toho-u.ac.jp
b) akira.kanaoka@is.sci.toho-u.ac.jp

We employed a large-scale, systematic-measurement study using one million Android apps collected from Google Play and five million Android apps collected from third-party marketplaces. For Google Play apps, we also employed temporal analysis to assess the changes in certificates over time. We extracted the digital certificates from the collected apps and examined the cryptographic algorithms used to sign them. We also analyzed several statistics, such as key lengths configured for cryptographic algorithms, validity periods of certificates, and a number of installations of apps signed using the weak cryptographic algorithms.

Our chief contributions can be summarized as follows:
- We clarified the threats caused by the use of weak cryptographic algorithms for signing Android apps (Section 4).
- Using a massive number of apps in the wild, we revealed that there are a non-negligible number of apps with security risks, because they were signed using a 512-bit RSA key or their signatures were generated with MD5; these algorithms are known to be vulnerable to attacks (Section 5).
- We identified the causal mechanisms that lead developers to use either the 512-bit RSA key or MD5 (Section 6).
- From the findings derived from the analysis, we provide proposals for the stakeholders, i.e., Android OS developers, Android market managers, and Android application developers (Section 7.2).

The rest of the paper is organized as follows. Section 3 presents the overview of the digital certificates used in the Android platform. In Section 4, we clarify the possible security threats that are caused by the use of weak cryptographic algorithms for signing Android apps. In Section 5, we present our findings through large-scale measurement and analysis. We identify the apps signed with weak cryptographic algorithms and their characteristics. In Section 6, we identify the causal mechanisms that lead developers to use weak cryptographic algorithms. In Section 7, we discuss the guidelines for thwarting the threats. Section 8 concludes our work.

2. Related Works

In this study, an investigation was conducted on the current Android APK, and the threat posed by the application was properly identified. Such large-scale surveys include Zmap, proposed by Durumeric et al. [2]. Zmap makes it possible to quickly crawl the Internet, which affects various studies. However, it does not correspond to the signature of Android's APK targeted by this research.

Martin et al. [3] surveyed literature thoroughly for a broader understanding of the Android market in addition to security. This study revealed that there has been no investigation of Android's APK signature.

Research on SSL/TLS was conducted as a large-scale survey related to encryption. Zmap also targeted SSL/TLS. However, Durumeric et al.'s study focused on SSL/TLS certificates [4], which included the relationship between the certificate authority and its trust relationship. Application of the results was expected, but most of the certificates used for Android's APK were self-signed certificates, as mentioned above. Therefore, the trust concerning the issuance of certificates was not discussed from the perspective of PKI in this study.

Many studies related to cryptographic threats related to Android applications were conducted. However, none of them targeted the APK signature.

Analysis of cryptographic APIs used by applications [5], [6], pseudorandom number generators [7], and imperfect SSL/TLS implementation [8], [9], [10], [11], [12], [13], [14] focused on cryptographic use inside applications.

Fahl et al. conducted the first survey of signatures on the Android APK [15]. The validity period of certificates, key length, and cryptographic algorithms for many Android APKs was shown. It identified problems on signature usage in Android applications. The results indicated that the discussion of threats and risks was limited. More in-depth exploration of its threats and risks are still required as well as countermeasures.

3. Overview of the Developer Certificates Used on the Android Platform

In this section, we review the developer certificate used on the Android platform.

3.1 Role of the Developer Certificates

Digital certificates are used to establish trust between applications published by a signer, that is, the developer of the apps. The digital certificates are introduced to reliably accomplish the following three functionalities.

Updating Apps When installing a new version of the application, the digital signer must be the same as the signer of the signature used in the current application. In other words, the digital certificate ensures that an attacker cannot let an end user who owns a device install a malicious version of a targeted app, even though the malicious version has the same package name as the targeted app.

App Modularity Android allows application packages (APKs) signed by the same certificate to run in the same process. If the applications request it, the system can then treat them as a single application. When AndroidManifest.xml files in each application have the same sharedUserID value and the applications are signed using the same signing key, then these applications belong to the same process. Applications can share data access and processing.

Code/Data Sharing Android provides signature-based permission enforcement. An application can expose functionality to another application that is signed with a specified certificate. By signing multiple APKs using the same signing key and signature-based permissions checks, applications can securely share code and data. There are four protection levels (**normal**, **dangerous**, **signature**, and **signatureOrSystem**) for permission. These limit the scope of use of given permission. Permissions with protection level **signature** and **signatureOrSystem** allow other applications signed using the same signing key to use this permission.

Certificates used for signing APK files do not have to be issued from a widely trusted certification authority (CA). Developers use the self-signed certificate issued by the developer or organization without problems. Therefore, an Android user cannot trust the signer of an app using only a given certificate. In the Google Play market, trust between developers and end users is established through the registration of developers, that is, all developers are required to register themselves using their personal information, including contact address. Thus, the primary role of the developer certificate is to guarantee the integrity of data and the authors of apps, but not to identify who the authors are.

3.2 Signing Apps

Android applications are digitally signed. Two types of signatures exist: those applied to the APK using the signing key and those applied to the certificates which correspond to the signing keys of APKs. All applications are signed and the certificates are attached in the APK file. The application developer owns the signing key for this certificate.

3.3 Requirements and Recommendations for Certificates

Google has made several requirements and recommendations for developers to publish an application on Google Play. First, two public-key cryptography algorithms are allowed for signing: RSA and DSA. Second, a validity period is required for the public key ending after October 22, 2033, also, is recommended for more than 25 years [16].

4. Threat Analysis

In this section, we first discuss several threats caused by using weak algorithms for signatures.

4.1 Impersonating a Targeted Developer

If a malicious user can impersonate a targeted developer, the malicious user could perform the following attacks:

Injecting Malicious Updates An attacker can allow an end user to install a malicious version, which overwrites the current one. It is also possible to update an application that repackages the current application as a new version, where the repackaged version includes malicious code.

Identical Modularization An attacker can run a malicious application in the same module as the target application. Malicious applications have access to various process details of the target application.

Abusing Permissions An attacker can prepare another application and use the permissions of the attack-target application.

4.2 Creation of Arbitrary Signatures

An attacker can create arbitrary signatures in several ways.

Key Leakage If An attacker obtains a signing key by leakage, any signature can be created.

Recovering the Signing Key If the signing key can be recovered from the verification key, a malicious user can create an arbitrary signature. Valenta et al. indicated that a 512-bit RSA key could be recovered within four hours at a cost of $75 [17]. The National Institute of Standards and Technology (NIST) does not currently recommend the 1,024-bit RSA key [18]. Since Google Play recommends a certificate validity period of 25 years or more, there is sufficient time for an attacker to recover even a 1,024-bit RSA key.

Collision Attack One potential mode of attack on hash functions is collision attacks, which searches for two messages that have the same hash value. A conflict with the MD5 algorithm has already been discovered [19]. As a more advanced attack method, the chosen-prefix collision attack was proposed [20], and a forged certificate using the collision was generated [21]. In 2017, a collision with SHA-1 was discovered [22].

In the case where a collision attack is used against a signature, two conflicting messages will be identified before granting a signature. Following this, one of the messages is allowed to sign. Therefore, collision attacks cannot be used for the purpose to which the signature was already given.

Second Pre-Image Attack The other mode of attack on hash functions is the second pre-image attack. The attack finds another message M', that will have the same hash value as $H(M)$, of a certain message, M. This is more difficult to enact than a collision attack. At present, successful cases of this attack have not been found for hash functions, including MD5 and SHA-1. If the attack succeeds on a hash value that has already been signed, the APK signature is critically affected.

If a target of an attack is a certificate, the target hash value for causing a collision attack or a second pre-image attack is limited to a single value. On the other hand, when the target becomes the APK, multiple hash values can be selected. Since the developer releases multiple applications, we can collect multiple hash values. Likewise, if the developer has been updating multiple times and it has been observed, we can also obtain different hash values for each version of that application. In the case of an attack targeting the APK, the possibility of attack success becomes much higher, since there are multiple hash values for the attack targets.

5. Analysis of Digital Certificates

PlayDrone is a system for collecting applications on Google Play proposed by Viennot et al. [23]. The APKs were downloaded and used for analysis. The number of APKs used for the analysis is 1,177,599. In this section, we conduct an analysis of the signature status. A similar analysis was previously performed by Fahl et al. [15], which focused on other aspects.

Table 1 Public key cryptography algorithm and key length used for the signature in the PlayDrone dataset.

Algorithm and Key Length	Number	Proportion
RSA4096	3,077	0.26%
RSA2048	580,134	49.24%
RSA1024	567,055	48.13%
RSA512	223	0.02%
DSA1024	26,697	2.27%
Others	5,002	0.08%

Table 2 Hash function used for APK signatures.

Algorithm	Number	Proportion
SHA-256	680	0.06%
SHA-1	1,123,779	95.45%
MD5	52,866	4.49%

5.1 Public Key Cryptography Algorithm and Key Length

Table 1 presents the public key cryptographic algorithm used for the signature of the APK file and the key length. "RSA1024" indicates that RSA is used as the public key cryptographic algorithm and the key length is 1,024 bits. Since multiple signatures are assigned to some APKs, the total number of keys exceeds the total number of APKs.

In RSA, 2,048-bit keys are the most frequently used, accounting for 49.24% of the total. Consequently, the 1,024-bit RSA key accounts for 48.13% of the total. A total of 223 APKs used the 512-bit RSA key.

5.2 Hash Function Used for APK Signing in the PlayDrone Dataset

Table 2 gives the examination results for the hash function used for the APK signature. SHA-1 is the most commonly used, with 95.45% of the total. MD5 is the second most commonly used at 4.49%.

5.3 Number of Downloads

The risk increases if an application signed using the weak cryptographic algorithm such as MD5 or 512-bit RSA key is well known. In this section, 52,866 applications that used MD5 for APK signing are focused. The number of downloads as an indicator that reflects its popularity is analyzed.

In the PlayDrone dataset, peripheral information of an application on Google Play, such as the number of downloads and genres, is provided as metadata in the JSON file format, separate from the APK data. Therefore, the metadata of the PlayDrone dataset is analyzed. The number of downloads is not an accurate number. For example, "1,500+" indicates that the number of downloads is 1,500 or more, and "10,000+" indicates that the number of downloads is 10,000 or more.

In this research, metadata collected from October 30–31, 2014 was used, and the number of downloads was investigated. The metadata dataset is divided into plural data blocks, and each block is serialized. The dataset on October 31, 2014, contained 256

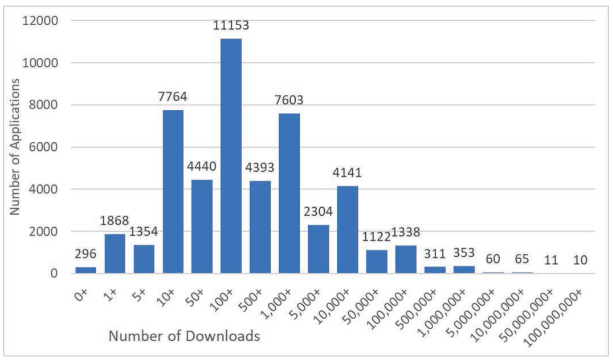

Fig. 1 Number of downloads of aplications that used MD5 for APK signatures in the PlayDrone dataset.

Table 3 Number of downloads of applications that used MD5 for APK signatures.

Downloads	Applications	Prop.
0+	296	0.61%
1+	1,868	3.84%
5+	1,354	2.79%
10+	7,764	15.98%
50+	4,440	9.14%
100+	11,153	22.96%
500+	4,393	9.04%
1,000+	7,603	15.65%
5,000+	2,304	4.74%
1,0000+	4,141	8.52%
5,0000+	1,122	2.31%
100,000+	1,338	2.75%
500,000+	311	0.64%
1,000,000+	353	0.73%
5,000,000+	60	0.12%
10,000,000+	65	0.13%
50,000,000+	11	0.02%
100,000,000+	10	0.02%

dataset blocks from “00” to “ff.” However, the “24” and “30” datasets were corrupted, so the shortfall was addressed from the October 30, 2014 dataset.

There are 499 applications with more than 1 million downloads, and 21 applications exceeding 50 million downloads (**Fig. 1**). According to Statista, there were approximately 1.86 billion Android users worldwide as of 2015 [*1]. Vulnerable algorithms are used for applications used by 1% or more of all users. The greater the spread of applications using MD5 for APK signing, the greater the threat of data leakage, etc. Detailed results are given in **Table 3**.

5.4 Temporal Analysis

The PlayDrone dataset was collected in 2014. A total of 52,866 APKs that used MD5 for APK signing in PlayDrone were reacquired under the same package name. A similar investigation was then made from 22,517 APKs obtained by reacquisition. The reacquisition of APK occurred from November 12–27, 2016. The results are given in **Table 4**.

The number of reacquired APKs was 22,517, which decreased from 52,833 original APKs. The cause of the decrease is that many download errors occur. When a download occurs, the server only issues a notification of the download. The reason for the download error is unknown. However, there are several reasons for errors. First, the application that was released when PlayDrone was acquired may not currently exist on the market. Second, the application may not correspond to the terminal used for downloading, which is the Nexus 7 (2012). Third, the application may not be distributed in the area where the account used for downloads belongs. An error may have occurred other than those mentioned above, causing the download to fail. Thus, it was not possible to reacquire all APKs that used MD5 for APK signature on the PlayDrone.

[*1] https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

Table 4 Hash function of the reaquired APKs that was signed using MD5 in 2014.

Algorithm	Number	Proportion
SHA-256	37	0.14%
SHA-1	2,801	10.34%
MD5	24,244	89.52%

Table 5 Signing key of the reaquired APKs that used the 512-bit RSA key in 2014.

Algorithm and Key Length	Number	Portion
RSA512	78	100.00%

Table 6 Summary of each dataset: Number of APKs, signatures, and certificates.

Dataset	APK	Signature	Certificate
PlayDrone	1,177,599	1,178,118	1,178,116
Alandroid	10,655	10,657	10,654
APPVN	34,415	34,433	34,485
Aptoide	138,122	138,172	138,291
Baidu	138,004	138,089	138,122
Blackmart	100,156	100,213	100,367
CafeBazaar	54,034	54,069	54,109
entumovil	235	235	235
GetJar	37,715	37,682	37,689
Mobogenie	31,546	31,570	31,590
MoboMarket	20,095	20,102	20,140
Uptodown	59,428	59,434	59,557
Yandex	22,964	22,969	22,975
zhushou360	204,416	204,567	204,673
Androzoo	3,834,514	3,830,320	3,839,527
Total	5,863,898	5,860,630	5,870,530

The total number of signatures obtained from the reacquired APK was 27,082. A total of 24,244 (89.52%) APKs still use MD5 for APK signing. From this result, it is clear that the majority of APKs using MD5 for APK signing in the PlayDrone continue to use MD5.

A similar investigation was conducted for the APK group that used the 512-bit RSA key. The APK was reacquired with the same package name, and a survey was conducted from 78 APKs obtained by reacquisition. The reacquisition of the APKs occurred on December 6, 2016. The results are given in **Table 5**. It was determined that all 78 APKs used and continue to use the 512-bit RSA key.

5.5 Apps Collected from Third-party Marketplaces

We conducted a survey using 15 datasets; a total of 5,863,898 APKs. **Table 6** gives the number of APKs in each dataset. Datasets are roughly divided into two markets: Google Play and third-party. The Androzoo dataset is a mixed dataset that includes the Google Play APKs and third-party APKs [24].

The following are the results of the analysis.

5.5.1 Certificate Signer

A total of 5,862,531 (99.85%) applications were signed using

Table 7 Total number of self-signed certificates from data that combines all datasets.

Signer type	Number	Proportion
Self-Signed	5,862,531	99.85%
Others	7,999	0.15%

Table 8 Validity period from the sum of all datasets.

Period (*n* Days)	Number	Proportion
$n < 9{,}125$	136,386	2.32%
$9{,}125 \leq n < 10{,}000$	2,284,764	38.92%
$10{,}000 \leq n$	3,449,382	58.76%

Table 9 Public key cryptography algorithm and key length used for signature from the sum of all datasets.

Algorithm and Key Length	Number	Proportion
RSA4096	13,130	0.22%
RSA2048	2,923,461	49.80%
RSA1024	2,802,787	47.74%
DSA1024	125,928	2.15%
Others	5,225	0.09%

self-signed certificates (**Table 7**). While Google Play allows self-signed certificates, this investigation shows that almost all developers are using self-signed certificates.

Certificates that are not self-signed can be divided into two major types. The first is "signed by the root CA of the developer organization". A typical case shows the organization has a root CA and a CA for android application code signing. The CA certificate for android application code signing is signed by its root CA. In these cases, root CAs are not a public CA that is trusted by the OSs or browsers.

The second is "code signing service". Most certificates of this type use the service of Symantec.

5.5.2 Period until Expiration

A validity period over 25 years for certificates used for APK signing is recommended in Google's documentation [16]. Also, it is stated that applications signed with a certificate key whose expiration date has not been reached before October 22, 2033, cannot be uploaded to the Google Play market.

Table 8 shows the result of the certificate validity period. The validity period was classfied according to the values recommended by Google. Previously, there were two recommendations for the validity period of certificates. First one is more than 10,000 days and the second one is more than 25 years (9,125 days). Current recommendation is only the second one [25].

At 2.32% of the total, 136,386 APKs had a validity period of fewer than 9,125 days. Most certificates less than 9,125 days were set with expiration dates after October 22, 2033.

The most common was certificates with a validity period of over 10,000 days, accounting for 58.76% of the total.

5.5.3 Public Key Cryptography Algorithm and Key Length

Table 9 shows the public key cryptographic algorithm used for the signature of the APK file and the key length.

The 2,048-bit RSA key was the most frequently used key length, accounting for 2,923,461 applications (49.80% of the total). The 1,024-bit RSA accounts for 2,802,787 applications (47.74% of the total). A total of 837 APKs use the 512-bit RSA key.

In DSA, the key of 1,024 bits is used most frequently, accounting for 2.15% of the whole. Incidentally, looking at DSA alone, 99.64% used keys of 1,024 bits.

Table 10 Hash function used for APK signatures.

Algorithm	Number	Proportion
SHA-256	5,619	0.10%
SHA-1	5,511,514	93.95%
MD5	349,478	5.96%

Table 11 Comparison between other datasets.

	Fahl et al. [15]	PlayDrone	Other Datasets
Self-signed Certs	99.98%	99.92%	99.85%
MD5	0.29%	4.49%	5.96%
RSA2048	51.6%	49.24%	49.80%
RSA1024	48.15%	48.13%	47.74%
RSA512	0.07%	0.02%	0.01%

5.5.4 Hash Function Used for Signature of APK

SHA-1 was the most frequently used hash function used for APK signatures, accounting for 5,511,514 applications (93.95% of the total). MD5 is used in 349,478 applications (5.96% of the total). **Table 10** shows the result of the hash function used for the APK signature.

5.5.5 Comparison between Google Play and Others

Datasets used for the investigation can be divided into two types. The PlayDrone dataset is based on the Google Play market. Alandroid, APPVN, Aptoide, Baidu, Blackmart, CafeBazzar, entumovil, GetJar, Mobogenie, MoboMarket, Uptodown, Yandex, and zhushou360 are based on other third-party markets. In this section, we compare the properties of Google Play and other markets.

Some factors almost identical for both Google Play and other markets. The certificate signer, certificate signing algorithm, and hash function used for the signature of the APK are almost identical across all markets. The proportion of each category shows similar values. For example, MD5 used for APK signature is 4.49% in Google Play and 5.96% in other markets. SHA-1 is 95,45% in Google Play and 93.95% in other markets. Thus, 99.92% of certificates in Google Play and 99.85% of certificates in other markets are self-signed.

For the signing key, DSA is used more frequently for Google Play (2.28%) than for other markets (2.15%). For RSA, 2,048-bit key is used more frequently for other markets (49.80%) than for Google Play (49.24%).

The factor that differs most is the period until expiration. In the Google Play market, the proportion of certificates having over 9,125 days until expiration is 99.55%, and 97.68% in other markets. In particular, the proportion of certificates having over 36,500 days (100 years) is 0.34% in Google play and 17.63% in other markets. APKs in Google play tend not to set longer expiration limits.

Part of these results are similar to the results of Fahl's investigation. **Table 11** shows a comparison of the results.

6. Identifying Causal Mechanisms

In this section, we study why developers used weak cryptographic algorithms when signing apps.

6.1 Default Setting of Development Environment

6.1.1 Signing Method

There are three representative ways of signing an Android application:

(1) Sign on the command line using Keytool and Jarsigner.
(2) Sign using the Eclipse ADT plug-in and ADT Export Wizard;
(3) Sign using the Generate Signed APK on Android Studio.

For commands and development tools, default key size, a signature algorithm for certificates, and a signature algorithm for applications may be different, depending on the version. MD5 or 512-bit RSA key which was found in applications in several datasets may be used, due to the difference. We investigated the differences by version and the possibility of using MD5 or the 512-bit RSA key.

6.1.2 Signing on the Command Line Using Keytool and Jarsigner

Keytool and Jarsigner are Java commands that are included in the JDK. The key length and certificate signature algorithm created by Keytool, and the signature algorithm used for JAR signatures by Jarsigner depend on the version of Java SE. **Tables 12**, **13**, and **Table 14** present the default key length for each version of Java SE 6-8 and the signature algorithm for the hash and signatures used for the signing certificate and signature algorithm of JAR.

From Table 14, we see that in Java SE 6, the signature algorithm for JAR becomes MD5. Also, from this description, the default generation of the 512-bit RSA key was not confirmed.

6.1.3 Application Development with Eclipse and Android Studio

Eclipse, an integrated development environment of Java, was the primary environment for Android application development until the end of 2015. Currently, AndroidStudio has been the official development environment since 2016. Eclipse developed Android applications using a plug-in that added Android Development Tools (ADTs) to the environment. ADT development and support was discontinued after Google's support development environment completed the transition to AndroidStudio in 2016, making it difficult to create the latest Android application in Eclipse.

Table 12 Default setting on keytool.

Version	genkeypair	genseckey	genkeypair with RSA option
JavaSE 6	DSA (1,024 bits)	DES	-
JavaSE 7	DSA (1,024 bits)	DES	RSA (2,048 bits)
JavaSE 8	DSA (1,024 bits)	DES	RSA (2,048 bits)

Table 13 Default setting on jarsigner for certificate signing.

Version	DSA	RSA	EC
JavaSE 6	SHA1 with DSA	MD5 with RSA	-
JavaSE 7	SHA1 with DSA	SHA256 with RSA	SHA256 with ECDSA
JavaSE 8	SHA1 with DSA	SHA256 with RSA	SHA256 with ECDSA

Table 14 Default setting on jarsigner for JAR file signing.

Version	DSA	RSA	EC
JavaSE 6	SHA1 with DSA	MD5 with RSA	-
JavaSE 7	SHA1 with DSA	SHA256 with RSA	SHA256 with ECDSA
JavaSE 8	SHA1 with DSA	SHA256 with RSA	SHA256 with ECDSA

To program Android applications in Java, the JDK is mandatory; therefore, both Eclipse and AndroidStudio must install it. For the signature, Eclipse uses the ADT Export Wizard from the ADT plug-in, and Android Studio uses the Generate Signed APK. Both use a graphical user interface (GUI) for signing, so signatures can be obtained without requiring commands to be typed via the command line. However, as Java and ADT commands are used behind the GUI, the signature information is considered to differ, depending on the version of the JDK and ADT.

Based on these, the default key length and signature algorithm for each version of AndroidStudio and Eclipse, and each version of the JDK and ADT, are surveyed. The environment was built in each version, and a signature by GUI was created by default. To confirm the default settings in these environments, a sample application was developed and signed using the default setting. Then, the certificate and the signature were extracted from the APK file of the sample application. The following information was extracted and compared:

- Public Key Cryptography Algorithm
- Key Length
- Signing Algorithm for APK
- Signing Algorithm for Certificates

6.1.4 Eclipse and AndroidStudio's Default Key and Signature Algorithm

The results of the automatic generation of signatures using AndroidStudio are given in **Table 15**. Besides, the automatic generation by Eclipse's ADT plug-in is presented in **Table 16**.

When checking the latest version of both environments, the key was a 2,048-bit RSA key, the signature algorithm of the APK was SHA-1, and the signature algorithm of the certificate was SHA256 with RSA. Moreover, the oldest version of AndroidStudio had the same signature information as the latest version.

In the case of Eclipse, which added ADT plug-in version 22.6.3 and was built with JDK 6.0, SHA-1 was used as the signature algorithm for APK. This result was different from Oracle's Java SE 1.6 JAR with MD5. From this, it is conceivable that the signature by Eclipse's ADT plug-in depends on ADT.

6.1.5 Short Summary of Default Configuration

It was determined that only Jarsigner on JDK version 6 uses MD5 for APK signatures, and no environment used the 512-bit RSA key as the default configuration. Other environments do not use weak algorithms.

Therefore, the default configuration of the development environments might not responsible as the primary cause of weak algorithms like in MD5 and the 51 bit RSA key.

6.2 Application-building Framework

Common factors between apps using weak algorithms are investigated for finding other causes.

First, apps using the 512-bit RSA key found in the PlayDrone are studied. Similar features were found in 211 out of 223 applications using 512-bit RSA keys. The issuer and subject information of the digital certificate used in these applications were as follows.

CN={*A Specific Service Name*} {*numbers*} OU={*A Specific Service Name*} O=Android Developers L=Anon

Table 15 Default setting of android studio.

Android Studio	JDK	PubKey Algorithm	Key Length (bits)	Signing Algorithm for APK	Signing Algorithm for Certificates
1.0 (Oldest)	8.0	RSA	2,048	SHA1	SHA256 with RSA
2.1.3 (Latest)	8.0	RSA	2,048	SHA1	SHA256 with RSA

Table 16 Default setting of Eclipse ADT plug-in.

ADT	JDK	PubKey Algorithm	Key Length (bits)	Signing Algorithm for APK	Signing Algorithm for Certificates
22.6.3.v2014	6.0	RSA	1,024	SHA1	SHA1 with RSA
23.0.7.2120684 (Latest)	8.0	RSA	2,048	SHA1	SHA256 with RSA

Table 17 Development tools considered to be used for application development.

Common Chars	Tool Name	Number	Prop.
Chars A	*Tool A*	9,515	18.00%
Chars B	*Tool B*	1,428	2.70%
Chars C	*Tool C*	346	0.65%
Chars D	*Tool D*	178	0.34%

Table 18 Development services considered to be used for application development.

Common Chars	Service Name	Number	Prop.
Chars E	*Service E*	278	0.53%
Chars F	*Service F*	232	0.44%
Chars G	*Service G*	224	0.42%

ST=Anon C=US

Although *numbers* are different for each application, all other information is identical. It also includes a specific service name. From responsible disclosure aspects, we anonymized the keyword as *A Specific Service Name*. The keyword is known as a tool that automatically stores fixed tasks of users with Android applications and settings. Therefore, these applications are expected to be automation tools based on the service. The tasks created with the service can be released as an independent application to the market.

We found the service and confirmed that the service was still working in 2016. We also had tried to make an application using the service. In that case, there was no selection of a signature key and algorithm, and the created APK had a 512-bit RSA key. Also, the issuer and subject information of the digital certificate was of the same format as above. From this result, 211 of the applications that used the 512-bit RSA key in PlayDrone were Tasker-based applications.

6.3 Online Application-building Service

The 52,866 applications that used MD5 for APK signing were also investigated. We focused on the package name for each application. Some of the packages that closely resemble the character strings included in the package were classified. There was created by tools and services that assist application developers (**Tables 17**, **18**). From responsible disclosure aspects, we anonymized the keyword. It can be seen that developers create many applications using MD5 for APK signatures with insufficient cryptographic knowledge using development tools and services.

7. Discussion

In this section, we first discuss the limitations of our work. We then present the guidelines for the stakeholders, which are derived through our findings. Finally, we share the ethical considerations in disclosing the security risks we found through this work.

7.1 Limitations

7.1.1 MultipleAPK

Apps using weak algorithms were found from PlayDrone dataset. The apps on PlayDrone were originally distributed on Google Play market.

To reduce the size of APK, Google Play allows developers to upload multiple APKs for a single application. Each APKs has information about target device specification on AndroidManifest.xml. It is called MultipleAPK. When a device accessed to an app on Google Play and the app is configured as MultipleAPK, one APK matched to the target device specification is selected and shown to download by Google Play.

If we want to analyze apps on Google Play in a rigorous manner, we have to consider the existence of bias of MultipleAPK. Gathering all APKs of MultipleAPK on Google Play market should be required. To the best of our knowledge, there is no dataset considering Multiple APK.

7.1.2 Free Applications

All of the datasets used in this investigation contain only free applications. Paid applications may have different characteristics.

7.1.3 Other enhancements on Google Play

Google Play gives several other enhancements. An application developer can use OBB (Opaque Binary Blob) as an extended area of the APK. Although an attacker can use this OBB to enable some attacks, the datasets used here does not include such data files. In addition, MultipleAPK, which enables developers to prepare multiple APKs according to the type of terminal, has not yet to be considered. Some crawlers might not be compatible with MultipleAPK.

7.1.4 Other Platforms

Code signing is now a technology that has been adopted by other platforms as well as Android. Here, we discuss the use of weak algorithms on other platforms.

In Windows, applications are usually signed. Developers need to have a certificate issued by a trusted certificate authority. MD5 is already unavailable for Windows code signing, and even SHA-1 is a transition target.

In Apple iOS, applications on the App Store which is the official marketplace for iOS applications are signed. Developers must receive a developer certificate from Apple's CA in order to register an application onto the App Store.

Compared to these two common platforms, it can be seen that Android accepts weak algorithms.

7.2 Guidelines for the Stakeholders

7.2.1 For OS Developers

Certificates with a validity period of 25 years or more using keys of 512 or 1,024 bits should be swapped instantly. Unfortunately, even though application components are the totally same, the application signed with the different certificate is considered to be a different application in the current Google Play market. In reality, even if the same developer uses two of the old key certificates and the new key certificate, on the market, the two are not considered the same developer. In other words, it is a model that does not consider certificate migration.

Otherwise, application developers need to announce, *"In this update, the terminal recognizes it as a separate application from the old application, please delete the old application"* to end users. This mechanism places the burden on end-users.

This may also become a significant risk, because the old application remains due to lack of understanding by the end-user.

7.2.2 For Market Managers

If certificate migration is not possible on the Android OS, some actions can be treated by market managers. In the current Google Play market, the trust of certificate issuer is not considered. Trust mechanism is given by the market itself. In that case, it is necessary to link old and new certificates and provide a mechanism to guarantee their identity for previously published applications.

For newly developed applications, the market should change its policy, check the key and signature algorithms, and issue a warning at uploading.

In 2017, Google launched a new service that allows developers to deposit their signing keys. Although it is useful as an approach to protect the key from leakage, threats pointed out in the paper cannot be solved. Weak algorithms continue to be accepted, and key migration remains impossible.

7.2.3 For Application Developers

Since the Android OS does not allow certificate migration even we have a key deposit service on Google, it is difficult for application developers to migrate certificates properly. As mentioned earlier, there is no other choice but to announce what is recognized as a separate application.

Instead, one possible mitigation method for developers is possible. In the follow-up investigation of applications using MD5 for signing APKs, several applications migrate from MD5 to SHA-1 for signatures to the APK. Developers can change the signature algorithm to the APK without changing certificates and keys. The Android OS and the Google Play market will continue to accept it as the same application.

Because the key of the certificate and the signature algorithm do not change, it is not a fundamental solution, but the level of protection of the APK is improved.

7.2.4 For App-Building Service Providers

It is difficult for the developer to establish a certificate concerning the purpose of the service. Therefore, it should be modified to a higher-level signature algorithm by default.

7.2.5 Others

It is relatively easy to develop the AndroidStudio plug-in that warns you if weak algorithms are set, and makes it easy to change the algorithm.

7.3 Ethical Considerations

App developers and app-building service providers know the potential risks caused using weak cryptographic algorithms for signing apps such as the 512-bit RSA key and MD5. To this end, we contacted national CERT to inform stakeholders about the potential risks associated with the use of weak cryptographic algorithms. The answer from national CERT was, "Please contact the concerned organizations. It would be good idea to contact Google at the same time."

8. Conclusion

In this work, we studied the use of signatures in Android applications using the vast collection of the official Android marketplace — PlayDrone dataset. We revealed that the use of weak cryptographic algorithms such as the 512-bit RSA key and MD5 was non-negligible; in total, 223 applications used the 512-bit RSA key and 52,866 applications used MD5 for APK signing. The finding implies that these apps are exposed to security threats, such as malicious updates and unintentional information leakage. As 99.55% of certificates associated with these apps had an expiration date of 25 years or more, the attackers may have sufficient time to accomplish the attack. While the PlayDrone dataset was collected in 2014, our follow-up study revealed that the use of 512-bit RSA key and MD5 were still there in 2016. We found that many users have installed the corresponding applications. We then explored the origins of the use of weak algorithms and identified the reasons, i.e., specific applications and online app-building services.

To address the threats caused by the inclusion of weak cryptographic algorithms, we advocate that mobile operating systems and mobile app distribution platform needs to have a mechanism that enables app developers to migrate the certificates of apps after the publication of apps. However, for that purpose, it is necessary to change the entire Android ecosystem, including the OS and markets.

References

[1] StatCounter: *Mobile and tablet internet usage exceeds desktop for first time worldwide*, available from ⟨http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide⟩ (accessed 2018-11-17).

[2] Durumeric, Z., Wustrow, E. and Halderman, J.A.: ZMap: Fast Internet-wide Scanning and Its Security Applications, *22nd USENIX Security Symposium* (*USENIX Security 13*), pp.605–620, USENIX (2013) (online), available from ⟨https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/durumeric⟩.

[3] Martin, W., Sarro, F., Jia, Y., Zhang, Y. and Harman, M.: A Survey of App Store Analysis for Software Engineering, *IEEE Trans. Software Engineering*, Vol.43, No.9, pp.817–847 (online), DOI: 10.1109/TSE.2016.2630689 (2017).

[4] Durumeric, Z., Kasten, J., Bailey, M. and Halderman, J.A.: Analysis of the HTTPS Certificate Ecosystem, *Proc. 2013 Conference on Internet Measurement Conference*, *IMC '13*, pp.291–304, ACM (online), DOI: 10.1145/2504730.2504755 (2013).

[5] Egele, M., Brumley, D., Fratantonio, Y. and Kruegel, C.: An Empirical Study of Cryptographic Misuse in Android Applications, *Proc. 2013 ACM SIGSAC Conference on Computer and Com-*

munications Security, *CCS '13*, pp.73–84, ACM (online), DOI: 10.1145/2508859.2516693 (2013).

[6] Shuai, S., Guowei, D., Tao, G., Tianchang, Y. and Chenjie, S.: Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications, *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pp.75–80 (online), DOI: 10.1109/DASC.2014.22 (2014).

[7] Kim, S.H., Han, D. and Lee, D.H.: Predictability of Android OpenSSL's Pseudo Random Number Generator, *Proc. 2013 ACM SIGSAC Conference on Computer and Communications Security*, *CCS '13*, pp.659–668, ACM (online), DOI: 10.1145/2508859.2516706 (2013).

[8] Fahl, S., Harbach, M., Perl, H., Koetter, M. and Smith, M.: Rethinking SSL Development in an Appified World, *Proc. 2013 ACM SIGSAC Conference on Computer and Communications Security*, *CCS '13*, pp.49–60, ACM (online), DOI: 10.1145/2508859.2516655 (2013).

[9] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B. and Smith, M.: Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security, *Proc. 2012 ACM Conference on Computer and Communications Security*, *CCS '12*, pp.50–61, ACM (online), DOI: 10.1145/2382196.2382205 (2012).

[10] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D. and Shmatikov, V.: The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software, *Proc. 2012 ACM Conference on Computer and Communications Security*, *CCS '12*, pp.38–49, ACM (online), DOI: 10.1145/2382196.2382204 (2012).

[11] Onwuzurike, L. and De Cristofaro, E.: Danger is My Middle Name: Experimenting with SSL Vulnerabilities in Android Apps, *Proc. 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, *WiSec '15*, pp.15:1–15:6, ACM (online), DOI: 10.1145/2766498.2766522 (2015).

[12] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C. and Vigna, G.: Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications, *NDSS*, Vol.14, pp.23–26 (2014).

[13] Reaves, B., Scaife, N., Bates, A., Traynor, P. and Butler, K.R.: Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World, *24th USENIX Security Symposium* (*USENIX Security 15*), pp.17–32, USENIX Association (2015) (online), available from ⟨https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/reaves⟩.

[14] Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z. and Khan, L.: Smv-hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in android apps, *Proc. 21st Annual Network and Distributed System Security Symposium*, *NDSS '14*, Citeseer (2014).

[15] Fahl, S., Dechand, S., Perl, H., Fischer, F., Smrcek, J. and Smith, M.: Hey, NSA: Stay Away from My Market! Future Proofing App Markets Against Powerful Attackers, *Proc. 2014 ACM SIGSAC Conference on Computer and Communications Security*, *CCS '14*, pp.1143–1155, ACM (online), DOI: 10.1145/2660267.2660311 (2014).

[16] Sign your app — Android Developers, available from ⟨https://developer.android.com/studio/publish/app-signing?hl=en⟩ (accessed 2019-04-02).

[17] Valenta, L., Cohney, S., Liao, A., Fried, J., Bodduluri, S. and Heninger, N.: Factoring as a service, *International Conference on Financial Cryptography and Data Security*, pp.321–338, Springer (2016).

[18] Barker, E.: NIST Special Publication 800–57 Part 1, Revision 4 (2016), available from ⟨https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-4/final⟩.

[19] Wang, X. and Yu, H.: How to Break MD5 and Other Hash Functions, *Advances in Cryptology – EUROCRYPT 2005*, pp.19–35, Springer Berlin Heidelberg (2005).

[20] Stevens, M., Lenstra, A. and de Weger, B.: Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities, *Advances in Cryptology - EUROCRYPT 2007*, pp.1–22, Springer Berlin Heidelberg (2007).

[21] Sotirov, A., Stevens, M., Appelbaum, J., Lenstra, A.K., Molnar, D., Osvik, D.A. and de Weger, B.: MD5 considered harmful today, creating a rogue CA certificate, *25th Annual Chaos Communication Congress*, No.EPFL-CONF-164547 (2008).

[22] Stevens, M., Bursztein, E., Karpman, P., Albertini, A. and Markov, Y.: The First Collision for Full SHA-1, *Advances in Cryptology – CRYPTO 2017*, pp.570–596, Springer International Publishing (2017).

[23] Viennot, N., Garcia, E. and Nieh, J.: A Measurement Study of Google Play, *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, *SIGMETRICS '14*, pp.221–233, ACM (online), DOI: 10.1145/2591971.2592003 (2014).

[24] Allix, K., Bissyandé, T.F., Klein, J. and Traon, Y.L.: AndroZoo: Collecting Millions of Android Apps for the Research Community, *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories* (*MSR*), pp.468–471 (online), DOI: 10.1109/MSR.2016.056 (2016).

[25] Signing Your Applications — Android Developers, available from ⟨https://web.archive.org/web/20120613232550/https://developer.android.com/guide/publishing/app-signing.html⟩ (accessed 2019-04-02).

Kanae Yoshida received her B.E. degree from Toho University in 2017. She joined FDC Inc. in 2017. She received the Best Student Paper Award at the Computer Security Symposium 2016 (CSS2016).

Hironori Imai received his B.E. degree from Toho University in 2018. He is currently a master course student at Graduate School of Science, Toho University. He received the Paper Award and the Presentation Award at the Multimedia, Distributed, Cooperative, and Mobile Symposium (DICOMO2018).

Nana Serizawa received B.E. and M.E. degrees in computer science from Waseda University in 2017 and 2019, respectively. Her research interest is psychological security.

Tatsuya Mori is currently a professor at Waseda University, Tokyo, Japan. He received B.E. and M.E. degrees in applied physics, and Ph.D. degree in information science from Waseda University, in 1997, 1999 and 2005, respectively. He joined NTT lab in 1999. Since then, he has been engaged in the research of measurement and analysis of networks and cyber security. From Mar. 2007 to Mar 2008, he was a visiting researcher at the University of Wisconsin-Madison. He received Telecom System Technology Award from TAF in 2010 and Best Paper Awards from IEICE and IEEE/ACM COMSNETS in 2009 and 2010, respectively. Dr. Mori is a member of ACM, IEEE, IEICE, IPSJ, and USENIX.

Akira Kanaoka received his Ph.D. degree in engineering from University of Tsukuba, Japan in 2004. He worked at SECOM Co., Ltd. from 2004 to 2007, and at University of Tsukuba from 2007 to 2013. He is currently an associate professor of Department of Information Science, Faculty of Science, Toho University. His research interests include usable security and privacy.