Characterizing Android App Signing Issues

Haoyu Wang[1], Hongxuan Liu[2], Xusheng Xiao[3], Guozhu Meng[4,5], and Yao Guo[2]
[1] Beijing University of Posts and Telecommunications, Beijing, China
[2] Key Lab of High-Confidence Software Technologies (MOE), Dept of Computer Science, Peking University, China
[3] Case Western Reserve University [4] SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
[5] School of Cyber Security, University of Chinese Academy of Sciences, China

***Abstract*—In the app releasing process, Android requires all apps to be digitally signed with a certificate before distribution. Android uses this certificate to identify the author and ensure the integrity of an app. However, a number of signature issues have been reported recently, threatening the security and privacy of Android apps. In this paper, we present the first large-scale systematic measurement study on issues related to Android app signatures. We first create a taxonomy covering four types of app signing issues (21 anti-patterns in total), including vulnerabilities, potential attacks, release bugs and compatibility issues. Then we developed an automated tool to characterize signature-related issues in over 5 million app items (3 million distinct apks) crawled from Google Play and 24 alternative Android app markets. Our empirical findings suggest that although Google has introduced apk-level signing schemes (V2 and V3) to overcome some of the known security issues, more than 93% of the apps still use only the JAR signing scheme (V1), which poses great security threats. Besides, we also revealed that 7% to 45% of the apps in the 25 studied markets have been found containing at least one signing issue, while a large number of apps have been exposed to security vulnerabilities, attacks and compatibility issues. Among them a considerable number of apps we identified are popular apps with millions of downloads. Finally, our evolution analysis suggested that most of the issues were not mitigated after a considerable amount of time across markets. The results shed light on the emergency for detecting and repairing the app signing issues.**

***Index Terms*—Signature, Vulnerability, Mobile App, Certificate**

I. Introduction

Mobile apps are distributed through app markets such as Google Play, where users can search and download desired apps. In the app releasing process, Android requires all apps cryptographically signed by developers, which is known as package signatures [1]. *App signing* is the primary security mechanism that protects the integrity of an app after it is released by the developer, for example ensuring that only the original developer can issue an update to an already installed app. The Android system uses this certificate to identify the author of an app. The certificate does not need to be signed by a certificating authority.

However, in recent years, a number of vulnerabilities related to app signing have been disclosed from time to time, posing great security risks to a significant number of Android apps and mobile devices. For example, the Janus vulnerability (CVE-2017-13156) [2] allows attackers to modify APKs without breaking their original signatures, which could affect almost all the apps signed with Android's original JAR-based signing scheme (V1 Signing Scheme) in mobile devices running Android systems between v5.0 and v8.0 [3]. The *Master Key vulnerability* (CVE-2013-4787) [4] was disclosed in 2013. It was reported that 99% of Android devices (by the date of July 2013) were affected by this vulnerability, which could also allow attackers to modify any legitimate signed apps without breaking their original signatures. Similarly, two other Android Master Key vulnerabilities [5], [6] were also discovered in Android 4.3, as malware were found using these vulnerabilities to inject malicious payload to legitimate apps.

In addition, some attackers are selling legitimate Android code-signing certificates to evade malware detection [7]. As many anti-virus engines use white-lists to filter apps created by legitimated developers, it is easy for malware to sneak into a mobile device if the malware is signed with the purchased certificates. Moreover, many amateur app developers (even the ones who created popular apps) use the private keys well-known in the community (e.g., publicly-known private keys included within the Android Open Source Project) to sign their apps, which makes it easy for attackers to replace the vulnerable apps with malicious ones without users' knowledge.

To address these issues, the signing schemes in Android have been evolving as well. On one hand, a number of bugs and vulnerabilities related to app signing are disclosed and then fixed during the evolution. On the other hand, due to numerous vulnerabilities found in the original V1 Signing Scheme [8], which has been adopted since the first version of Android, Android has introduced new signing schemes in its later versions. For example, Android Nougat (v7.0) introduced the APK Signing Scheme (i.e., V2 Signing Scheme) [9] to provide APK-level signing. An improved version of the APK Signing Scheme (i.e., V3 Signing Scheme) [10] was introduced in Android Pie (v9.0).

However, the app signing issues have not been systematically studied, especially when considering that there are a variety of severe signing issues, as well as millions of apps and developers in the ecosystem. Although Google has introduced new signing schemes to enhance security, it is still unclear how many apps have been suffering from known signing issues in the wild. Besides, as a large portion of Android devices are running legacy Android system versions [11], little is known on how many attackers have exploited the existing vulnerabilities to perform possible attacks in the wild.

Contributions. In this paper, we perform the first *large-scale* and *systematic* study of Android app signing issues. We first compile a taxonomy of 21 anti-patterns of app signing

(cf. **Section III**), including 2 app-level vulnerabilities, 6 types of possible attacks (5 of them are performed by exploiting system-level vulnerabilities), one compatibility issue, and 12 types of releasing bugs. Based on these anti-patterns, we developed a tool to automatically detect each type of the issues (cf. **Section IV**). To measure the presence of signature-related issues, we crawled 5.03 million app items from 25 app markets including Google Play (over 2.95 million distinct APKs in total due to the overlapping among markets) and applied our tool to these apps to detect app signing issues (cf. **Section V**). We studied the results to analyze the distribution of apps with signing issues from various aspects including app markets, app categories, app popularity and release/update time. At last, we studied the evolution of app signing issues (cf. **Section VI**), and performed a post analysis seven months later to measure how many apps with signing issues have been removed or mitigated. Among many interesting results and observations, the following are the most prominent:

- **93.7% of the apps (roughly 2.7 million) studied in this paper could be exploited on devices with Android versions prior to 7.0**, as they only adopted the V1 signing scheme, even though the V2 scheme had been introduced for over 1.5 years by the date of our study.
- **App signing issues are prevalent in both Google Play and alternative markets.** Roughly 7% to 45% of the apps in the 25 studied markets have been found containing at least one issues, which allow attackers to inject malicious payloads via bypassing verification and replacing unprotected files with malicious payloads in the signed APKs. Such issues can even be found in many popular apps with millions of downloads.
- **A significant number of apps (over 65K) are found to be signed with publicly-known keys, which allow attackers to arbitrarily modify the apps without breaking its original signatures**, indicating that most of the developers paid little attention to app signing issues, or simply were unaware of the potential risks. **These apps have aggregated over 5.7 billion installs in total.** Even some popular apps use publicly-known private keys, e.g., com.shuqi.controller, with over 100 million downloads, was found using the "testkey" to sign itself.
- **94 apps (435K installs in total) were found exploiting the Master Key vulnerability to perform attacks**, and most of them were confirmed as malware by VirusTotal. **Over 1K apps were found being compromised, with over 7.1 billion app downloads in total.** Attackers try to remove ad libraries or resource files to create ad-free apps or compromise the functionalities of the apps.
- **Over 90K apps were found containing release bugs or compatibility issues that may lead to installation failures** on certain Android versions, including some apps with billions of downloads (e.g., com.kugou.android).
- **Most of the apps with signing issues have been released years ago** (e.g., more than 50% of the apps that exploit vulnerabilities were released before 2016), which suggested that they may have impacted millions of users for years. Besides, our post analysis suggested that **most markets had not removed/updated the apps with signing issues after 7 months since our initial study**. Such findings indicate that many app markets paid little attention to (or were unaware of) the security issues caused by app signing, which could make these markets an easy target to disseminate malware.

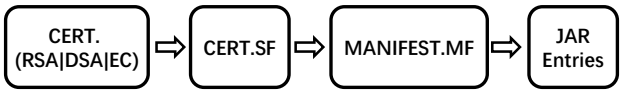

Fig. 1. The protection chain of the V1 signing scheme.

To the best of our knowledge, this is the first systematic study of app signing issues at scale, longitudinally and across various dimensions. Our results motivate the need for more research efforts to disclose the widely unexplored app signing issues and further improve the app ecosystem.

II. Background

A. App Signing Keys

Android enforces a self-signed mechanism – an app should be signed with its *developer's certificate* before it is installed, so as to prevent the apps from being tampered. The developer holds the *private key* (with the extension .pk8) of the certificate, and uses it to sign the APK. The private key must be kept secret and protected by a password. The *public key* is used to verify its signature, which is visible to everyone.

B. App Signing Schemes in Android

There are three signing schemes used in Android [1].

- *JAR signing scheme (V1)* : The V1 scheme, introduced since Android 1.0, is based on JAR signing [8]. It has been introduced since the first version of Android.
- *APK signing scheme (V2)*: The V2 scheme (APK-level signing) [9] was introduced in Android 7.0. The contents of an APK file are hashed and signed, and then the resulting signing block is inserted into the APK.
- *APK signing scheme (V3)*: This is an improved version [10] of V2, introduced in Android 9.0. It contains additional information in the signing block.

For compatibility and security concerns, it is recommended by Google to sign apps with all the three schemes, first with V1, then V2, and finally V3. Devices running out-of-date Android systems typically ignore the V2 and the V3 signatures, thus V1 signatures should always be included.

*1) **JAR Signing Scheme (V1):*** A JAR-signed APK must contain the exact files listed in META-INF/MANIFEST.MF and all the files must be signed by the same set of certificates. All signature-related files are stored in the META-INFO/ directory, including *MANIFEST.MF*, *CERT.SF*, and *CERT.(RSA|DSA|EC)* (Note that the *.SF

and *.(RSA|DSA|EC) can be any signer-customized strings). These files form the protection chain, as shown in Figure 1.

MANIFEST.MF contains the hash results of all source files in the APK file to prevent them from being tampered.

CERT.SF contains a file-level hash value of the MANIFEST.MF and hash values of each section of MANIFEST.MF. In the Android system, the framework first verifies the file-level hash value of the MANIFEST.MF. If that fails, the hash value of each MANIFEST.MF section is verified instead.

CERT.(RSA|DSA|EC) Android supports three signature algorithms: RSA, DSA, and ECDSA (introduced in Android 4.3). CERT.(RSA|DSA|EC) is used to verify the signature of "CERT.SF". It includes the certificate meta info (subject, issuer, series number, etc.), the signature of "CERT.SF" signed by developers' private keys, and the public key.

*2) **APK Signing (V2 & V3 Schemes)**:* V1 signatures does not protect some parts of the APK, such as ZIP metadata and the files located in the META-INF directory. Only uncompressed file contents are verified in V1 (not the whole APK), which allows modifications to be made to the APK file after signing (e.g., Janus and Master Key vulnerabilities).

To overcome the limitation of V1, the V2 and V3 schemes consider all the binary contents of the whole APK file. V2 and V3 signing insert a *Signing Block* into the APK file immediately before the ZIP Central Directory section, which is located at the end of the file. Any modifications to the APK, including ZIP metadata modifications, will invalidate the APK signature. The new formats are backwards compatible, so APKs signed with the new signature schemes can be installed on legacy Android devices (which simply ignore the extra data added to the APK), as long as these APKs are also V1-signed.

III. A Taxonomy of App Signing Issues

To the best of our knowledge, no previous work has compiled a taxonomy of app signing issues because the relevant issues have not been studied systematically. In order to provide an extensive taxonomy covering most of the signing issues, we have investigated app signing issues in the following means. First, we resort to Common Vulnerabilities and Exposures (CVE) and Android Vulnerability [12] for searching related vulnerabilities using keywords such as "Android" and "signature". We have identified 5 vulnerabilities (CVE-2013-4787, ANDROID-9695860, ANDROID-9950697, FakeID, CVE-2017-13156) related to the Android app signing process. All of them are system-level vulnerabilities that could be exploited by attackers. Second, we manually inspect the verification process in the Android framework (mainly for checking the release bugs and the compatibility issues related to app signing), search signature-related questions (using keywords including APK, signing, signature, etc.) from StackOverflow, and summarize the issues found in technical reports. As a result, we compiled a taxonomy of 21 anti-patterns of app signing, as shown in Table I. We have classified them into 4 categories based on the severity levels and impacts:

- **Vulnerabilities.** Apps have known signing vulnerabilities, or they could be potentially exploited by attackers.
- **Exploits.** Apps are tampered or exploited by attackers using known vulnerabilities (both system-level and app-level). Note that all the CVEs we summarized are system-level vulnerabilities that could be exploited by attackers.
- **Compatibility issues.** This type is usually introduced by using unsupported digest/signature algorithms, which could lead to installation failures in certain Android versions (based on its supported minimum SDK versions).
- **Release bugs.** This type of issues are generally caused by the developers in the apps' release process (e.g., use the packing tools or releasing tools improperly), which could lead to app installation failures in most cases.

A. App Vulnerabilities

We have found two types of app-level signature vulnerabilities, including (1) signing apps with a publicly-known private key and (2) unprotected contents in the META-INF folder.

*1) **Vul-1 - Signing Apps with publicly-known Private Keys**:* In general, private keys should be kept secret in order to prevent unauthorized modifications to the original app. However, many privacy keys are well known in the Android development community. The most famous set of keys are the publicly-known private keys included in the AOSP project. The standard Android build uses four known keys, all of which can be found at `build/target/product/security`. For example, *TestKey* is the generic default key for packages that do not otherwise specify a key. Other publicly-known keys include *Platform (key)*, *Shared (key)* and *Media (key)*.

For apps signed with the publicly-known keys, it is easy for attackers to replace this vulnerable app with another one (possible with malicious payloads), without user's knowledge.

*2) **Vul-2 - Unprotected Contents in the META-INF**:* The V1 scheme verifies the integrity of all files in the APK except those inside the META-INFO directory, which could introduce security issues. On one hand, malicious payloads can be hidden in this directory, and dynamically loaded at runtime (e.g., an app may implement the logic to iterate the META-INFO directory). On the other hand, for the legitimate apps that put unprotected contents in the directory, attackers could easily modify the APK through simply replacing the files inside META-INFO with malicious payloads. Note that the security risks caused by this vulnerability usually depend on the type and the content of the unprotected files. For example, if developers put unprotected libraries under this directory, it is easy for attackers to replace them with malicious ones.

B. Security Exploits

*1) **Attack-1: Exploiting Master Key Vulnerability**:* MANIFEST.MF contains a digitally signed list of checksums for the rest of the archive. Before app installation, the files in the APK are extracted and their digests are compared with the corresponding checksums in this list. If there is a mismatch, the verification will fail and the installation will be rejected. However, if the developer puts two files of the same name into the APK, the verifying process will verify the first file, but install and use the second file [4], which is

TABLE I
A TAXONOMY OF 21 ANTI-PATTERNS RELATED TO APP SIGNING.

Issue Type	Issue	V1	V2	System Versions	Impact
Vulnerabilities	Signing apps with publicly known private keys	Y	Y	All Version	Modify app without breaking its signature
	Unprotected Contents in the META-INF	Y	-	Before v7.0	Replace the unprotected files
Exploits	Exploiting Master Key Vulnerability	Y	-	Before V4.3	Modify app without breaking its signature
	Compromise the Integrity of APK	Y	-	Before v6.0	Remove files without breaking its signature
	Exploiting Janus Vulnerability	Y	-	Before v7.0	Modify app without breaking its signature
	Exploiting Unsigned Shorts Vulnerability	Y	-	Before v4.3	Modify app without breaking its signature
	Exploiting Unchecked Name Vulnerability	Y	-	Before v4.4	Modify app without breaking its signature
	Exploiting the Fake ID Vulnerability	Y	-	Before v4.4	Modify app without breaking its signature
Release Bugs	Mismatch between signature and *.SF	Y	-	All version	Installation Failure
	Mismatch between *.SF and *.MF	Y	-	All versions	Installation Failure
	Incomplete *.SF	Y	-	All versions	Installation Failure
	Incomplete *.MF	Y	-	All versions	Installation Failure
	Without *.MF	Y	-	All versions	Installation Failure
	Mismatch between *.MF and JAR Entry	Y	-	All versions	Installation Failure
	Cannot find any signature	Y	Y	All versions	Installation Failure
	Signed by different signature groups	Y	-	All versions	Installation Failure
	Rollback protection issue	Y	-	After v7.0	Installation Failure
	V2-related bug	-	Y	After v7.0	Installation Failure
	Extra byte at the end of Zip file	Y	Y	All versions	Installation Failure
	Cannot extract files from Zip	Y	Y	All versions	Installation Failure
Compatibility	Unsupported digest algorithm	Y	-	Specific Version	Installation Failure

the underlying reason leading to the master key vulnerability. This vulnerability allows attackers to insert malicious payloads in the package. The attacker can exploit the original apps by adding an additional malicious classes.dex file and also an additional Android manifest file. Such exploits were found in many real attack cases [13]. It was patched by Google in Jelly Bean, and affects Android systems between 1.6 and 4.2.

*2) **Attack-2: Compromise the Integrity of APK Files** :* In general, all the files should be protected by MANIFEST.MF to prevent them from being tampered with. If there are some missing files in MANIFEST.MF, it is possible that (1) the APK has been modified by the attackers, as the attackers could remove files from the zip file without breaking the signature protected by the JAR signing scheme, or (2) it incurs certain bugs during the APK packing process. Thus, we categorize this type of issues into the attack category since it indicates the integrity of the APK has been compromised. Android has fixed this attack surface by improving `StrictJarFile` to better handle the issue of missing files and ensure that all manifest files are present in the jar since Android v6.0. Thus, this attack could only target Android OS versions prior to 6.0.

*3) **Attack-3: Exploiting Janus Vulnerability**:* APK files could contain arbitrary bytes (also called padding) at the start, before its zip entries. The V1 scheme only takes into account the zip entries, and ignores any extra bytes when computing or verifying the app's signature. Thus, a file can be both a valid APK file and a valid dex file. As a result, attackers can pretend a malicious dex file as an APK file, without affecting its signature. The Android runtime then accepts the APK as a valid update of an earlier legitimate version of the app.

The Janus vulnerability affects V1-signed apps running on Android OS 5.0 to 8.0 [3]. Apps signed with the V2 and V3 schemes and running on the devices supporting the latest signature schemes are protected against this vulnerability.

*4) **Attack-4: Exploiting Unsigned Shorts Vulnerability**:* Discovered in 2013, it is also known as the "second Masterkey" vulnerability [14]. The underlying reason is that the file offsets in zips are supposed to be unsigned but are interpreted as signed, causing that *the contents to be verified* differ from *the content to be executed*. Several different techniques [5] can exploit this vulnerability. It is much more powerful than the "MasterKey" vulnerability, as "MasterKey" only allows attackers to replace the file contents present in the original signed zip, while this vulnerability could allow attackers to insert arbitrary new files that did not exist in the original zip previously. It was patched in Android Jelly Bean, and thus affects Android prior to 4.3.

*5) **Attack-5: Exploiting Unchecked Name Vulnerability**:* The signature verification process in Android prior to v4.4 does not check the lengths of file names correctly [15]. It assumes that the lengths of the file names are the same in both the local file header section as well as the central directory section of the Zip file header. To exploit this vulnerability, the attackers woould first generate a difference between how the zip files are verified compared with how they are extracted, so that it allows files in an existing APK to be replaced with new ones. For example, one could set the length of the file name in the local file header section to a size large enough to skip the length of the real name (which was defined in the central directory) and the data that will be used, and then insert the malicious code after the data that will be verified.

*6) **Attack-6: Exploiting the Fake ID Vulnerability**:* The underlying reason of this vulnerability is that the Android package installer (e.g., createChain() and findCert() methods of the Android JarUtils class) does not properly validate an app's certificate chain [16]. A malicious app can claim to be issued by another identity and impersonates a privileged app to gain access to vendor-specific privilege resources. This attack could affect Android OS versions between 2.1 and 4.4.

C. Release Bugs

We have summarized 12 types of signing-related release bugs, which are classified into three categories.

*1) **V1-related Bugs**:* The verification process of the V1 scheme follows a protection chain (cf. Fig. 1). MANIFEST.MF contains the message digests of all source files in the APK to prevent their integrity from tampering. There must be at least one SF file (e.g., CERT.SF) that stores the base64-encoded codes of the message digest of MANIFEST.MF, and the message digests of all the digests stored in MANIFEST.MF. For each SF file, there must be a corresponding signature file (e.g., CERT.RSA or CERT.DSA or CERT.EC) that stores the digital signature of the SF file, and its signing certificate. Thus, any inconsistency in the protection chain will lead to bugs, which will make the app installation process fail. In general, these bugs are introduced when developers pack the apps, which should be avoided, since apps with these bugs could not be installed successfully. By analyzing the protection chain, we have identified 9 types of V1-related release bugs: (1) *Verification failure between the signature file and the SF file*, (2) *Verification failure between the SF file and MANIFEST.MF*, (3) *Incomplete or missing SF files*, (4) *Incomplete MANIFEST.MF*, (5) *Missing MANIFEST.MF*, (6) *Verification failure between MANIFEST.MF and JAR entry*, (7) *No signature files*, (8) *The files signed by different signature groups*, and (9) *Rollback protection issue*.

For rollback protection, Android requires that the V2-signed APKs that are also V1-signed must contain an `X-Android-APK-Signed` attribute in the main section of their SF files. When verifying the V1 signature, the APK verifier is required to reject APK files that do not have a signature for the APK signature scheme (e.g., V2 scheme).

Apps with rollback protection issues cannot be installed on Android 7.0 and newer versions. Apps with other V1-related issues cannot be installed on all Android versions.

*2) **V2-related Bugs**:* For APK-level protection, a V2-signed APK consists of four sections, including (1) Contents of ZIP entries, (2) APK Signing Block, (3) ZIP Central Directory and (4) ZIP End of Central Directory. These sections form a protect chain [9]. Any verification failures during the process would lead to V2-related bugs. Apps with such bugs cannot be installed on Android 7.0 and up.

*3) **Zip-related Bugs**:* We have identified two types of zip-related bugs. The first is "*extra bytes at the end of Zip file*". As the legacy Android systems enforce loose zip verification, apps with this type of bug could be installed on Android systems with versions prior to v5.0. Android introduces "libziparchive" to verify zip files since v5.0, which is stricter than the original *libdvm* library used for extracting zip files. The second type is "*failed to extract certain files from the zip file*", which occurs during the extraction process of the zip files. Apps with this bug cannot be installed on any Android devices.

D. Compatibility Issues

The supported digest/signature algorithms are updated when Android OS evolves. For example, SHA256withRSA is available for minimum SDK 18 (Android v4.3) and up, while SHA256withDSA could only be used for minimum SDK 21 (Android v5.0) and up, and SHA256withEC for minimum SDK 18 and up. As a result, JAR signatures containing unsupported digest algorithms will lead to compatibility issues. In this paper, we resort to APKSigner to get the JAR signing digest algorithm used for signing an APK using the provided key, and compare with the *minSdkVersion* (minimum API Level) of the platform on which the APK may be installed.

IV. Study Methodology

We present the details of our characterization study on app signatures in this section. We first describe the dataset used for our study. Then, we present the study design and research questions we focus in this paper. At last, we present the tool we developed for conducting the study on the dataset.

A. Dataset

To measure the presence of signature-related issues, we first make efforts to implement different crawlers to harvest mobile apps from 25 popular Android app stores, as listed in Table II. Note that besides Google Play, we also crawled 24 popular alternative markets in order to understand the distribution of APK signing issues globally. As Google Play is restricted in some countries (e.g., China), Android users have to resort to various alternative app markets.

As shown in Table II, we have crawled 5.03 million app items (with all the metadata and APKs) during December 2017 and January 2018. Over 200K of them are crawled from Google Play, while the remaining 4.8 million apps are crawled from major Chinese app markets. Since developers could release the same apps (with the same package name and hashing value) to multiple markets, our dataset contains 2,951,017 distinct APKs (with different hashing value) in total. We believe our dataset is large enough to study the presence of app signing issues and perform comparative studies across Google Play and alternative app markets.

Post Analysis. After 7 months (August 2018), we launched a second, two-day crawling campaign for analyzing whether any of the studied apps with signing issues have been removed or updated from each individual market (cf. Section VI).

B. Study Design

In this paper, we focus on the following research questions:

- **RQ1: What is the distribution of V1 and V2 signatures in the wild[1]?** By the date of our first crawling, the APK signing scheme (V2) had been released for 1.5 years. As the V1 signing scheme is known to be less secure, it is interesting to study how the developers have adopted the more secure V2 signing scheme.
- **RQ2: How many apps are exposed to the security risks introduced by APK signing issues?** Are there any correlation between app popularity and app categories? Are there any differences across app markets?

[1]Note that the V3 signing scheme was introduced with Android Pie (v9.0) in August 2018, thus our dataset contains no V3-signed apps.

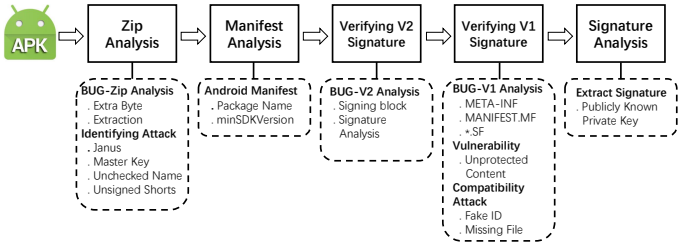

Fig. 2. The workflow of our app signing analysis tool.

- **RQ3: The evolution of app signing issues**. How long have these vulnerable apps been released to the markets? Have the apps with signing issues been removed or updated during the evolution?

C. Methodology

We have developed an automated tool to check the signature-related issues in Android apps. As shown in Figure 2, our tool is comprised of the following five main parts.

1) Zip Analysis: The main purpose of Zip analysis is to check the integrity of a Zip file and extract all the necessary files. First, we search the End of Central Directory (EOCD) record from the tail of Zip, in order to verify whether it is a valid Zip file. Based on the position of EOCD, we could analyze whether there are extra bytes at the end of Zip file (cf. **Zip-bug-1**). Then, we parse each item in the Central Directory, and locate the corresponding local file header (LFH). We then check the consistence between the metadata of the files in CD an LFH to verify whether the Zip file has exploited the "unchecked name length" vulnerability. (cf. **Attack-5**) We further analyze the File Offset of CD and LFH to see whether they are exploited to perform attacks. (cf. **Attack-4**) After this step, we extract files from Zip based on the information listed in CD and LFH, and then verify them. (cf. **Zip-bug-2**) To identify attacks that exploit the Janus vulnerability, we check the Zip heading to see whether there exist dex file paddings (cf. **Attack-3**). To identify the attacks that exploit Master Key vulnerability, we iterate all the files and verify whether there are duplicate file names (cf. **Attack-1**). Note that as the attacks could only be triggered at specific system versions (cf. Table I), we also combine the following manifest analysis results (e.g., minimum API level) to measure whether such attacks could be successfully performed.

2) Manifest Analysis: AndroidManifest can be extracted during Zip analysis. By analyzing AndroidManifest, we could get the basic information of the app, including its package name and the minimum supported API level. The information of the minimum Android API level could offer insights about whether app developers are trying to target top-end users. Note that the detection of compatibility issues will use the information of the minimum Android API level (cf. **Compatibility**).

3) Verifying V2 Signatures: We first identify the APK Signing Block, parse it and extract the APK Signature Scheme v2 block (ASSB) from it. Then we extract all the Signers from ASSB. For each Signer, we parse it and get all the signed data, the list of signatures and the corresponding public key. For each signature, we analyze its digest algorithm and pick the signature with the strongest algorithm (out of SHA-512, SHA-384, SHA-256, SHA-1). Then we use the Public Key and the selected Signature to verity Signed Data, and further extract Certificates and Digests from the Signed Data, and verify the integrity of the APK file (cf. **V2-bug**).

4) Verifying V1 Signatures: We first identify and parse META-INF and MANIFEST.MF (cf. **Vul-2**). We check if each of the hashing value listed in MANIFEST.MF is consistent with the corresponding file (cf. **V1-bug-5, V1-bug-6, ATTACK-2**), and make sure that each item listed in MANIFEST.MF exist in the APK (cf. **V1-bug-4**). For each signature file (i.e., RSA/DSA/EC), we first identify the corresponding SF file (cf. **V1-bug-1, V1-bug-3**). To identify attacks that exploit the Fake ID vulnerability, we use the method [17] patched by the Android framework to check the certificate chain signatures (cf. **ATTACK-6**). For each *SignerInfo* in the signature, we check the corresponding SF file and verify whether the digest algorithm is supported in the corresponding API level (cf. **COMPATIBILITY**). If the SF file is protected by V2 but we cannot find the V2 signature, we will report it as a releasing bug (cf. **V1-bug-9**). We use the SF file to verify MANIFEST.MF and analyze if MANIFEST.MF is modified (cf. **V1-bug-2**). Then we verify whether each file is signed by the same signature group (cf. **V1-bug-8**).

5) Signature Analysis: At last, we extract all the signature files (both V1 and V2), compare them with publicly known signatures (cf. **Vul-1**), and further analyze the consistency between signatures (e.g., apps signed by multiple signatures).

Implementation. The implementation of verifying V2/V1 signatures of the tool is based on APKSigner [18], a widely used tool to sign APKs and to confirm that an APK's signature will be verified successfully. As APKSigner is only able to identify the bugs that lead to unsuccessful app installation, we have implemented our own code to identify the vulnerabilities and attacks as mentioned above.

V. Results and Analysis

In this section, we first provide some general statistics to understand the distribution of V1 and V2 signatures in the wild (**RQ1**). Then we seek to investigate the detailed signing issues for the 5 millions apps we collected (**RQ2**).

A. RQ1: The Distribution of Signing Schemes.

V1 Signing Scheme vs. V2 Signing Scheme We have analyzed 2,951,017 distinct APKs in total. Although the V2 scheme has been introduced for 1.5 years prior to our crawling process, it is surprising to see that more than 93.7% of APKs (2,765,752) still use only the V1 signing scheme, while only 6.3% of APKs (185,150 in total) in our dataset have adopted the V2 signing scheme. Note that 185,139 apps use both signing schemes, which means 11 apps use only the V2 signatures. This result suggests that most of the apps in our dataset are exposed to the attack surfaces of V1 signatures, e.g., attackers

TABLE II
OVERALL RESULTS OF OUR MEASUREMENT STUDY.

Market	#Apps	Vulnerability		Attack		Compat.	Developing Bug			Total	Percentage
		Vul-1	Vul-2	Attack-1	Attack-2	C1	Zip Bug	V1 Bug	V2 Bug	-	-
Google Play	219,944	11	15,510	7	55	418	0	81	0	16,084	7.31%
Tencent	636,665	19,066	27,601	24	159	1,202	38,637	149	43	85,109	13.34%
Baidu	381,419	10,651	83,898	14	165	669	929	659	130	96,985	25.43%
360	162,584	2,713	9,478	9	91	795	272	32	10	13,363	8.22%
Huawei	106,672	157	45,324	0	10	580	9	95	52	46,191	43.3%
Xiaomi	169,502	1,321	53,147	0	142	698	8,904	7,313	1,834	72,980	43.06%
Wandoujia	560,662	6,916	33,641	17	178	1,215	1,073	254	67	43,174	7.70%
HiAPK	238,787	10,409	15,116	21	56	394	74	110	29	26,151	10.95%
AnZhi	225,659	21,590	21,843	6	90	607	247	455	52	44,739	19.8%
91	11,822	369	808	0	5	15	13	3	1	1,212	10.25%
OPPO	483,201	7,064	56,531	31	73	1,370	6,890	177	43	71,622	14.82%
25PP	1,060,464	11,860	55,477	33	289	22,58	1909	364	95	71,999	6.79%
Sougou	201,041	4,326	44,175	6	112	831	6,398	8,181	514	63,908	31.79%
Gfan	11,121	283	1,141	0	3	30	5	6	0	1,466	13.18%
Meizu	80,179	923	15,717	3	15	286	260	100	38	17,269	21.54%
DCN	18,796	769	4,612	0	5	199	11	35	5	5,588	29.73%
LIQUCN	198,034	3,931	23,707	6	112	1,214	357	152	36	29,426	14.86%
APPChina	39,092	809	4,925	1	14	116	77	46	14	5,970	15.27%
10086	4,640	97	369	0	1	10	5	12	0	493	10.63%
Lenovo MM	36,293	886	9,478	0	15	169	25	87	24	10,649	29.34%
ZOL	6,412	64	2,788	0	3	32	2	11	1	2,900	45.23%
NDUO	19,331	262	1,854	0	21	55	23	68	11	2,287	11.83%
CNMO	4,893	289	364	0	9	9	26	12	7	711	14.53%
PCOnline	140,905	5,131	9,372	0	55	340	1,734	80	24	16,524	11.73%
APPCool	12,769	130	965	0	4	38	28	34	17	1,212	9.49%
Total	-	65,374	370,138	94	1,110	8,518	62,311	17,836	2,849	-	-

could exploit various V1 vulnerabilities to perform attacks on devices with Android system versions prior to 7.0.

Apps with Multiple Signatures. Another interesting observation is that some apps have been signed by multiple certificates. For example, 128 apps in our dataset are signed by 2 signatures, 2,003 apps are signed by 3 signatures, 15 apps are signed by 4 signatures, and 39 apps are signed by 5 signatures. Note that Google Play does not accept APKs with multiple signatures, thus all these apps are found in alternative markets. One possible reason is that they tried to use multiple signatures to distinguish app versions or app release channels.

TABLE III
THE TOP 5 APPS USING *testkey* IN OUR DATASET.

Package Name (MD5)	App Installs	Market
com.shuqi.controller (04B8E1ED1F724E210BBBE6EBF75308A5)	100,000,000	Baidu
com.kiloo.subwaysurf (CFBAE893E9B7B25928C62BDEED8B3CEF)	100,000,000	Baidu
com.aiyou.mhsjuu (EE345CB5A869D27471ED40FAE4ED5BDF)	77,840,000	Tencent
com.cheercode.phonegame1 (AE976AA9669FBA6D42C8808D8B8F8456)	76,690,000	Tencent
com.mango.sanguo15.ruanyou (778A75DF40AE339B0796B6597FBE3BA1)	72,720,000	Tencent

B. RQ2: App Signing Issues

1) Overall Results: We present our primary exploration results on the 21 anti-patterns in Table II. Around 6.79% to 45.25% of apps in each market have been found containing at least one issue. Even in Google Play, roughly 7.3% of our crawled apps have various signing issues. For three alternative markets (Huawei, Xiaomi and ZOL), over 40% of the apps are exposed to signing risks. **This result suggests that the signing issues are prevalent across markets.**

The two vulnerabilities are most popular across markets, accounting for more than 80% of the apps with signing issues. To our surprise, over 65K apps were found using public known keys, which allow attackers to arbitrarily modify the apps without breaking its signatures. More than 370K apps contain unprotected contents, which offer opportunities for attackers to modify and replace them without developers' knowledge. 94 apps were found exploiting the Master Key vulnerability to perform attacks, and most of them were confirmed as malicious apps. Over 1,000 apps were found being compromised. Roughly 90K apps include release bugs or compatibility issues that may lead to unsuccessful installation.

C. Detailed Results

1) Vulnerabilities: Over 65K apps were found using the known keys to sign apps. Although the four keys provided by AOSP ("media", "platform", "shared" and "testkey") were released 10 years ago, all of them were found still being used in our dataset. There are 24 apps using the "media" key, 746 apps using the "platform" key, 23,619 apps using the "shared" key and 40,985 apps using the "testkey". **These apps have aggregated 5.8 billion app installs in total.** To our surprise, **even some popular apps (with millions of downloads) use these publicly known keys, which expose them to great security risks, e.g., attackers could arbitrarily modify the app without developers' knowledge.** Table III lists the top

5 apps ordered by app installs that use testkey. For example, app "com.shuqi.controller" is a famous novel reading app, and "com.kiloo.subwaysurf" is a popular game app. But they were found using "testkey" to sign themselves.

More than 553K apps in our dataset include some kind of unprotected contents in the META-INF directory. Note that it does not mean that all of them could be seriously attacked by hackers, as some of the unprotected contents are resource files that cannot incur serious security risks. However, it is still not recommended by Android to list unprotected files in META-INF, as **attackers could easily tamper with the app and modify it to replace the unprotected files without breaking the V1 signature**. The severity of risks introduced by this vulnerability depends on the content of the unprotected files.

2) Exploits: Although we have categorized 6 kinds of exploits and developed a tool to detect them, only two kinds of exploits have been found in our dataset. 94 apps (435K app installs in total) exploit the Master Key vulnerability to perform possible attacks. We further upload these apps to VirusTotal, an online malware detection service that embed more than 60 anti-virus engines. The result suggests that **85 of them are labelled as "Virus:Android.Masterkey" or "CVE-2013-4787".** Table IV lists top 3 such apps ordered by the number of flagged engines on VirusTotal.

The integrity of 1,110 apps were found being compromised, i.e., missing files listed in the MANIFEST.MF. **These apps have aggregated over 7.1 billion downloads in total.** We further investigate this issue, and found 156 of them with removed code, and the remaining ones with removed resource files. For example, the Baidu ad library "biduad_plugin/__pasys_remote_banner.jar" in app "com.aew.vbsz" has been removed. **One possible reason is that attackers try to remove ad libraries to create an ad-free app or compromise the functionalities of the apps. These attacks could be successfully performed on systems prior to 6.0 (cf. Section 2.3).** Roughly 40% of them have been flagged by at least one VirusTotal engine, and over 13% of them have been flagged by at least 10 anti-virus engines.

To identify attacks that exploit the Janus vulnerability, our tool automatically checks whether any of the apps add extra bytes to the start of Zip file. Although 4 such apps are found with this behavior, none of them attach the dex file. Thus, we did not find any cases that exploit the Janus Vulnerability.

Besides, we found 6 apps with long extra field lengths and 41 apps with inconsistent file name lengths between CD and LFH. However, all of them are breaking the resource files, while none of them break the classes.dex files. Furthermore, we did not identify apps that break the certificate chain.

3) Compatibility Issue: We analyzed the distribution of the digest/signature algorithms in our dataset, as shown in Figure 3. Most of the apps (over 90%) use SHA1WITHRSA and SHA256WITHRSA algorithms. We have identified 8,518 apps with compatibility issues in our dataset, and found that most of them are introduced by the *SHA256WITHRSA* algorithm. As *SHA256WITHRSA* is supported on API levels higher than 18, apps that declared *minsdkversion* lower than 18 will have compatibility issues, leading to installation failures on devices with lower API levels. **We even found many popular apps (with billions of downloads) with compatibility issues.** For example, "com.autonavi.minimap" is a popular app with over 1 billion installs that were found with compatibility issues, such that it cannot be installed on devices with earlier Android versions, although many old devices are still popular in countries such as China.

TABLE IV
APPS THAT EXPLOIT THE MASTER KEY VULNERABILITY.

Package Name (MD5)	App Installs	VT (# Engines, Flag)
air.com.baobaogame.MathBearAndroid.EN (53BD19EEED64F1182C993DB01CB11000)	500	23, MasterKey
air.com.shuchao.app.A36house (E4359956968BD988478652EB63F6D6B8)	96	14, CVE-2013-4787
air.GreenCloudSmasherFree (135EE5428EF595E8FB95581BC5F5F101)	10000	12, MasterKey, Revmob

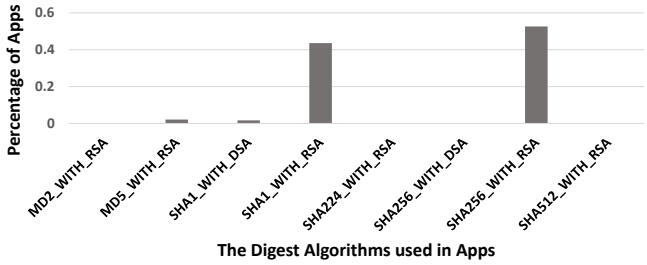

Fig. 3. The distribution of used digest algorithms.

4) Release Bugs: Most of the apps with release bugs were found in alternative markets. V2-related bugs and Zip bugs were not found in Google Play. 81 apps in Google Play have V1 bugs, and all of them belong to the Rollback Protection issue, i.e., the JAR signature file indicates that the APK is supposed to be signed with the V2 signature scheme (in addition to V1) but no V2 signature was found in the APK, which will lead to installation failure in the systems supporting V2 signatures. **A large portion of apps with Zip bugs were found in Tencent Myapp, Xiaomi and OPPO markets, which suggests that these markets do not enforce strict app regulation, as these apps are definitely low-quality apps that cannot be installed on any Android devices.** 17,836 apps in total have V1 bugs, and most of them (over 80%) belong to the issue that "mismatch between *.MF and JAR entry", which will lead to installation failure too. V2 related bugs were found in 2,849 apps, as only over 185K apps have adopted V2 signatures, over 1.5% of them have release bugs, which will lead to installation failures on devices with system versions higher than 7.0. This result suggested that many low-quality developers have little experiences in releasing apps and they were not even aware that their apps cannot be installed on any devices successfully.

D. The Distribution of App Signing Issues

We then analyze the distribution of app signing issues according to app category and app popularity (downloads).

	VUL-1	VUL-2	ATK-1	ATK-2	CMPT-1	BUG-V1	BUG-V2	BUG-Zip
READING	0.01	0.02	0.00	0.01	0.02	0.02	0.02	0.02
BROWSER	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
BUSINESS	0.01	0.06	0.00	0.03	0.05	0.03	0.05	0.02
COMMUNICATION	0.01	0.04	0.01	0.02	0.03	0.03	0.03	0.02
EDUCATION	0.00	0.07	0.02	0.03	0.10	0.07	0.15	0.03
ENTERTAINMENT	0.00	0.02	0.02	0.01	0.01	0.00	0.00	0.01
FINANCE	0.01	0.05	0.00	0.02	0.06	0.05	0.13	0.02
GAME	0.31	0.11	0.58	0.46	0.23	0.26	0.01	0.21
HEALTH	0.00	0.04	0.00	0.01	0.03	0.04	0.07	0.01
INPUT METHOD	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LIFE	0.07	0.14	0.07	0.06	0.11	0.13	0.15	0.09
LOCATION	0.00	0.04	0.00	0.02	0.03	0.03	0.06	0.02
MUSIC	0.00	0.02	0.00	0.01	0.01	0.01	0.01	0.01
NEWS	0.01	0.03	0.00	0.01	0.02	0.02	0.03	0.02
PERSONALIZATION	0.36	0.01	0.00	0.01	0.02	0.09	0.00	0.05
PHOTO	0.01	0.02	0.00	0.01	0.02	0.03	0.02	0.02
SECURITY	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SHOPPING	0.02	0.06	0.00	0.03	0.04	0.07	0.10	0.02
SOCIAL	0.00	0.03	0.01	0.01	0.03	0.03	0.07	0.01
TOOLS	0.04	0.06	0.00	0.03	0.05	0.03	0.04	0.04
VIDEO	0.00	0.02	0.00	0.01	0.02	0.01	0.03	0.02
OTHER	0.12	0.15	0.28	0.19	0.14	0.02	0.00	0.36

Fig. 4. The distribution of app signing related issues across app categories. Each column adds up to 100%.

	0-100	100-1k	1k-10k	10k-100k	100k-1M	1M-10M	10M-100M	>100M
VUL-1	0.09	0.18	0.27	0.36	0.09	0.00	0.00	0.00
VUL-2	0.07	0.18	0.23	0.22	0.15	0.11	0.03	0.00
ATTACK-1	0.14	0.86	0.00	0.00	0.00	0.00	0.00	0.00
ATTACK-2	0.13	0.36	0.24	0.16	0.09	0.00	0.02	0.00
COMPATIBILITY	0.11	0.21	0.28	0.19	0.13	0.08	0.01	0.00
BUG-V1	0.10	0.25	0.27	0.20	0.15	0.04	0.00	0.00
BUG-V2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
BUG-ZIP	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

(1) Google Play

	0-100	100-1k	1k-10k	10k-100k	100k-1M	1M-10M	10M-100M	>100M
VUL-1	0.60	0.23	0.11	0.04	0.01	0.00	0.00	0.00
VUL-2	0.45	0.19	0.15	0.11	0.07	0.02	0.01	0.00
ATTACK-1	0.75	0.17	0.08	0.00	0.00	0.00	0.00	0.00
ATTACK-2	0.59	0.16	0.14	0.08	0.02	0.00	0.00	0.00
COMPATIBILITY	0.55	0.17	0.12	0.10	0.04	0.01	0.00	0.00
BUG-V1	0.46	0.19	0.13	0.10	0.09	0.03	0.00	0.00
BUG-V2	0.14	0.09	0.12	0.30	0.19	0.14	0.02	0.00
BUG-ZIP	0.92	0.05	0.03	0.01	0.00	0.00	0.00	0.00

(2) Tencent Myapp

	0-100	100-1k	1k-10k	10k-100k	100k-1M	1M-10M	10M-100M	>100M
VUL-1	0.05	0.27	0.45	0.18	0.04	0.01	0.00	0.00
VUL-2	0.07	0.28	0.33	0.22	0.07	0.02	0.00	0.00
ATTACK-1	0.06	0.59	0.35	0.00	0.00	0.00	0.00	0.00
ATTACK-2	0.08	0.33	0.37	0.17	0.04	0.01	0.00	0.00
COMPATIBILITY	0.10	0.38	0.33	0.14	0.03	0.02	0.01	0.00
BUG-V1	0.14	0.26	0.30	0.18	0.11	0.02	0.00	0.00
BUG-V2	0.10	0.24	0.22	0.31	0.07	0.04	0.00	0.00
BUG-ZIP	0.07	0.38	0.33	0.19	0.03	0.00	0.00	0.00

(3) Wandoujia

	0-100	100-1k	1k-10k	10k-100k	100k-1M	1M-10M	10M-100M	>100M
VUL-1	0.00	0.76	0.15	0.08	0.02	0.00	0.00	0.00
VUL-2	0.00	0.26	0.27	0.20	0.15	0.09	0.02	0.00
ATTACK-1	0.00	0.90	0.06	0.03	0.00	0.00	0.00	0.00
ATTACK-2	0.00	0.70	0.18	0.08	0.01	0.03	0.00	0.00
COMPATIBILITY	0.00	0.43	0.31	0.18	0.05	0.02	0.01	0.00
BUG-V1	0.00	0.31	0.25	0.21	0.18	0.04	0.01	0.00
BUG-V2	0.00	0.05	0.21	0.35	0.37	0.02	0.00	0.00
BUG-ZIP	0.00	0.97	0.02	0.01	0.00	0.00	0.00	0.00

(4) OPPO

Fig. 5. The distribution of downloads for the apps with signing issues.

1) Category: Note that each Android market implements its own app taxonomy. For example, Google Play defines 33 app categories (excluding subcategories for Game), while Huawei Market only has 18 categories. In order to understand the general distribution of app signing issues across categories, we manually developed a consolidated taxonomy containing 22 app categories, and map all the categories of 25 app markets to this taxonomy, as shown in Figure 4.

As shown in Figure 4, **most app signing issues were found in categories including GAME, LIFE, PERSONALIZATION, and EDUCATION**. One possible reason is that these categories are most popular across markets. For example, roughly 50% of the apps that exploit existing vulnerabilities to perform attacks were found in the GAME category, while roughly 36% of apps with publicly known signature were found in PERSONALIZATION. Most of the V2 related bugs were found in EDUCATION, LIFE and FINANCE, while Zip and V1 related bugs were found mostly in the GAME category.

2) App Popularity: We further investigate the correlation between signing issues and app popularity (the number of app downloads). Note that, as the app downloads across different app markets vary greatly, we perform per-market analysis here. Figure 5 listed the distribution of signing issues in each of the 4 top app markets due to space limitation.

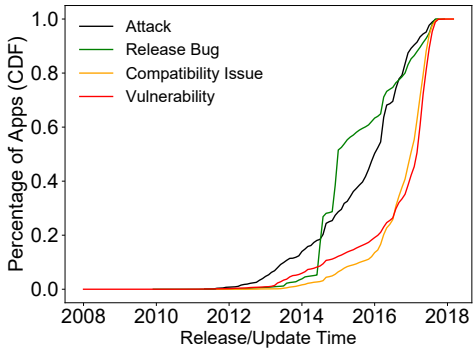

Fig. 6. The distribution of release/update time for all the identified apps.

To our surprise, although the apps with less popularity ($< 10K$) account for a large portion of the issues, **a considerable number of them are popular apps with millions of downloads**. For example, over 10% of the apps with issues found in Google Play have downloads higher than 1 million. The result is more prevalent on popular apps of the alternative markets, roughly 20% apps with V1 bugs and V2 bugs in the Baidu market have more than 1 million app installs, suggesting that these apps could introduce installation failure issues (at least in some system versions), which will greatly affect users' experience and the market's reputation as well.

The apps with vulnerability and compatibility issues were distributed across different app download ranges. In Google Play, roughly 10% of the apps with known signatures have installs higher than 100K, and roughly 10% of the apps with compatibility issues have installs higher than 1 million. The exploited apps (attacks) in general have fewer app downloads. For example, all of the apps utilize Janus vulnerabilities have less than 1K installs in Google Play. More than 97% of apps with Zip bugs have less than 1K app downloads in Tencent and OPPO markets, while the numbers in Baidu and Wandoujia markets are outliers, which are 39% and 45% respectively.

VI. The Evolution of Signing Issues

In this section, we study the evolution of signing issues (**RQ3**). We first analyzed the release/update time of these apps with signing issues. As most of the issues would lead to great security risks or installation failures, we want to examine how long they have been staying in the markets. Then we perform a post analysis seven months later to measure how many apps with signing issues have been removed or mitigated.

1) Release/Update Time: As shown in Figure 6, we have investigated the distribution of release/update time for all the issues we identified across four categories. **Over half of the bugs and attacks were released before 2016, which means that they have affected millions of users for at least two years (2 years prior to our first crawling process)**.

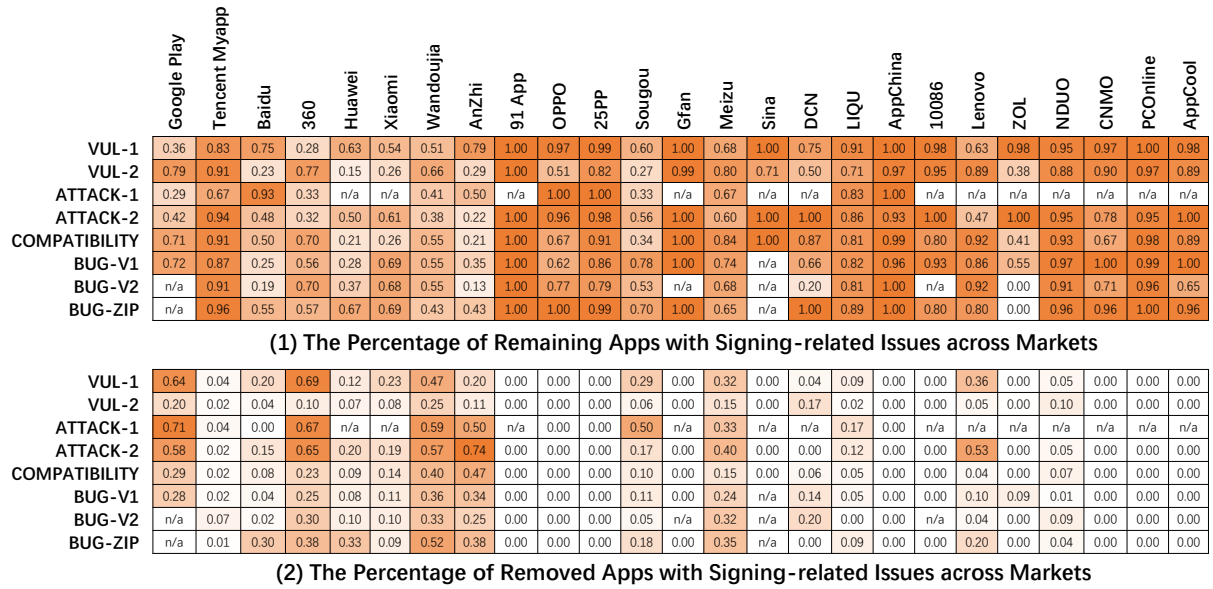

Fig. 7. The percentage of remaining/removed apps with signing related issues across markets after 7 months.

Over 70% of the vulnerabilities and compatibility issues were released during 2016 and 2018.

This result suggested that, on one hand, **the developers of these apps usually paid little attention to the signing issues**, and some of them are even unaware whether their apps could be successfully installed on users' devices. On the other hand, **although each app market claims to enforce app inspection on malicious code and app clones, they do not enforce strict/any app inspection on signing issues**, especially for alternative markets (e.g., Tencent, Baidu and Xiaomi) with a large number of low-quality apps with signing issues.

2) Post Analysis: Our results reveal that each market hosts a significant number of apps with signing issues. We performed a second app crawling on each app store about 7 months after the first one in order to quantify: (1) whether the app markets made any effort to remove those samples and (2) whether the app developers identified the issues and updated the apps.

For each market, we crawled the apps with signing issues and labeled them as: (1) remaining unchanged (identical APK MD5 hashing), (2) removed, and (3) updated (to a newer version). Note that we exclude HiAPK from the analysis as it has discontinued its services before our second crawling.

Figure 7 shows the percentage of the remaining and removed apps with signing issues. To our surprise, for 11 out of 25 markets, more than 90% of the apps with signing issues still remain in the market without any updates, as shown in Figure 7(1). The situation is the worst for the 91, Gfan, AppChina and PC Online markets, almost all of the apps with signing issues remained in the markets without any updates. Anzhi, Huawei and Google Play addressed part of the vulnerable apps, but more than 50% of the apps with signing issues still remained in the markets. By further analyzing the removed vulnerable apps (cf. Figure 7(2)), it is interesting to see that almost none of the apps with signing issues were removed in 12 alternative markets (e.g., 91, OPPO, 25PP). Google Play, 360 market, Wandoujia and Anzhi have removed the most number of the apps with signing issues. On a per-issue basis, most of the apps that are vulnerable to attacks (cf. Section III-B) have been removed in Google Play, 360 and Anzhi. However, most of the apps with compatibility issues, V1 bug issues, and V2 bug issues did not get any updates.

This result suggests that the app regulation and app maintenance behaviors across markets differ significantly. Most of the alternative markets do not even remove the risky apps, which could lead to severe consequences such as compromising app users' security and privacy and having negatively impacts on the brands of the app markets.

VII. Discussions

A. Implication

We believe that our efforts can positively contribute to different stakeholders in the mobile app ecosystem.

App Markets. We found most markets paid little attention to security issues introduced in the app signing process. These markets host a large number of low-quality apps and developers, and are even exploited by malicious developers to disseminate malware. Thus, app markets should (1) improve their app regulation process to eliminate apps with signing issues before they enter the market, and (2) deploy automated tools to identify/remove apps with signing issues, remove low-quality developers, identify signing-related attacks, and improve the app ecosystem.

App Developers. Experiment results suggested that many app developers are unaware of the signing issues. Very few developers have adopted the V2 signatures, while many developers use publicly-known private keys. Even some popular app developers cannot deal with app signing correctly and many of them suffer from compatibility issues. Our work could help

app developers identify and eliminate these signing issues, thus helping improve app quality.

App Users. As most of the issues listed in this paper were focused on the V1 signing scheme, a majority of them cannot cause serious security issues on Android systems that support V2 and V3 signing schemes (after 7.0). Our work could help app users be aware of the severity of signing issues, and further eliminate the issues by updating their devices to up-to-date system versions and choose apps from high-quality markets.

Research Community. Our work could help encourage more research on app signing issues, such as additional app signing issues and advanced approach to identify them. Besides, further studies could focus on the new signing mechanisms and third-party signing frameworks (e.g., Baidu OASP [19]), as well as automated exploitation of apps with signing issues and how to identify such attacks.

B. Threats to Validity

To the best of our knowledge, this work is the first attempt in the community towards characterizing app signing issues in large scale. Our study, however, carries several limitations.

First, we focus on 21 kinds of signing issues, which were summarized from CVE and existing technical reports. This taxonomy might be incomplete, and we did not identify new app signing vulnerabilities or anti-patterns. Nevertheless, it is surprising to see that a large number of apps have been exposed to security issues, although these issues were known to the community for a long time. Besides, our measurement study is limited by our dataset. On one hand, our dataset is a bit outdated and does not cover the V3 signing scheme. On the other hand, most of the apps in our dataset were crawled from Chinese alternative app markets, which is not representative enough to characterize the global app signing issues.

VIII. Related Work

Measurement Study of App Security Issues. A number of studies have measured the mobile app ecosystem in large scale from different angles, including malware [20], [21], [22], [23], repackaged apps [24], [25], [26], low-quality apps [27], [28], permission issues [29], [30], [31], [32], third-party tracking [33], [34], [35], [36], [37], [30], fraudulent behaviors [38], [39], [40], [41], and promotion attacks [42], [43], [44], etc. Besides, cryptography APIs have been widely studied in the mobile app ecosystem [45], [46], [47], [48], [49]. For example, Egele *et al.* [45] empirically analyzed cryptographic misuses in Android apps and found that over 88% of apps make at least one mistake. iCryptoTracer [46] investigated the iOS app's usage of cryptographic APIs and observed that more than 65% of iOS apps contain various security flaws caused by cryptographic misuses. Backes *et al.* [47] observed the misuse of cryptographic APIs in mobile ad libraries. Wang *et al.* [49] proposed a framework to investigate OAuth implementation issues and found that 86.2% of the apps incorporating OAuth services are vulnerable. Many other studies are focused on analyzing vulnerabilities including the SSL/TLS issues [50], [51], [52], the open port vulnerability [53] and the ICC issues [54], [55], [56], [57], etc.

App Authorship/Developer Analysis. Oltrogge *et al.* [58] have investigated online app generators and found that they are suffering from well-known security issues, while developers are unaware of these hidden problems. CredMiner [59] studied the prevalence of unsafe developer credentials and found that over half of them using free email services and Amazon AWS are vulnerable. Gonzalez *et al.* [60] proposed to analyze the authorship attribution of Android apps based on a set of features extracted from the decompiled binary. Their results suggested that they could achieve 97.5% accuracy on identifying developers. Wang *et al.* [61] performed the first large scale study of the mobile app ecosystem from the perspective of app developers. The results suggested that over 70% of the apps with severe privacy risks are created by 1% developers. The follow-up work [62] has analyzed over 1 million Android app developers across Google Play and 16 popular alternative markets, from different angles including developing, releasing, app maintenance and malicious behaviors.

App Signing for Securing App Installation. Barrera *et al.* [63] conducted a detailed analysis of the app installation process to study the update integrity and UID assignment. They found empirical evidence that Android's current signing architecture does not encourage best security practices, and the limitations of Android's UID sharing method force developers to write custom code for secure data transfer. Baton [64], provides a mechanism to enable transparent key updates or certificate renewals. These work were focused on exposing the limitations of the signing architecture, while our work conducted a systematic study to identify issues in existing apps and shed light on attacks on the apps.

IX. Conclusion

In this work, we have conducted a large-scale measurement study of Android app signing issues in the wild. We first created a taxonomy of 21 anti-patterns, and then developed an automated tool for conducting the characterization study. We have studied over 5 million apps across 25 markets, and revealed that various signing issues are prevalent in Google Play and alternative markets. Furthermore, evolution analysis suggested that most app markets have paid little attention to the security issues caused by app signing, as almost all the apps with signing issues were not removed/mitigated. We believe that our research efforts can positively contribute to bring developer and mobile user awareness of signing issues, attract the interests of the research community and regulators, and promote best operational practices across market operators.

Acknowledgment

This work is supported in part by the National Key Research and Development Program (2017YFB1001904), the National Natural Science Foundation of China (grants No.61702045, No.61772042 and No.U1836211), NSF (CNS-1755772), Beijing Natural Science Foundation (No.JQ18011) and a research grant from Huawei. Yao Guo is the corresponding author.

REFERENCES

[1] (2018) Apk signing. [Online]. Available: https://source.android.com/security/apksigning/

[2] "New Android vulnerability allows attackers to modify apps without affecting their signatures," 2019, https://www.guardsquare.com/en/blog/new-android-vulnerability-allows-attackers-modify-apps-without-affecting-their-signatures.

[3] "CVE-2017-13156," 2019, https://nvd.nist.gov/vuln/detail/CVE-2017-13156.

[4] "APK duplicate file vulnerability," 2019, http://www.androidvulnerabilities.org/ vulnerabilities/APK_duplicate_file.

[5] "Android Bug Superior to Master Key," 2018, http://www.saurik.com/id/18.

[6] "Yet Another Android Master Key Bug," 2018, http://www.saurik.com/id/19.

[7] "Hackers are selling legitimate code-signing certificates to evade malware detection," 2018, https://news.ycombinator.com/item?id=16445845.

[8] Oracle. (2018) Signed jar file. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html#Signed_JAR_File

[9] "Scheme v2," 2018, https://source.android.com/security/apksigning/v2.

[10] "Scheme v3," 2018, https://source.android.com/security/apksigning/v3.

[11] "Android Distribution and Android Market Share," 2018, https://data.apteligent.com/android/.

[12] "Android Vulnerability," 2018, http://www.androidvulnerabilities.org/all.

[13] (2018) First malicious use of 'master key' android vulnerability discovered. [Online]. Available: https://www.symantec.com/connect/blogs/first-malicious-use-master-key-android-vulnerability-discovered

[14] "APK unsigned shorts," 2018, http://www.androidvulnerabilities.org/vulnerabilities/APK_unsigned_shorts.

[15] "APK unchecked name," 2018, http://www.androidvulnerabilities.org/vulnerabilities/APK_unchecked_name.

[16] "FakeID," 2019, https://androidvulnerabilities.org/vulnerabilities/Fake_ID.

[17] (2018) Add api to check certificate chain signatures. [Online]. Available: https://android.googlesource.com/platform/libcore/+/2bc5e811a817a8c667bca4318ae98582b0ee6dc6

[18] "apksigner," 2018, https://developer.android.com/studio/command-line/apksigner.

[19] (2019) Online app status protocol. [Online]. Available: https://github.com/baidu/OASP

[20] H. Wang, Z. Liu, J. Liang, N. Vallina-Rodriguez, Y. Guo, L. Li, J. Tapiador, J. Cao, and G. Xu, "Beyond google play: A large-scale comparative study of chinese android app markets," in *Proceedings of the 2018 Internet Measurement Conference (IMC '18)*, 2018, pp. 293–307.

[21] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *Proceedings of the 2017 International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '17)*, 2017, pp. 252–276.

[22] H. Wang, H. Li, and Y. Guo, "Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of google play," in *Proceedings of the The World Wide Web Conference (WWW '19)*, 2019, pp. 1988–1999.

[23] H. Wang, J. Si, H. Li, and Y. Guo, "Rmvdroid: towards a reliable android malware dataset with app metadata," in *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*, 2019, pp. 404–408.

[24] Y. Ishii, T. Watanabe, M. Akiyama, and T. Mori, "Appraiser: A large scale analysis of android clone apps," *IEICE TRANSACTIONS on Information and Systems*, vol. 100, no. 8, pp. 1703–1713, 2017.

[25] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: A scalable and accurate two-phase approach to android app clone detection," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA '15)*, 2015, pp. 71–82.

[26] L. Li, T. F. Bissyandé, H.-Y. Wang, and J. Klein, "On identifying and explaining similarities in android apps," *Journal of Computer Science and Technology*, vol. 34, no. 2, pp. 437–455, 2019.

[27] H. Wang, H. Li, L. Li, Y. Guo, and G. Xu, "Why are android apps removed from google play?: a large-scale empirical study," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*, 2018, pp. 231–242.

[28] V. N. Inukollu, D. D. Keshamoni, T. Kang, and M. Inukollu, "Factors influencing quality of mobile apps: Role of mobile app development life cycle," *arXiv preprint arXiv:1410.4537*, 2014.

[29] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the android permissions system," in *Proceedings of the 28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 603–620.

[30] H. Wang, Y. Li, Y. Guo, Y. Agarwal, and J. I. Hong, "Understanding the purpose of permission use in mobile apps," *ACM Transactions on Information Systems (TOIS)*, vol. 35, no. 4, p. 43, 2017.

[31] H. Wang, Y. Guo, Z. Tang, G. Bai, and X. Chen, "Reevaluating android permission gaps with static and dynamic analysis," in *Proceedings of the 2015 IEEE Global Communications Conference (GLOBECOM '15)*, 2015, pp. 1–6.

[32] H. Wang, J. Hong, and Y. Guo, "Using text mining to infer the purpose of permission use in mobile apps," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*, 2015, pp. 1107–1118.

[33] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, and C. K. P. Gill, "Apps, trackers, privacy, and regulators," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS '18)*, 2018.

[34] 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 *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP '16)*, 2016, pp. 357–376.

[35] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th international conference on software engineering companion*, 2016, pp. 653–656.

[36] L. Li, T. Riom, T. F. Bissyandé, H. Wang, J. Klein *et al.*, "Revisiting the impact of common libraries for android-related investigations," *Journal of Systems and Software*, vol. 154, pp. 157–175, 2019.

[37] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: scalable and precise third-party library detection in android markets," in *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*, 2017, pp. 335–346.

[38] B. Andow, A. Nadkarni, B. Bassett, W. Enck, and T. Xie, "A study of grayware on google play," in *Proceedings of the 2016 IEEE Security and Privacy Workshops (SPW)*, 2016, pp. 224–233.

[39] Y. Hu, H. Wang, Y. Zhou, Y. Guo, L. Li, B. Luo, and F. Xu, "Dating with scambots: Understanding the ecosystem of fraudulent dating applications," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[40] Y. Hu, H. Wang, L. Li, Y. Guo, G. Xu, and R. He, "Want to earn a few extra bucks? a first look at money-making apps," in *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*, 2019, pp. 332–343.

[41] F. Dong, H. Wang, L. Li, Y. Guo, T. F. Bissyandé, T. Liu, G. Xu, and J. Klein, "Frauddroid: Automated ad fraud detection for android apps," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*, 2018, pp. 257–268.

[42] H. Zhu, H. Xiong, Y. Ge, and E. Chen, "Discovery of ranking fraud for mobile apps," *IEEE Transactions on knowledge and data engineering*, vol. 27, no. 1, pp. 74–87, 2014.

[43] B. Sun, X. Luo, M. Akiyama, T. Watanabe, and T. Mori, "Padetective: A systematic approach to automate detection of promotional attackers in mobile app store," *Journal of Information Processing*, vol. 26, pp. 212–223, 2018.

[44] Q. Guo, H. Wang, C. Zhang, Y. Guo, and G. Xu, "Appnet: understanding app recommendation in google play," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics (WAMA '19)*, 2019, pp. 19–25.

[45] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 20th ACM Computer and Communications Security Conference (CCS '13)*, 2013, pp. 73–84.

[46] Y. Li, Y. Zhang, J. Li, and D. Gu, "icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications," in *Proceedings of the International Conference on Network and System Security*, 2014, pp. 349–362.

[47] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 23rd ACM Computer and Communications Security Conference (CCS '16)*, 2016, pp. 356–367.

[48] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler *et al.*, "Cognicrypt: Supporting developers in using cryptography," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*, 2017, pp. 931–936.

[49] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu, "Vulnerability assessment of oauth implementations in android applications," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 61–70.

[50] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in) security," in *Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12)*, 2012, pp. 50–61.

[51] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smvhunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[52] L. Onwuzurike and E. De Cristofaro, "Danger is my middle name: experimenting with ssl vulnerabilities in android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2015, p. 15.

[53] Y. J. Jia, Q. A. Chen, Y. Lin, C. Kong, and Z. M. Mao, "Open doors for bob and mallory: Open port usage in android apps and security implications," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P '17)*, 2017, pp. 190–203.

[54] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, 2015, pp. 280–291.

[55] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (AsiaCCS '17)*, 2017, pp. 71–85.

[56] A. Sadeghi, H. Bagheri, and S. Malek, "Analysis of android inter-app security vulnerabilities using covert," in *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, 2015, pp. 725–728.

[57] Y. K. Lee, J. Y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic, "A sealant for inter-app security holes in android," in *Proceedings of the 2017 International Conference on Software Engineering (ICSE '17)*, 2017, pp. 312–323.

[58] M. Oltrogge, E. Derr, C. Stransky, Y. Acar, S. Fahl, C. Rossow, G. Pellegrino, S. Bugiel, and M. Backes, "The rise of the citizen developer: Assessing the security impact of online app generators," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP '18)*, 2018, pp. 634–647.

[59] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, "Harvesting developer credentials in android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2015, pp. 23:1–23:12.

[60] H. Gonzalez, N. Stakhanova, and A. A. Ghorbani, "Authorship attribution of android apps," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (CODASPY '18)*, 2018, pp. 277–286.

[61] H. Wang, Z. Liu, Y. Guo, X. Chen, M. Zhang, G. Xu, and J. Hong, "An explorative study of the mobile app ecosystem from app developers' perspective," in *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*, 2017, pp. 163–172.

[62] H. Wang, X. Wang, and Y. Guo, "Characterizing the global mobile app developers: a large-scale empirical study," in *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems (MobileSoft '19)*, 2019, pp. 150–161.

[63] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot, "Understanding and improving app installation security mechanisms through empirical analysis of android," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices (SPSM '12)*, 2012, pp. 81–92.

[64] D. Barrera, D. McCarney, J. Clark, and P. C. van Oorschot, "Baton: Certificate agility for android's decentralized signing infrastructure," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks (WiSec '14)*, 2014, pp. 1–12.