More Than You Signed Up For: Exposing Gaps in the Validation of Android's App Signing

Norah Ridley, Enrico Branca, Natalia Stakhanova

University of Saskatchewan, Canada
norah.ridley@usask.ca
enrico.branca@usask.ca,natalia@cs.usask.ca

Abstract. Android's ubiquitous flexibility has helped it to become one of the most widely used mobile operating systems in the world. However, the convenient and extensive access to phone resources adopted by Android has revealed inefficiencies of its existing protections. Among these protections is application (app) signing. This process is intended to maintain the integrity of the app after it is released, and it provides users with confidence in the app's authenticity. We analyze the functionality of Android signature verification and identify the logical gaps in the process that can be used to hide a malicious payload. We demonstrate the security implications of these gaps.

1 Introduction

Android has become one of the most widely used mobile operating systems (OS) in the world. Central to Android's ecosystem is its app signing mechanism, a security feature designed to ensure the integrity and authenticity of mobile applications (apps). By requiring developers to digitally sign their apps, this process helps to ensure that the app has not been tampered with since its release and to verify the identity of developers. Despite its importance, this mechanism has flaws and has faced scrutiny over the years. Numerous examples have highlighted the limitations of app signing.

In this paper, we examine the app signing process in recent Android versions. We demonstrate how its design flaws and inconsistent implementation can be exploited by attackers, rendering mobile attacks virtually undetectable.

Over the years, Android has strengthened its app integrity protection by gradually introducing more restrictive signing schemes. The v1 signature scheme was the Android's original mechanism for signing applications. Based on the Java Archive (JAR) file signing format, it used to aggregate multiple files into a single file for easier distribution. In the v1 scheme, a digital signature was created for each file within the Android Package Kit (APK) format, which offered integrity protection of individual files but not of the entire APK. The limited protection provided by the v1 signature scheme resulted in the infamous Janus vulnerability that allowed attackers to inject malicious code into APK files without altering their original cryptographic signatures [13]. Similarly, the Master Key vulnerability enabled modifications to previously signed applications while preserving the

validity of their original signatures [12]. Due to these issues, Android introduced more robust v2 and v3 signature schemes that implement several mechanisms to provide stronger guarantees for app integrity.

In this work, *we show fundamental issues in the design and validation of the APK signing block.*

First, both schemes leverage a separate APK signing block to store app signatures. This block contains a sequence of signatures in the form of ID-value pairs prefixed and suffixed with the data size. The key observation that makes our attacks possible is that the sequence of signatures is never verified. During app signature verification, known ID-value pairs are evaluated while unknown pairs are ignored. As a result, an APK signature block provides space for embedding an attack payload.

Second, unlike v1, which only verified individual files within the APK, v2 and v3 validate the integrity of the entire APK file to prevent unauthorized modifications to the APK after signing. The verification process is simplified by using a single hash of the entire APK. To be precise, the hash is calculated only for APK content (ZIP entries), the ZIP Central Directory, and the End of Central Directory (EoCD). Although the integrity of the signed data inside the APK signing block is protected, the signing block as a container is left without integrity verification, making it possible to modify the signing block after signing.

Third, the APK signing block is intentionally variable in size, allowing it to store digital signatures as well as any additional metadata. This design inherently offers significant space for embedding attack payloads.

In this paper, *we leverage these observations and show how the discovered issues in the v2 and v3 signature schemes make it possible to craft an Android app that can deliver and execute a malicious payload without raising any flags during the official app verification process.*

Our proposed attack leverages the internal structure of the signing block, which provides a variable space to embed an attack payload. We demonstrate that this approach can successfully deliver a wide range of attack payloads, including text and code, to Android devices through the APK signing block of a seemingly benign app that does not request any dangerous permissions. In addition, we show that these seemingly benign apps containing the payloads can be successfully distributed through traditional app distribution platforms. We upload the proof-of-concept app to the APKPure[1] and Aptoide[2] markets and bypass their security checks.

Responsible disclosure. We followed the responsible disclosure process and informed Google of the discovered issue in November 2024. Upon receiving our report, the Android security team assigned it a critical severity. In January 2025, Google determined that the identified issue was not a security vulnerability, which nevertheless required remediation. As a result of our disclosure, Google informed us that the Google Play's developer console will be updated to reject pre-signed APKs that contain unknown or unsupported IDs.

[1] https://apkpure.com/

[2] https://en.aptoide.com/

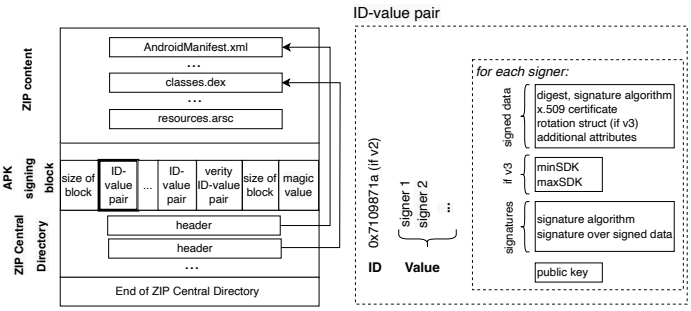

Fig. 1: APK structure

2 Background

The Android application file is a self-contained ZIP container that includes the compiled code, assets, and resources. Android apps can be installed on the device or published on the app distribution platforms using the APK or Android App Bundle (AAB) formats. The Play Store, Google's official store, is one option for distributing apps. Alternatively, a developer can upload an app to third-party markets, many of which are popular in some countries (e.g., China, India).

Regardless of the format, Android requires all apps to be digitally signed before they are installed on a device or updated by the developer. For apps using APKs, the developer has to manually sign the app using the app signing key. The app signing key refers to a pair of cryptographic keys: a public key and a private key. The public key is embedded within the X.509 certificate along with the app's developer and key information. The certificate is then included in the APK file. Apps distributed in Android App Bundle format are signed with an upload key first and then uploaded to Google Play Store where, after verification of the upload certificate, they are signed with the app signing key. While many markets support apps in both formats (e.g., Samsung Galaxy Store), Google Play Store began restricting APK format in 2021. This change only allows developers to update apps in APK format.

App signing. Application signing serves two main purposes: developer identification and integrity verification. Essentially, this key mechanism verifies that an app delivered to a user's phone is unmodified. Currently, Android supports four APK signature schemes [6]: v1, v2, v3, and v4.

The v1 signature scheme, the original signing scheme, was introduced in Android 1.6 and is based on the JAR signing mechanism. It verifies the integrity of individual files within the APK, ensuring that none of the signed files has been tampered with. The v1 scheme is used to verify older apps designed for Android 7.0. To address the limitations of v1, Android introduced *the v2 signature scheme* in Android 7.0 [4]. Unlike v1, v2 validates the entire APK, including ZIP metadata. This feature ensures that any modification to the APK invalidates the signature. *The v3 signature scheme*, introduced in Android 9.0, builds on v2 by

adding support for key rotation. This allows developers to specify new signing keys for future app updates, enabling them to replace compromised or outdated keys. The most recent addition, *the v4 signature scheme*, was introduced in Android 11. The v4 scheme requires a complementary v2 or v3 signature. By default, the v4 signature is stored in a separate file [5]. While v4 is used by Google Play, it is not mandatory for APKs.

Once an app is installed on an Android device, the package manager performs signature verification and records metadata about the app including its signing data and the signing certificate.

APK structure. An overview of the APK structure is shown in Figure 1. From a signing perspective, the APK is divided into four sections: the contents of ZIP entries (APK content), the APK Signing Block, the ZIP Central Directory (which contains offsets to ZIP entries), and the End of Central Directory (EoCD). The APK Signing Block, introduced with the v2 signing scheme, contains v2 and v3 signatures while maintaining backward compatibility with the v1 scheme. The APK signing block starts with an eight-byte *size of block* field that specifies the total size of the APK signing block. The size field is followed by a sequence of ID-value pairs, then a second eight-byte *size of block* field, which is expected to store the same value as the first field. The final element in the signing block is the magic value "APK Sig Block 42" that is used to identify the signing block during parsing and also to mark the end of the signing block.

The ID-value pair sequence contains information about the app's signers, with each pair representing a specific signing scheme identified by its unique ID. For every signer, this includes the *signed data*, *signature*, and the *public key*. The signed data part contains APK digests (digests of APK content, the ZIP Central Directory, and EoCD), signature algorithms, signer's x.509 certificate, and signer's public key. In addition to signing schemes identified by their unique IDs, Android defines several other IDs recognized by the verification process. One such ID corresponds to the *verity ID-value pair*. The primary purpose of this pair is to maintain the alignment of the APK signing block. It does not contain meaningful signing information and consists of padding to align the block size as a multiple of 4,096 bytes although this alignment is not enforced. During signing, `apksigner` positions the verity ID-value as the last entry in the ID-value pair sequence. The verity pair is followed by the size field and the magic value.

Integrity verification of an APK is based on the digests of the file content. The digest is computed over APK content, the ZIP Central Directory, and EoCD by splitting them into chunks and calculating the digest for each of them separately. The resulting digests are combined in a Merkle tree fashion into one or more APK digests that are stored in the signed data part of the block. Signature algorithm IDs used in the generation of digests are stored in both the signed data part and in the signature part of the signing block.

3 Related work

Android malware and attacks. Over the years, several studies have demonstrated that app repackaging is an effective method for distributing malware onto

users' devices [9, 17, 21]. Poeplau et al. [14] showed how malware can use dynamic code-loading techniques to circumvent offline vetting mechanisms (e.g., Google's Bouncer). Their analysis of apps from the Google Play Store demonstrated that although benign apps have legitimate reasons for using code-loading techniques, developers often improperly implement them. Zhang et al. [20] showed that Android's data cleanup mechanism is ineffective, leaving data residue (including private information) after an app is uninstalled. Du et al. [7] identified significant vulnerabilities in Android's shared storage system. To illustrate the severity of their findings, the authors crafted an end-to-end attack that impersonated both users during a voice chat session. Ruggia et al. [15] mounted a state inference-based phishing attack by exploiting Linux's *inotify* component. Their proposed attack assumed that the attacker knew the installation path of their target app, and thus, had the potential to affect all apps regardless of the Android version. While these attacks have an impact on signed apps, they do not specifically target app integrity protections. In contrast, our work focuses on exploiting the limitations of these protection mechanisms to introduce arbitrary content into the signing block.

Android app signing issues Numerous studies have considered app signing issues in real-world apps. Vidas and Christin [16] gathered 41,057 apps from 194 alternative markets and 35,423 apps from the Google Play Store. Their subsequent analysis of the dataset showed the heavy reuse of signing certificates. For example, 48% of the certificates from Google Play were reused. Fahl et al. [8] assessed 989,935 free apps from Google Play and found that app signing practices were generally not transparent or secure. The dataset collected by Lindorfer et al. [11] contained over one million apps from between 2010 and 2014 with approximately 40% being identified as malware. They noted that 2.26% of the apps were signed with a publicly available test key, making them potentially vulnerable to attacks. They also observed the Master Key vulnerability being exploited in 1,152 samples (0.11%), all from 2013 and 2014, and only in malware. The study by Wang et al. [18] showed that in spite of the introduction of v2 and v3 signing schemes, over 93% of the apps only used the v1 scheme. Furthermore, over 65,000 apps were signed with publicly known keys, which could allow attackers to arbitrarily modify such apps without breaking their original signatures. These studies do not examine the underlying reason for app signing issues. In this work, we consider these problems by introducing a new form of attack that exploits the failures of the verification process.

4 Attack

4.1 Threat model

In our model, we assume that the attacker's goal is to successfully install malware on the target user's device. Specifically, the malicious app carrying the payload must: ① remain undetected during the verification and review process that apps undergo before being distributed to users, and ② successfully install on the device and execute the payload.

The app can be an original app developed and signed by the attacker that appears benign but conceals the malicious payload in the signing block. Similar to other attacks on Android, our approach assumes the victim has been deceived into installing a malicious app [7, 10, 15, 19, 20]. Once installed on the device, the app must be run locally by the user to trigger the payload, which will be executed within its application sandbox. As such, the effectiveness of the payload is limited by the privileges and data access that the app itself is granted. To reduce the chances of raising suspicion, the app's permission requirements are kept minimal. In fact, no permissions are needed to activate the embedded payload. The app in this attack must have a signing block, i.e., it is signed with the v2 and/or v3 schemes. Furthermore, in the case of arbitrary code execution, it must execute without causing the app to crash. The success of the attack depends on a combination of these factors, and if any of the required conditions are not satisfied, then the vulnerability cannot be triggered.

4.2 Android specifications loopholes

We analyze the official Android specifications on app signing and signature block structure for the v2 and v3 signature schemes to identify potential weaknesses. Next, we examine the implementation of app signing and signature verification to confirm these vulnerabilities and to assess how they might enable malicious manipulations. For our implementation analysis, we focus on the process of manually signing apps using `apksigner` [2]. Google offers several methods for developers to sign their apps: they can sign their apps manually using Android Studio or `apksigner`, or they can opt for Google to manage app signing through the Google Play Store. Google's official tool, `apksigner`, is included in the Android SDK Build Tools package [1]. According to Google, `apksigner` can sign and verify signatures for apps intended to run on all versions of the Android platform [2]. Our analysis of the signing and verification process reveals several loopholes:

① *The APK signing block is not signed.* According to the official Android documentation, the v2 and v3 signature schemes protect the integrity of the contents of the ZIP entries, the Central Directory, and the EoCD. However, the APK signing block remains largely unprotected with only the "signed data" section inside the v2 and v3 ID-value pairs being protected by digest(s). This limited protection allows an attacker to modify the APK signing block after the app was signed without breaking the app's integrity.

② *Verification ignores unknown ID-value pairs.* The signing block contains a sequence of app signatures in the form of ID-value pairs, prefixed and suffixed with fields that indicate the block size. The verification process ensures that the size fields in the APK signing block match and that the block conforms to the specified size. However, only known ID-value pairs are evaluated and unknown pairs are ignored. This oversight allows an attacker to embed arbitrary content into the APK signing block without compromising the required structure of the signing block.

③ *The APK signing block has variable size.* It is intended to accommodate not only digital signatures but also any additional metadata or information required

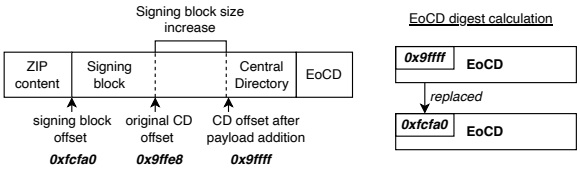

Fig. 2: The Central Directory offset treatment

during the signing process. The block size varies based on the number of signature schemes used and the size of the cryptographic keys and signatures. This flexible design provides significant space for embedding attack payloads.

④ *The Central Directory offset is never verified.* Since the APK signing block precedes the Central Directory section, any change in size of the APK signing block will also change the Central Directory's offset. When this occurs, the Central Directory offset that is stored in the EoCD must also be updated for the APK file to be parsed. If this offset in the EoCD is updated, the digest calculated over the modified EoCD will no longer match the original digest. In theory, this design should limit modifications that alter the APK signing block's size. However, to support legitimate modifications such as the addition of a new signature, during integrity verification Android treats the offset of the APK signing block as the offset of the ZIP Central Directory (Figure 2).

During verification, the ZIP components are extracted using the offset that correctly reflect the position of the Central Directory section after the signing block's size modification. This offset is then disregarded, and the offset of the APK signing block is instead used during digest calculation. In other words, the actual offset of the Central Directory is never verified, and the size of the signing block is altered without causing integrity verification to fail. As a result, an attacker can inject an unlimited payload into the APK signing block without invalidating the APK's signature.

4.3 Attack strategy

The design and implementation issues with the APK signing block can lead to various security vulnerabilities. In this work, we leverage the APK signing block as a vehicle for our attack, focusing specifically on the v2 and v3 signing schemes. We demonstrate how a malicious Android app can deliver and execute a payload without triggering any flags during the official app verification process.

Given an Android app signed with the v2 and/or v3 signing schemes, we can craft a malicious payload and embed it inside the app's signing block. We parse the APK and extract a copy of its signing block. We then locate the bytes representing the sequence of ID-value pairs in the signing block and inject our malicious payload at a target location. Afterwards, we overwrite the APK's original signing block with our modified version. *These modifications do not invalidate the app's original signature.*

In most cases, inserting the payload into the signing block increases its size. Since the signing block is designed to be extensible, we exploit this feature to conceal the payload in the signing block. To ensure that the modified APK can still be parsed accurately by Android tools, we make several adjustments:

① We replace the values in the two *size of block* fields with the new signing block size. ② We rewrite the Central Directory and the End of Central Directory (EoCD). ③ We update the Central Directory offset in the EoCD.

Payload. For our experiments, we test four different types of the payload.

- *EICAR Antivirus Test File.* Developed by the European Institute for Computer Antivirus Research[3] and the Computer Antivirus Research Organization, this file is used to test the response of antivirus programs without using a real malicious payload. This test allows us to assess whether antivirus programs can detect malicious content concealed in the APK signing block.
- *Text.* The text payload is a sequence of ASCII characters stored in a text file. For our experiments, we use a 7KB text file that contains 1,000 words of *lorem ipsum*. The text payload serves as an initial exploration on the types of content that can be embedded in the signing block.
- *Image.* Our image payload is a 829KB JPEG image file. Since image files can be used to conceal malicious content, this payload helps us to investigate the limits (if any) on the type of content that can be inserted into the APK signing block.
- *Code.* For our code payload, we implement a simple JavaScript code snippet that adds two integer numbers. Despite its brevity, this type of payload demonstrates that syntactically valid code can be concealed in the APK signing block and then executed after app installation.

Payload packaging. Android does not limit the size of the APK signing block, and although Android suggests that the size of the signing block should be a multiple of 4,096, it is not enforced. Consequently, there is a variable-size space for embedding an attack payload. The structure of the APK Signing Block is defined as a sequence of ID-value pairs that is prefixed and suffixed with fields containing the size of the signing block. This format potentially allows attackers to disguise a malicious payload as a legitimate ID-value pair. In our experiments, we explore the following approaches:

- *v2 and v3 ID-value pair.* We explore the use of IDs that correspond to a v2 signing scheme and a v3 signing scheme. For both v2 and v3 signature blocks, we create a new pair with the payload packaged as a value in a pair.
- *Verity ID.* We use an existing verity pair by inserting the payload into the value of the existing pair. To ensure that the size of the verity pair remains consistent, we adjust the remaining padding bytes by removing extra padding (if payload is smaller than the original padding) or increase the size of the pair (if the length of the payload exceeds the length of the verity pair).

[3] https://www.eicar.org/

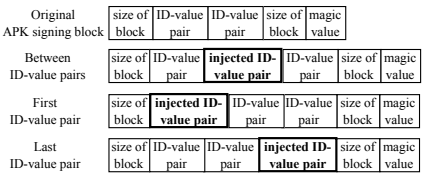

Fig. 3: Payload injection locations

– *Arbitrary ID.* We create a new ID-value pair with an arbitrary ID (0x716f43c0) and the payload embedded as value.
– *Raw byte form.* Our last method of adding the payload does not rely on the ID-value pair organization of the APK signing block. We directly insert the payload into the bytes of the APK signing block.

Payload injection location. We define three locations in the APK signing block where the packaged payload can be inserted as shown in Figure 3.

– *First ID-value pair.* The packaged payload is inserted in the APK signing block after the first *size of block* field and before the existing sequence of ID-value pairs. As a result, when the payload is packaged as an ID-value pair, this injected pair becomes the first pair in the sequence. If the payload is in the raw byte form, the payload is inserted directly after the first eight bytes of the APK signing block (the defined length of size field).
– *Between ID-value pairs.* To maintain consistency across our tests, this case is treated as the midpoint of the ID-value pairs sequence. The insertion index is calculated by dividing the length of the sequence by two, and we use integer division to obtain a whole number index without the fractional part.
– *Last ID-value pair.* The payload is inserted in the APK signing block after the last pair in the sequence, which is directly before the second *size of block* field and the magic value. This location is the last place in the signing block that the payload can be inserted without breaking the signing block's structure.

5 Attack implementation

To validate our proposed attack strategies, we implemented a custom app and tested the strategies on physical phones with the two latest versions of Android.

5.1 Testing App

We developed a custom Android app ($minSDK = 33$ and $targetSDK = 34$) in Android Studio Ladybug 2024.2.1, which was the latest version at the time of writing. Our app was created using the Kotlin programming language. The app's primary function is to process embedded payloads hidden within the APK

signing block. Specifically, the app locates its file path on the device, parses itself, and extracts its signing block. It retrieves the sequence of ID-value pairs and parses each pair to identify the injected payload. The app does not require user interaction and is designed as a single-page application with two user interface elements: a TextView for displaying the embedded text payload and an empty ImageView for displaying the embedded image payload. The app handles the payloads as follows:

- *EICAR.* We do not define a trigger for this payload form. Instead, we only check that the app can be installed and opened without crashing.
- *Text.* The text payload is converted from an array of bytes to a string, which is then displayed using a TextView element.
- *Image.* We decode the byte array that represents the image payload to a Bitmap and then set the Bitmap as the content of the ImageView element.
- *Code.* We use the RhinoScriptEngine[4] from the JSR223[5] application programming interface (API), a scripting interface for Java Virtual Machine (JVM) languages, to execute the code payload (represented as a string) in our app. To ensure that the payload extracted from the signing block is syntactically correct JavaScript, we remove all zero bytes from the byte array that represents the payload and convert the filtered array to a string. The string is passed to the script engine and executed, and the result is returned and set as the value of the TextView element.

After building an aligned release version of the app using Android Studio, we signed the app using `apksigner` (revision 35.0.0 of Android SDK Build Tools). For signing we used a 384-bit elliptic curve cryptography (ECC) key. We refer to this app as *base app*. Since our experiments examined both the v2 and v3 schemes, we created two versions of this base testing app: one app signed with the v2 scheme and the other app signed with the v3 scheme. These two base apps serve as a baseline in our experiments.

5.2 Implementation of attack scenarios

To demonstrate each of the attack strategies, we modified two original apps creating 120 apps with a payload. For each strategy, we extract the signing block from the original APK file, inject a payload and replace the original signing block with the modified version following Algorithm 1. We implemented this approach using the Python3 programming language.

To parse APK file and locate the signing block, we follow the approach typically used for parsing any ZIP archive. We start at the End of Central Directory (EoCD) to locate the starting offset of the Central Directory. We use the extracted offset to jump directly to the start of the Central Directory. We then backtrack to locate the magic "APK Sig Block 42" bytes. Since Android uses the signing block to support the v2 and v3 signing schemes—and indirectly the v4

[4] https://github.com/APISENSE/rhino-android
[5] https://www.openhab.org/docs/configuration/jsr223.html

Algorithm 1: Inserting the payload into the APK signing block

```
   /* locate APK signing block                                                    */
 1 read the bytes of the APK file;
 2 get the starting offset of the Central Directory;
 3 position = Central Directory offset - 16 bytes;
 4 magicBytes = read the bytes from position to (position + 16);
 5 if magicBytes ≠ "APK Sig Block 42" then
 6 |   return;                                          // signing block not found
 7 end
   /* extract APK signing block bytes                                             */
 8 position = Central Directory offset - 24 bytes;
 9 sizeOfBlockField2 = read the bytes from position to (position + 8 bytes);
10 position = −sizeOfBlockField2 + 8 bytes;
11 sizeofBlockField1 = read the bytes from position to (position + 8 bytes);
12 if sizeOfBlockField1 ≠ sizeOfBlockField2 then
13 |   return;                                       // size fields must be equal
14 end
15 position = position - 8 bytes;
16 originalSigningBlockOffset = position;
17 signingBlock = read bytes from position to (position + sizeOfBlockField2);
   /* add payload to a copy of the APK signing block                              */
18 parse signingBlock;
19 get ID-value pair sequence;
20 package payload;
21 if insertion location is first then
22 |   newSigningBlock = insert payload at the front of the pair sequence bytes;
23 end
24 if insertion location is between then
25 |   newSigningBlock = insert payload at midpoint of the pair sequence bytes;
26 end
27 if insertion location is last then
28 |   newSigningBlock = insert payload at the end of the pair sequence bytes;
29 end
   /* save Central Directory and EoCD offsets and bytes                           */
30 newSigningBlockSize = sizeOf(newSigningBlock);
31 rewrite newSigningBlock two size of fields with newSigningBlockSize;
32 cdAndEocd = extract Central Directory and EoCD bytes from APK;
33 eocdOffset = save offset of EOCD;
34 cdOffset = save offset of CD;
35 sizeDifference = sizeOf(newSigningBlock) - sizeOf(old signing block);
   /* replace original APK signing block with the modified copy                   */
36 open APK in binary read and write mode;
37 position = originalSigningBlockOffset;
38 write newSigningBlock to APK file at position;
39 write cdAndEocd to APK file at (position + newSigningBlockSize);
40 resize bytes of APK file to position;
41 newCdOffset = cdOffset + sizeDifference;
42 position = eocdOffset + sizeDifference + 16;
43 write newCdOffset to APK file at position;
```

scheme since it requires a complementary v2 or v3 signature—these magic bytes (and the signing block) are not present only when the APK is signed with the v1 scheme. If the signing block is indeed located before the Central Directory, these magic bytes are found in the 16 bytes directly before the starting offset of the Central Directory. Moving backwards from this offset, we parse the second *size of block* field that provides the starting offset of the APK signing block. We save this offset to use during the last steps of the APK modification.

We extract a copy of the signing block and locate the bytes between the two *size of block* fields. These bytes represent the sequence of ID-value pairs within

the signing block, and all of our insertions occur in this section. Locating the offsets for payload insertion as the *First pair* and *Last pair* is straightforward.

Finding the correct offset for the *Between pairs* insertion is more complicated, as it requires clearly delineating the ID-value pairs in the sequence. According to the Android specifications, the first 12 bytes of each pair represent the ID and length of the pair. We follow this approach to delineate the pairs and compute the offset that represents the midpoint between the pairs in the sequence.

After finding the insertion location, we package the payload as follows:

v2 and v3 ID. Upon their build, the original v2 and v3 base apps have two ID-value pairs, i.e., one ID-value pair containing original signer information and the verity ID-value pair (ID=0x42726577). For both apps, we create a new pair corresponding to v2 (ID=0x7109871a) and v3 (ID=0xf05368c0) signing schemes, respectively. We package the payload as a value in these new pairs. As a result, the modified app's signing block contains 3 pairs.

Verity ID. Verity ID-value pairs present a special case. As opposed to other cases where original pairs are not altered, we modify the value of the original verity ID-value pair (ID=0x42726577) already present in the original base apps to include the payload.

Consequently, if the payload injection location is first, the modified verity pair is moved to be the first pair in the sequence. If the payload location is last, the modified verity pair remains in its original position in the sequence.

Finally, if the payload location is between two valid pairs, we duplicate an existing first ID-value pair containing signing information. Since the signing information in the second pair is identical to the original first pair, it does not affect the signing profile of the app. We then insert the modified verity pair containing the payload between two legitimate ID-value pairs.

Arbitrary ID. This case is similar to the treatment of v2 and v3 ID pairs. We create a new pair with a non-existent (ID=0x716f43c0) and inject the payload as a value in this new pair.

Raw bytes. We follow the same approach for inserting this form of the payload in the signing block as we do for the ID-value pair packaging forms. Unlike these forms, however, we do not prefix the payload with the 12 bytes that specify pair ID and length.

Adjusting the size. Android does not limit the size of the signing block. Depending on its size, injecting the payload may increase the size of the APK signing block. If the payload (packaged as a verity pair) is smaller than the verity pair padding, it is padded with zero bytes to maintain the original verity pair size. However, if the payload is larger than the original verity pair length, the length is adjusted to reflect the new size of the pair. In cases where the signing block increases, we rewrite the values stored in the signing block's two *size of block* fields to avoid breaking the structure of the signing block.

At this point in the process, we have not modified the original APK. For our changes to take effect in the APK, we must overwrite the original signing block and adjust the APK file's metadata to ensure that the APK can still be parsed after our modification. We extract the bytes representing the original

Central Directory and EoCD and their corresponding offsets. We will use this information to realign the APK's structure after modification.

To assemble a modified APK file, we jump to the saved starting offset of the APK signing block and overwrite the original signing block with our modified copy of the signing block. To ensure that the APK's data aligns with its structure, we update the offset of the Central Directory in EoCD so the signing block is reachable when parsed. Finally, we append the APK's Central Directory and EoCD bytes directly after the end of the signing block.

As the final step in this process, we manually verify that the payload was correctly packaged and inserted in the right location. We use Python script to print out all of the pairs in the signing block to confirm that the payload was inserted at the correct index in the sequence and that the other pairs were not affected by the insertion.

5.3 Verification

We verify the modified apps using `apksigner`. We use the *--print-certs* option, which allows us to verify that the original APK's signing certificate remains in place and is correctly readable by `apksigner`. Although our embedded payloads are not supposed to be inside the signing block, most of them are innocuous. We test the detection tools' abilities to identify the embedded content in APK signing block on the example of the EICAR payload since it is intended to text antivirus software. We use VirusTotal's API v3 to scan the modified APKs.

5.4 Testing

For our evaluation of attack strategies, we use the following devices: Google Pixel 8 (Android 14, API 34) and Google Pixel 7 Pro (Android 13, API 33). Our testing apps are designed for $minSDK$=33 and $targetSDK$=34, which means the apps are compatible with devices running Android 13 (API level 33) or higher and are optimized to work with Android 14 (API level 34). This is the primary justification for the selection of evaluation devices.

Each modified app is installed on an Android device using `Android Debug Bridge (adb)`, a command-line tool that enables us to issue actions on devices [3]. If the installation is successful, we further verify that the payload is triggered successfully. Since the main functionality of the app is to handle and display the payload (or the result of executing the payload), we check that the payload is triggered by opening the app and manually verifying the content displayed by the app.

6 Experimental results

6.1 Verification

We verified all 120 apps with an embedded payload. We used `apksigner` to confirm that apps' original signatures were verified successfully on the two versions

Table 1: Experimental results

Signature scheme	Payload packaging	Payload	apksigner verification			Payload triggering					
						Android 13			Android 14		
			First	Between	Last	First	Between	Last	First	Between	Last
v2	v2 ID	EICAR	×	✓	✓	○	●	●	○	●	●
		Text	×	✓	✓	○	●	●	○	●	●
		Image	×	✓	✓	○	●	●	○	●	●
		Code	×	✓	✓	○	●	●	○	●	●
	v3 ID	EICAR	×	×	×	○	○	○	○	○	○
		Text	×	×	×	○	○	○	○	○	○
		Image	×	×	×	○	○	○	○	○	○
		Code	×	×	×	○	○	○	○	○	○
	Arbitrary ID	EICAR	✓	✓	✓	●	●	●	●	●	●
		Text	✓	✓	✓	●	●	●	●	●	●
		Image	✓	✓	✓	●	●	●	●	●	●
		Code	✓	✓	✓	●	●	●	●	●	●
	Verity ID	EICAR	✓	✓	✓	●	●	●	●	●	●
		Text	✓	✓	✓	●	●	●	●	●	●
		Image	✓	✓	✓	●	●	●	●	●	●
		Code	✓	✓	✓	●	●	●	●	●	●
	Raw byte form	EICAR	×	✓	✓	○	●	●	○	●	●
		Text	×	✓	✓	○	◐	◐	○	◐	◐
		Image	×	✓	✓	○	◐	◐	○	◐	◐
		Code	×	✓	✓	○	◐	◐	○	◐	◐
v3	v2 ID	EICAR	✓	✓	✓	●	●	●	●	●	●
		Text	✓	✓	✓	●	●	●	●	●	●
		Image	✓	✓	✓	●	●	●	●	●	●
		Code	✓	✓	✓	●	●	●	●	●	●
	v3 ID	EICAR	×	✓	✓	○	●	●	○	●	●
		Text	×	✓	✓	○	●	●	○	●	●
		Image	×	✓	✓	○	●	●	○	●	●
		Code	×	✓	✓	○	●	●	○	●	●
	Arbitrary ID	EICAR	✓	✓	✓	●	●	●	●	●	●
		Text	✓	✓	✓	●	●	●	●	●	●
		Image	✓	✓	✓	●	●	●	●	●	●
		Code	✓	✓	✓	●	●	●	●	●	●
	Verity ID	EICAR	✓	✓	✓	●	●	●	●	●	●
		Text	✓	✓	✓	●	●	●	●	●	●
		Image	✓	✓	✓	●	●	●	●	●	●
		Code	✓	✓	✓	●	●	●	●	●	●
	Raw byte form	EICAR	×	✓	✓	○	●	●	○	●	●
		Text	×	✓	✓	○	◐	◐	○	◐	◐
		Image	×	✓	✓	○	◐	◐	○	◐	◐
		Code	×	✓	✓	○	◐	◐	○	◐	◐

✓- APK is verified by apksigner, × - APK is not verified by apksigner
● - Installs and triggers the payload, ◐ - Installs but crashes, ○ - Does not install

of the Android platform supported by our test APK (API 33 and 34). Table 1 presents the results of verifying the test apps. The majority of our test apps were successfully verified by `apksigner`. In these cases, the embedded payload did not affect the app's signing profile.

Arbitrary ID. Regardless of the payload injection location, all apps in which the payload was packaged as an ID-value pair using an arbitrary ID were successfully verified. This outcome is not surprising, as Android's policy is to ignore ID-value pairs with unknown IDs.

Verity pairs. Similarly, test apps with the payload embedded in the verity pair were all verified by `apksigner`. Our manual inspection of `apksigner`'s source code revealed that the contents of the verity block are not checked during verification. As a result, the embedded payload remains undetected.

v2 and v3 pairs. The results of the tests where we packaged the payload as a pair with a v2 ID or v3 ID varied. Apps with EICAR, text, or code payloads inserted as the first pair in the sequence failed verification due to a malformed list of signers. Similarly, the image payload failed verification, but this was due

to a *java.lang.IllegalArgumentException: Negative length* exception. The root cause of these failures is consistent: the format of the pair did not adhere to Android's defined specifications, leading to unexpected behaviour during parsing by `apksigner`. The difference between these errors lies in their handling: the malformed signer error was anticipated and appropriately caught by `apksigner`, while the conditions triggering the Java exception were not.

When the payloads were inserted into the APK signing block between two valid ID-value pairs or as the last pair in the sequence (v2 payload in originally v2 apps, v3 payload in originally v3 apps), `apksigner` verification was successful. This is due to the verification logic of `apksigner`, which verifies the first known ID-value pair that it expects according to the app's *maxSDK*. Consequently, when the pair with the payload was inserted behind a valid ID-value pair (i.e., between and last insertion locations), `apksigner` verified the first expected pair and did not proceed to the pair that contained the payload resulting in successful app verification.

If the first encountered pair is an older signing scheme (e.g., v2) compared to what is expected, `apksigner` proceeds until it finds a pair corresponding an expected scheme. In all cases, where the payload, packaged as a v3 pair, was inserted into an APK originally signed with the v2 scheme, all tests failed. For APKs with a $maxSDK \geq API\ 33$ (Android 13), `apksigner` expected the v3.1 and v3 signature schemes. As a result, `apksigner` proceeded to a v3 ID-value pair to verify. Since the format of the v3 pair's value (which contained the payload) did not match the expected structure, verification failed. A similar issue occurs when the payload, packaged as a v2 pair, is inserted as the first pair into an APK already signed with the v2 scheme. In this case, since no v3 signing pair is present, `apksigner` proceeds to verify the first v2 pair it encounters. However, this pair contains the embedded payload, which does not conform to the expected format and leads to verification failure. Likewise, a v3 APK with the payload embedded in the first v3 pair also fails verification for the same reason.

Raw byte form. The test apps with the payload added to the APK signing block in its raw byte form at the start of the sequence failed verification. Our investigation revealed that `apksigner` mistakenly interpreted the first 12 bytes of the raw payload as the ID and length fields of a valid ID-value pair. This misinterpretation caused `apksigner` to treat a larger chunk of bytes—extending beyond the boundaries of our embedded payload—as the value field. As a result, the legitimate ID-value pairs following our payload and containing the signing information were incorrectly interpreted. This effectively rendered the APKs' signing data inaccessible to `apksigner`, leading to verification failure.

Overall, our results show that there are several viable payload packaging and insertion location combinations that an attacker can reliably exploit.

VirusTotal analysis. To establish a baseline, we analyzed the EICAR text file[6] using VirusTotal. Out of 76 detection engines included in the VirusTotal platform, 66 flagged this file as malicious. We then analyzed the 30 modified

[6] https://www.eicar.org/download-anti-malware-testfile/

APK files with the EICAR payload embedded in their signing block. For 29 of the modified APK files, the detection rate dropped significantly to 5/76. The same five engines consistently detected the EICAR payload (Alibaba, Ikarus, Fortinet, Google, and SymantecMobileInsight) while the majority of engines categorized the files as undetected. A small number of engines failed or timed out during analysis. The one exception was the case in which the EICAR payload was packaged as a v3 ID-value pair and inserted between other pairs in the testing app that was signed with the v3 scheme. Only four engines (Alibaba, Ikarus, Fortinet, and SymantecMobileInsight) detected the EICAR payload; the Google engine reported a failure.

6.2 Testing

After verification, we confirmed that ① the modified apps can be installed on the physical devices, and ② the embedded payloads can be successfully accessed or triggered.

The results of our analysis are shown in Table 1. Overall, the installation successes and failures align with the verification results with a few exceptions. *Injecting the payload as part of a verity ID-value pair or a pair with an arbitrary ID consistently results in successful verification, installation, and reliable triggering of the payload.*

Packaging the payload as a v2 or v3 pair is less consistent. The v2 APKs with the payload packaged as a v3 pair were not verified and, as a result, did not install on any of the devices. Similarly, the v2 and v3 APKs with the payload packaged as the first v2 and v3 pairs, respectively, did not install. The rest of the APKs were verified and successfully installed on the devices, allowing access to the embedded payload.

The apps whose signing blocks contained the payloads in the raw byte form behaved differently if the payload was inserted between or after valid pairs. Although the apps installed on the device successfully, they crashed when they were opened. Further manual investigation showed that this behaviour was due to an out-of-bounds error that caused an incorrect interpretation of the payload bytes as a length-prefixed field. When the text, image, and code payloads (raw bytes) were inserted between pairs or as the last pair in the sequence, their initial bytes were interpreted as the ID and length fields of a pair. In most cases, these initial bytes did not correspond to the actual payload length (as they were not intended for this purpose). As a result, parsing exceeded the number of available bytes, causing an out-of-bounds exception. The EICAR payload did not cause this behaviour due to its size. In either case, crafting a payload to match the expected length can potentially resolve these errors and allow for more consistent parsing of payloads.

6.3 App distribution through app market

The final step in our analysis is to verify whether our crafted app with an embedded payload can be successfully distributed through an app distribution plat-

Table 2: Results of malicious app analysis (case study)

Strategy	Verified by apksigner	VirusTotal analysis		Installation status	
		Detection rate	Google engine detection	Android 13	Android 14
Baseline	✓	27/75	Yes	●	●
Direct injection	✓	5/76	Yes	●	●
Compress	✓	6/76	Yes	●	●
Encode	✓	0/76	No	●	●
Reverse	✓	0/76	No	●	●
Split ($n = 2$)	✓	3/76	Yes	●	●
Split ($n = 4$)	✓	0/76	No	●	●

✓- APK is verified by apksigner, ● - APK installs on the device

form. For this experiment, we designed an app that allows users to increment or decrement a counter. Since app markets do not accept applications without clear functionality, we could not use our base app. We signed this app with a v2 signature and embedded our code payload as an ID-value pair with the arbitrary ID that we used in our previous experiments. We inserted the packaged payload between the v2 pair containing the legitimate signing information and the verity pair. We then uploaded this app to APKPure and Aptoide, which are alternative app markets. Since the Google Play Store requires developers to upload their apps in the AAB format, we do not include it in our analysis.

APKPure and Aptoide verified the app without issues and made it publicly available to users. We downloaded the app's file from each market to inspect its signing block. In both cases, the code payload remained intact, confirming that neither market had stripped our signature during the upload process. The APK files from APKPure and Aptoide were successfully installed on both devices, and once opened, each app displayed the results of the code execution.

7 Case study

Until now, our evaluation of attack strategies was confined to innocuous payloads. As a proof-of-concept, we modified the test app with the v3 signing scheme to include a malicious payload in the form of an APK file obtained from the MalwareBazaar repository.[7] This executable was a Trojan malware that targets Android mobile banking apps by disguising itself as a legitimate app.[8] We packaged the malware as an ID-value pair with the same arbitrary ID that we used in our experiments. Similar to our previous experiments, we inserted the malicious payload between the original v3 ID-value pair and the verity pair of our app.

To establish a baseline, we analyzed the sample using VirusTotal. Out of 75 engines, 27 (including Google's detection engine) classified it as malicious. In contrast, the detection rate for our malicious app was significantly lower. Only 5 out of 76 engines, including Google, identified it as malware.

Since the Google engine detected our app as malicious, it is unlikely that such an app would be allowed to be distributed through the Google Play platform. To evade Google detection and obscure the payload, we explored four strategies:

[7] https://bazaar.abuse.ch/

[8] SHA256: b8ea74902684dcced62a5ca2c1d6932659decfefcbdb2615bfe5899e05eb1451

– *Compress.* We compressed the malicious payload to the ZIP format before its insertion into the signing block. The detection rate increased compared to the previous test as 6 out of 76 engines detected its malicious content.
– *Encode.* We encoded the payload as the Base64 format before its insertion. This strategy was successful, and none of the engines detected the malware.
– *Reverse.* We reversed the order of the payload's byte sequence. As with the encoding strategy, none of the 76 engines detected malicious payload in the manipulated APK.
– *Split.* We divided the payload's byte sequence into n equal subsequences and packaged each as a separate arbitrary ID-value pair. For our first test, we split the payload into 2 subsequences creating 2 pairs. The pair containing the first half of the payload was inserted between the v3 signing block and the verity pair while the pair containing the second half was inserted as the last pair in the sequence. Only 3 engines, including Google, detected the malicious payload. When we divided the payload into 4 subsequences and distributed them throughout the signing block, none of the engines, including Google, detected the malicious content.

For all apps generated in this case study, we used `apksigner` to verify the signatures of the modified malicious APKs. As `apksigner` is not designed to detect malicious content, it is unsurprising that all the APKs produced during our tests were verified successfully. We also attempted to install each of the modified APKs on the two Google devices. Like the EICAR payload experiments, we did not define a trigger and only checked that the APK could be installed and would not crash when opened. All apps successfully installed and could be opened without crashing.

8 Discussion

For Android development team:
① *APK signing block content verification.* The lack of block verification is a key factor enabling our attacks. This issue could be addressed by implementing a digest over the entire APK (as the v4 scheme does), the entire APK signing block, or the contents of the signing block. Each approach, however, has its limitations. For instance, the v4 signature file is stored alongside the APK file, which poses challenges for APK distribution. Protecting the integrity of the entire APK signing block could limit the extensibility of the signing mechanism. Calculating a digest over the ID-pair sequence still leaves room for payload injection. However, when combined with the recommendations presented below, this approach can reduce the likelihood of signing block exploitation. ② *Address the mismatch between signing and validation functionalities.* Updates made to the signing block content during the signing process should be reflected in corresponding checks during the verification process. For example, the integrity and expected format of the verity pair should be thoroughly verified. ③ *Comprehensive verification.* All content within the APK signing block should be verified, and unverified content

should be rejected. This would eliminate scenarios where ID-value pairs with unrecognized IDs are ignored and allowed to remain in the block. ④ *Unambiguous official specifications.* The official Android documentation assumes the presence of only one v2 or v3 signing pair, leading to the verification of the first encountered pair. This oversight effectively allows an attacker to exploit additional pairs for malicious purposes. ⑤ *ZIP offset verification.* The digest calculation should include the actual CD offset. This would prevent modifications to the APK signing block size after the app is signed.

For app distribution market operators:

Strengthen app vetting. Market operators could further enhance its vetting process for apps submitted to the market, particularly focusing on the APK signing block. The vetting process could ensure compliance with official specification, employ deeper analysis of embedded content, and leverage advanced malware detection techniques.

Recommendations for the Android development team:

- *APK signing block content verification.* The lack of signing block verification is a key factor in enabling our attacks. This issue could be addressed by implementing a digest over the entire APK (as the v4 scheme does). However, because the v4 signature file is a separate file that is stored alongside the APK, this approach poses challenges for APK distribution. Another approach would be to implement a digest over the APK signing block, which may ultimately limit the extensibility of the signing mechanism. Additionally, the more conservative approach of only implementing a digest over the ID-value pair sequence still leaves room for payload injection. While each of these three proposed approaches have their drawbacks, they can reduce the likelihood of signing block exploitation when combined with the recommendations presented below.
- *Address the mismatch between signing and validation functionalities.* Updates made to the signing block content during the signing process should be reflected in corresponding checks during the verification process. For example, the integrity and expected format of the verity pair should be verified.
- *Comprehensive verification.* All content within the APK signing block should be verified, and unverified content should be rejected. This would eliminate scenarios where ID-value pairs with unrecognized IDs are ignored and allowed to remain in the block.
- *Unambiguous official specifications.* The official Android documentation assumes the presence of only one v2 or v3 signing pair, leading to the verification of the first encountered pair. This oversight effectively allows an attacker to exploit additional pairs for malicious purposes.
- *ZIP offset verification.* The digest calculation should include the actual Central Directory offset. Using this offset would require a significant redesign of the verification process especially with respect to the addition of new signing material. Nevertheless, it is a necessary step towards preventing modifications to the APK signing block after the app is signed.

Recommendations for app distribution market operators:

– *Strengthen app vetting.* Market operators could further enhance their vetting process for apps submitted to their markets, particularly focusing on the APK signing block. The vetting process could ensure compliance with official specification, employ deeper analysis of embedded content, and leverage advanced malware detection techniques.

Limitations. Our work has two major limitations. *First,* we assume that the APK signing block's internal structure and position within the APK file adheres to Android documentation. Our method for parsing the APK file to extract the signing block follows the Android-defined approach. However, we are aware that some third-party app markets (e.g., F-Droid) use different APK signing block parsing methods. In such scenarios, our embedded payload may be detectable, and consequently, our attacks may not succeed in those markets. *Second,* we do not include APKs with mixed signature schemes in our analysis (i.e., v1 and v2, v1 and v3, v2 and v3). We do not address the v1 signing scheme due to the differences in signing implementation between the v1 and v2 schemes. In other words, modifying an APK signed with the v1 scheme changes its content and makes the manipulation detectable by `apksigner`.

9 Conclusion

App signing is a critical protection mechanism that ensures the integrity and authenticity of mobile apps. Our approach exploits logical weaknesses in the design and implementation of the verification process to embed arbitrary content in the APK signing block. This attack is feasible because validation tools operate under the assumption that all components present are created according to Google's guidelines for generating and signing APKs. This assumption creates significant blind spots in Android tools, leaving ample space within the APK file for embedding attack payloads.

References

1. Android: Android sdk platform tools. https://developer.android.com/tools#tools-build (2023)
2. Android: apksigner. https://developer.android.com/tools/apksigner (2023)
3. Android: Android debug bridge (adb). https://developer.android.com/tools/adb (2024)
4. Android: Apk signature scheme v2. https://source.android.com/docs/security/features/apksigning/v2 (2024)
5. Android: Apk signature scheme v4. https://source.android.com/docs/security/features/apksigning/v4 (2024)
6. Android: Application signing. https://source.android.com/docs/security/features/apksigning (2024)
7. Du, S., Zhu, P., Hua, J., Qian, Z., Zhang, Z., Chen, X., Zhong, S.: An empirical analysis of hazardous uses of Android shared storage. IEEE Transactions on Dependable and Secure Computing **18**(1), 340–355 (2021)

8. Fahl, S., Dechand, S., Perl, H., Fischer, F., Smrcek, J., Smith, M.: Hey, NSA: Stay away from my market! future proofing app markets against powerful attackers. In: 2014 ACM CCS. pp. 1143–1155. ACM (2014)
9. Jung, J.H., Kim, J.Y., Lee, H.C., Yi, J.H.: Repackaging attack on Android banking applications and its countermeasures. Wireless Personal Communications **73**, 1421–1437 (2013)
10. Kar, A., Stakhanova, N.: Exploiting Android browser. In: CANS. pp. 162–185. Springer-Verlag (2023)
11. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Veen, V.v.d., Platzer, C.: ANDRUBIS – 1,000,000 Apps later: A view on current Android malware behaviors. In: BADGERS. pp. 3–17 (2014)
12. NVD: Cve-2013-4787. https://nvd.nist.gov/vuln/detail/CVE-2013-4787 (2013)
13. NVD: Cve-2017-13156. https://nvd.nist.gov/vuln/detail/CVE-2017-13156 (2019)
14. Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., Vigna, G.: Execute this! analyzing unsafe and malicious dynamic code loading in Android applications. In: NDSS. pp. 1–16 (2014)
15. Ruggia, A., Possemato, A., Merlo, A., Nisi, D., Aonzo, S.: Android, notify me when it is time to go phishing. In: IEEE EuroS&P. pp. 1–17 (2023)
16. Vidas, T., Christin, N.: Sweetening Android lemon markets: measuring and combating malware in application marketplaces. In: ACM CODASPY. pp. 197–208. ACM (2013)
17. Vidas, T., Votipka, D., Christin, N.: All your droid are belong to us: A survey of current Android attacks. In: WOOT. pp. 1–10. USENIX Association (2011)
18. Wang, H., Liu, H., Xiao, X., Meng, G., Guo, Y.: Characterizing Android App Signing Issues. In: IEEE/ACM ASE. pp. 280–292. IEEE Press (2019)
19. Xu, H., Yao, M., Zhang, R., Dawoud, M.M., Park, J., Saltaformaggio, B.: DVa: Extracting victims and abuse vectors from Android accessibility malware. In: USENIX. pp. 701–718. USENIX Association (2024)
20. Zhang, X., Ying, K., Aafer, Y., Qiu, Z., Du, W.: Life after app uninstallation: Are the data still alive? data residue attacks on android. In: NDSS. pp. 1–15 (2016)
21. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: IEEE S&P. pp. 95–109 (2012)