Manifest Problems: Analyzing Code Transparency for Android Application Bundles

Florian Draschbacher
Graz University of Technology
Secure Information Technology Center Austria
Graz, Austria
florian.draschbacher@iaik.tugraz.at

Lukas Maar
Graz University of Technology
Graz, Austria
lukas.maar@iaik.tugraz.at

***Abstract*—In 2018, Google introduced a new app distribution format called AAB (Android Application Bundle), which replaced APK (Android Package) as the required format for all new app submissions to Google Play in 2021. Apps are still delivered to end users as APK files, but they are now generated and signed on the app store operator's infrastructure. Most crucially, this change requires developers to hand over their APK signing key to the app store operator, enabling them to arbitrarily manipulate apps prior to delivery to end users. To address this, Google has introduced the Code Transparency scheme to verify the integrity of APKs generated from AAB files. However, due to the lack of independent studies, the exact security properties of Code Transparency remain unclear.**

In this paper, we present the first comprehensive analysis of the security of Code Transparency and the AAB format. We thoroughly investigate the design and implementation of the Code Transparency scheme, discussing in detail the technical possibilities attackers have for manipulating apps that use it. Additionally, we conduct a large-scale study on AAB and Code Transparency in practice. To this end, we evaluate the prevalence of both technologies among 3.5 million real-world apps, analyze their susceptibility to our attacks, and carry out a case study that demonstrates the practical security implications of attacks on Code Transparency.

Our analyses indicate that Code Transparency suffers from severe design and implementation flaws that allow app store operators to execute code in the context of any app without disturbing its Code Transparency signature.

***Index Terms*—Android Security, Code Transparency, Software Integrity**

1. Introduction

The adaptability to a wide range of hardware platforms has been a significant advantage of Android in the competitive mobile OS market. However, it has also led to a fragmented device landscape and a wide range of security and usability issues that continue to plague the Android ecosystem. To address these issues, Google has made considerable efforts, including initiatives such as *Project Treble* [1], which simplified adapting OS updates to devices, *Project Mainline* [2] to modularize the OS into components updatable through Google Play, or *Android Jetpack* [3] which provides developers with app building blocks that reliably work across Android versions.

Another recent change in this series of initiatives was the introduction of the Android Application Bundle (AAB) format for packaging applications for submission to app stores in 2018 [4]. Until that point, developers usually shipped each application to app stores as a single universal APK file containing resources for all supported device configurations. App stores would then serve these exact files to end-user devices, where the resources designed for other device configurations would waste storage space. Although the new AAB submission format contains compiled code and resources for all possible device configurations as well, it only serves as an intermediary format. The app store operator uses it to *"generate and serve optimized APKs for each device configuration"* [5], which saves storage space and network bandwidth.

Since the introduction of the AAB format, Google has put a lot of effort into establishing it as the new distribution standard for the entire Android app industry. In 2020, the format was made mandatory for any new app submissions to Google Play [6]. Since May 2023, all apps or updates that support Android TV need to use the AAB format [7]. The format has also been adopted by all major third-party Android app stores, including Samsung's Galaxy Store, Huawei AppGallery, and Amazon Appstore.

However, besides improving the user experience, the AAB format has considerable consequences for the security architecture of the Android OS. The Android Platform Security Model [8] considers the application developer one of the three stakeholders that need to agree on sensitive operations being carried out on the device. Specifically, it is the application (on behalf of the application developer) that decides on the access to login credentials or other valuable user data stored in its private data folder. Since the application encodes the developer's intentions regarding sensitive operations affecting the app and its data, it must remain unmodified through the entire supply chain between the developer and the end user. Any code injected into an application somewhere along the supply chain would later be executed in the process context of the application itself,

where it is able to alter its security policies.

The security consequences of AAB became particularly problematic when Google in 2020 made it mandatory for any new app submission to Google Play [6]. After the developer community voiced its concerns regarding that change [9], Google in 2021 introduced the optional Code Transparency (CT) scheme for AAB files. It is intended to provide cryptographically proofable integrity guarantees for key parts of an Android application. Despite the security-critical role of CT for the AAB format, the reliability of its integrity guarantees has never been examined. Furthermore, it remains entirely unclear how and if the scheme is used in practice, whether it is deployed correctly, and whether its design and implementation align with the reality of modern mobile applications.

In this paper, we present the first comprehensive analysis of the security of CT for the AAB format. We establish attack scenarios against CT based on Google's official documentation, related supply-chain hardening techniques and the Android Platform Security Model. From these attack scenarios, we identify three design flaws in the CT scheme's design, as well as three flaws in the *bundletool* reference implementation. As a key observation in our analysis, we note that CT's exclusive focus on Dalvik Executable (DEX) and Shared Object (SO) files entirely neglects the powerful role of resource files such as the app manifest on the Android platform.

Subsequently, we show how an attacker can exploit the identified flaws. We consider three attacker models of varying power that are in line with Google's Code Transparency threat model. In these scenarios, a supply chain attacker compromises the APK signing key to sign manipulated app builds. The CT is supposed to help identify these modifications. Yet, under all attacker models, the attacks we identify allow gaining code execution in the context of an application without invalidating its CT.

Furthermore, we investigate the use of AAB and CT in practice. To this end, we carry out automated analyses on more than 3.5 million Android applications from Google Play and Huawei AppGallery. Our results show that despite the serious security implications of the AAB format it was designed to protect against, CT is almost non-existent (0.0014 % of apps submitted as AAB) in real-world applications. We also evaluate the eligibility for CT of the most popular apps from Google Play. We find that if they used CT, more than 50 % of them would be susceptible to code execution attacks even under the least powerful attacker model.

Overall, our results show that CT in its current form is mostly ineffective and virtually unused. This leaves developers unprotected against the security repercussions of the AAB format, which they are increasingly pressed to use. We therefore discuss possible improvements to CT and suggest app distribution solutions that offer the same advantages as AAB without compromising on security.

Contributions: To summarize, our key contributions are:

- We thoroughly analyze the design and implementation of Code Transparency for Android Application Bundles, identifying multiple flaws.
- We demonstrate attacks that exploit the identified flaws to gain code execution or data access in the context of target applications without invalidating their CT.
- We conduct the first large-scale survey of CT in practice, analyzing applications from Google Play and Huawei AppGallery. We evaluate their use of AAB and CT and their susceptibility to the vulnerabilities we identified in CT's integrity guarantees.

Disclosure: We disclosed our findings to Google and are currently discussing how to mitigate the attacks we identified, ultimately improving Android security. One of the reported issues already received a CVE[1] and was fixed in Android 14.

Outline: Section 2 provides background on the Android platform. Section 3 introduces the Android Application Bundle format and Code Transparency scheme. In Section 4, we discuss our attack scenarios, which we use in our security analysis in Section 5. Section 6 presents attacks based on the identified flaws, before Section 7 evaluates Code Transparency in practice. We discuss our findings in Section 8, elaborate on related work in Section 9, and conclude our paper in Section 10.

2. Background

2.1. Android and its Security Architecture

Android is the most popular operating system for mobile devices, most notably for smartphones. To facilitate adoption by handset manufacturers, the platform is based on a Linux kernel and developed as an open-source project led by Google. Android supports user-installed software, which is executed inside a sandbox that enforces strict isolation between applications. Most users download apps from the platform's official Google Play app store, although the platform (through a user setting) allows installing applications from arbitrary sources. This possibility has led to the emergence of third-party app stores.

2.1.1. Android Platform Security Model. In 2018, key security architects in the Android team published the Android Platform Security Model [8], which formalizes the core principles of Android's security architecture. The platform uses a multi-party consent model, where operations that affect apps must be agreed upon by the end user, the operating system (OS), and the application developer (through policies expressed in the app code). This consent model spans the traditional notion of subjects (e.g., users and processes) accessing objects (e.g., files and network sockets). The party that creates an object is implicitly granted control over it. For example, the system automatically creates a private data folder for every app. The user's consent to this action is given through triggering the app installation. The OS ensures that no other application may access this folder without consent from the three main parties.

1. CVE-2023-21387

2.1.2. Permission System. The only way for an app to utilize re- sources shared with the rest of the system is through well-defined APIs under tight control of the OS kernel and system framework. For accessing most of these APIs, apps need to obtain the corre- sponding permission from the OS. Any permission an app may request at runtime needs to be statically declared to the OS in the app's application manifest (see Section 2.2). While the user implicitly grants an app's declared permissions during installation, some particularly dangerous permissions require explicit user consent during runtime. The permissions known to the Android OS are grouped into different protection levels. Only a subset of these levels is accessible to user-installed applications.

2.1.3. Privileged Applications. The Android security architecture assumes some applications preinstalled on the device are more trustworthy than user-installed applications. They, therefore, are allowed to request more privileged permissions belonging to the *privileged* protection level. We refer to these applications as *privileged applications* in the following. Crucially, even if these apps hold a privileged position on the device, they may *not* access any other (less-privileged) app's process space, influence its execution, or access its private data directory.

2.2. Android Package Format

Android applications run in the Android-specific ART Java runtime, which requires code to be compiled to the special Dalvik bytecode format. Performance-critical functionality can be provided as native machine code in ELF shared object files. Applications need to be packaged into an Android Application Package (APK) file before they may be installed on a device. These files are signed ZIP archives whose contents follow a specific structure. The main elements are:

- *DEX Files*: Dalvik Executable files that contain *managed code*, i.e. the Dalvik bytecode encoding the app's functionality. Any file named `classes.dex` or `classes{`n`}.dex` (for any $n > 1$) in the APK root is loaded into the app's classpath.
- *Shared Libraries*: ELF binaries that contain *native code* usually compiled from C or C++. Since they contain machine code, they require adaptation to every Instruction Set Architecture (ISA). Native libraries typically reside in the `libs` folder in the APK, from where they must be loaded via managed code.
- *Resources*: UI layout definitions, strings for localizations, and more that are compressed during compilation are stored in the `res` folder and indexed in `resources.arsc` in the APK's root. Resources carry a configuration qualifier, such as a language, screen density, or OS version. It is, e.g., possible to mark an image as high-dpi, so the Android framework loads it only on devices with a high pixel density.
- *Assets*: Arbitrary files that are not modified during compilation. These are stored in the `assets` folder in the APK.

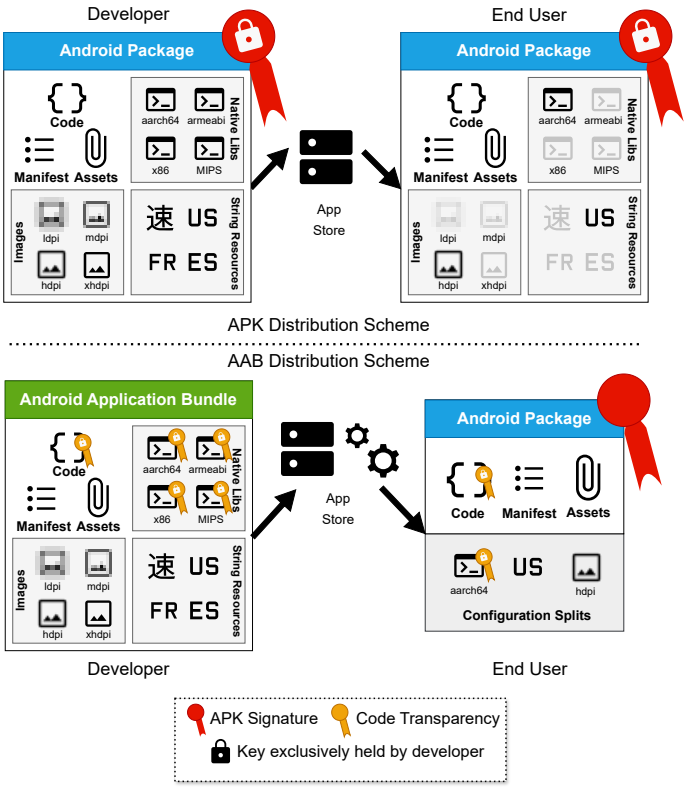

Figure 1: Comparison between old APK and new AAB distribution schemes. Unused resources and native libraries wasted storage space in the old APK distribution scheme. In the new AAB scheme, the app store serves optimized APKs for every device configuration. Note how the optional Code Transparency covers DEX files and native libraries, while the APK signature covers the entire APK file. APKs generated from AAB files are signed by the app store, which is now in possession of the app signing key. Only the Code Transparency key is exclusively held by the app developer.

- *Application Manifest*: `AndroidManifest.xml` in the APK root effectively serves as the contract between the app and the OS. It declares the app components (e.g., UI activities, broadcast receivers, content providers, etc) that are exposed to the rest of the system, as well as the system functionality (e.g., permissions) that the app wishes to use.

2.3. Universal APKs, Multi APKs, Split APKs

APK files that contain resources and native libraries for a variety of device configurations are called *Universal APKs*. Alternatively, developers may produce multiple APK builds of their app targeting different device configurations, called *Multi APKs*. Android devices (running Android 5.0 or newer) allow a single app to be installed from a collection of APK files, called *Split APKs*.

2.4. APK Signature

Before an APK file can be installed on an Android device, it needs to be signed. The signature covers all contents of the APK file and is only checked during installation. It serves as an integrity guarantee for an APK's delivery (no tampering between its creation and installation) and an app's installation (updates must come from the original app author). Since APK signing certificates are usually self-signed, the APK signature does not provide any means for authenticating the developer of an app. By ensuring that app updates are signed with the same certificate as the original installation, the signature protects the app's private data.

For the first decade of the Android OS's existence, developers typically were responsible for packaging and signing APK files for their apps. They then submitted the signed APK files to app stores, which would relay them to end users. Since they would retain exclusive control over the APK signing key, developers could use hardware-assisted app attestation [10] at runtime to ascertain that the executed APK file had not been manipulated.

3. Android Application Bundles and Code Transparency

With the Android Application Bundle (AAB) format, Google introduced a distinction between the publishing and delivery formats of Android applications. While the Android OS still only accepts app installations from APK files, developers now submit their apps to distributors (such as app stores) in the new AAB format. The distributor then generates APK files optimized for end-user devices from the submitted AAB file. The difference between the old APK and the new AAB distribution scheme is illustrated in Figure 1.

3.1. AAB File Structure

Like the APK file format, AAB is based on ZIP archives. However, an AAB file retains more of the high-level structure from the app project, facilitating the generation of optimized APK files later on. More specifically, data inside an AAB file is organized in modules. Every app is required to have a base module shipped with every installation. Further app functionality may be clustered into additional modules that can be installed on demand later. Every module inside the AAB is organized in its own top-level folder. Each module folder contains a file structure that is very similar to that inside an APK file. A noteworthy difference is that all DEX files that are to be copied into the root folder of generated APK files are stored in a subfolder named `dex` inside each module folder.

3.2. Bundletool

Google made all tooling required for building and processing AAB files available under an open-source license in the *bundletool* project[2] to facilitate adoption by third-party app stores. *bundletool* is used by Google Play and third-party app stores, as well as by the Android Gradle Plugin that compiles app projects in the official Android Studio IDE [11]. Additionally, it may be invoked directly through a command-line interface. The tool provides functionality for creating AAB files from a set of individual app modules or for generating a set of optimized APK files from an AAB file.

3.3. APK Generation

When building APKs from an AAB file, *bundletool* will generate a set of *Split* APKs. These include a base APK for the base module and several configuration splits containing different variants of resources and native libraries. Finally, Split APK files will also be generated for any feature modules and their configuration splits. The possible dimensions for generating configuration splits are language, screen density, and processor Application Binary Interface (ABI)[3]. For example, for a complete installation of an app, it may be necessary to install the base split, a configuration split for high-density screens, one for the English language, and another for native libraries in the `arm64-v8a` ABI. Typically, app stores automatically download all Split APK files suitable for the user's device configuration. At the developer's discretion, feature modules may either be distributed as add-ons the base module dynamically downloads on demand or already as part of the initial installation.

The AAB format was designed to also support legacy devices incompatible with Split APKs. For apps targeting OS versions prior to Android 5.0, *bundletool* therefore also generates Multi APKs for all possible combinations of APK splits.

3.4. Code Transparency

This work refers to Google's definition of Code Transparency (CT): *"CT is an optional code signing and verification mechanism for apps published with the Android App Bundle"* [12]. This mechanism is only implemented in external tooling. There is no infrastructure in the Android OS itself for verifying the CT of APK files.

A developer can add CT to an application through *bundletool*'s command line interface or through its Android Gradle Plugin integration. This process injects a JSON Web Token (JWT) file into the app's AAB. This JWT file *"contains a list of DEX files and native libraries included in the bundle, and their hashes"* [12]. Listing 1 shows an example of the content of a CT JWT file. The list in the JWT file is signed with the developer's private CT key, which they are supposed never to share with another party. The CT JWT file is copied into all APK files generated from the AAB.

2. bundletool. https://github.com/google/bundletool
3. The ABI here roughly corresponds to a processor ISA.

```
{
  "codeRelatedFile": [
    {
      "path": "base/dex/classes.dex",
      "type": "DEX",
      "sha256": "7b22...d19e"
    },
    {
      "path":
↪  "base/lib/arm64-v8a/libjsc.so",
      "type": "NATIVE_LIBRARY",
      "apkPath":
↪  "lib/arm64-v8a/libjsc.so",
      "sha256": "b63e....7a49"
    },
  ],
  "version": 1
}
```

Listing 1: The signed content of a Code Transparency JWT

CT was designed to be verifiable by both developers and end users. *Bundletool* provides functionality for checking whether the JWT in an APK contains a valid hash for all the app's DEX or SO files. End users also need to verify that the key used for signing the JWT is the legitimate developer's CT key. This requires an additional secure channel between the developer and user to communicate the legitimate key.

3.5. Code Transparency Attack Scenarios

Android CT is intended to thwart supply-chain attacks that happen in the app distribution channel between an app's developer and an end user. As stated by Google, verifying the CT gives the developer *"assurance that, even if the APK itself was re-signed during distribution, the code verified by CT hasn't been modified"* [13]. CT verification is used *"for the purpose of inspection by developers and end users, who want to ensure that code they're running matches the code that was originally built and signed by the app developer"* [12]. Since Google has not provided a definition for the addressed supply-chain scenarios in the documentation for CT, we refer to Google's Android Binary Transparency scheme [14]. It is the equivalent of CT for protecting firmware images against supply-chain attacks. There, Google states that *"software supply chains are increasingly vulnerable to attacks, ranging from compromised signing keys to surreptitious code injection to insider attack"* [14]. In fact, similar events have happened in the past. For example, platform signing keys from Android vendors were found to have been leaked in 2022, enabling attackers to install apps with system privileges on affected devices[4]. There have also been examples in the past of CA key compromises [15], which play a similarly important role in the Internet's Public Key Infrastructure as app signing keys do on Android. As evident from the existence of CT for AAB, Google considers these realistic scenarios in app distribution as well.

4. Issue 100: Platform certificates used to sign malware. https://bugs.chromium.org/p/apvi/issues/detail?id=100

4. CT Attacker Models

This section discusses the attacker goals and models we use in our analyses. All attacker models are based on the assumption that an attacker exploits an app store's access to APK signing keys (as necessary for AAB) to produce and distribute malicious APKs that bear the legitimate signing certificate. On paper, Code Transparency promises the possibility to detect such attacks.

4.1. Attacker Goals

In the subsequent analyses in Section 5, we consider a scenario in which a malicious party aims to compromise a concrete *app installation*, i.e., aims to deliver a manipulated target application to a specific victim user. Moreover, we assume that the attacker wishes to operate stealthily, i.e., by only temporarily deploying the manipulated app as an update to a legitimate installation and later rolling back all changes. It is worth noting that compromising an APK that bears the legitimate APK signature is particularly desirable for an attacker. APKs modified in this way cannot be detected through app attestation and can be installed as updates to a legitimate install, thereby gaining access to all its private data.

In line with the Android Platform Security Model [8], we assume the attacker wants to compromise the main assets of an application: First, by gaining control over an application's execution flow (i.e., gaining code execution), the attacker can perform arbitrary actions from the target application's context. Second, by gaining control over a target application's private data directory, the attacker can extract sensitive data, e.g., credentials that they may misuse for user impersonation.

To summarize, we identify these two attacker goals that CT's integrity guarantees should prevent:

- **G1: Code injection.** Any attempt of an attacker to manipulate an APK for changing its runtime behavior should lead to the APK failing CT verification.
- **G2: Data access.** Any attempt of an attacker to manipulate an APK for gaining access to the application's private data directory should lead to the APK failing CT verification.

Please note that our analyses assume apps are installed and executed on an Android system that passes the Android Compatibility Test Suite, i.e., that runs a fully patched (as of August 2023) untampered (unrooted) specification-compliant OS.

4.2. Attacker Models

All attacker models we consider in our analyses assume an attacker holds some position in the app supply chain

between the developer and the end user. Through this position, the attacker gets access to the APK signing key for producing APK files that can be installed as an update to an original legitimate APK. We assume that developers are aware of the possibility for app manipulations in the supply chain when using the AAB format and precautionarily employ CT as a means to detect these manipulations. We further distinguish the attacker models in the degree of access the attacker has to the victim's device. Two attacker models assume an attacker who has compromised an app store operator's infrastructure (e.g., as an inside attacker). The last attacker model considers scenarios in which only the APK signing keys are compromised. The existence of CT shows that Google considers these scenarios to be realistic.

- **M1: Privileged app store.** In addition to being in possession of the APK signing key, the attacker can execute code from a privileged application installed on a target device. It is crucial to note that without modifying the target APK, even a privileged application is subject to the access policies described in the Android Platform Security Model [8]. In particular, it does not have any means to execute code in the context of another application or access its private data folder. Virtually all Android vendors operate their own privileged app stores, which, if compromised, allow an attacker to act under this model. Examples would be the *Google Play Store* on most Android devices, or *Huawei App Gallery* on devices produced by Huawei.
- **M2: Unprivileged app store.** The attacker may execute code in an unprivileged (user-installed) app store application on the victim device. A multitude of third-party app stores are available for Android, which, if compromised, can be exploited to the effect of this attacker model.
- **M3: Other role in the supply chain.** The attacker does not have any means to execute code on the victim device other than through injecting it into the target application.

5. Security Analysis of CT

In this section, we discuss the security flaws we identify in CT for AAB from analysing the official documentation and the source code of both *bundletool* and the Android Open Source Project. We distinguish between flaws inherent to the CT design and those that only affect the *bundletool* reference implementation.

5.1. Design Flaws in CT

Based on the security goals of CT we establish in Section 4, we identify the following flaws in its design.

F1: Optionality. Although intended to mitigate the severe security repercussions of the AAB distribution scheme, CT is only an optional feature. In fact, developers may only learn about its existence through subpages hidden in the Android documentation. While hampering the broad adoption of CT, its optionality also has serious consequences for the scheme's security. It means that users cannot know whether an APK lacking CT was submitted like this by the developer or subjected to its malicious removal somewhere along the supply chain.

F2: Scope. CT lacks coverage of key possibilities for manipulating an app's behavior. More specifically, the application's manifest is not covered, nor are its assets or resources. Since applications are free to include executable code in formats of their choosing at arbitrary locations in their APKs, this leaves severe gaps in the integrity guarantees of CT.

F3: Communication Channel. The CT design leaves a key question unanswered. The official documentation notes that the *"public [CT] key of the developer [...] must be provided by the developer over a separate, secure channel"* [12]. While the general problem is not specific to CT but associated to public key distribution in general, the CT does not provide any clues as to how such a separate secure channel to the user may be established, given that all information published by the distributor must not be trusted.

5.2. Implementation Flaws in *bundletool*

In addition to the design flaws described above, we also identify several implementation flaws in the *bundletool* reference implementation of CT for AAB.

F4: Certificate Reuse. In the documentation for CT, Google states that the CT key *"should be a unique key that is different from the app signing key"* [12]. In fact, this is an essential requirement for the integrity guarantees the scheme aims to provide. Given that the app signing key must be shared with the app distributor, the latter may otherwise regenerate the CT after manipulating SO or DEX files. However, *bundletool* does not enforce this crucial requirement. The tool accepts the same key being used for both adding CT to an AAB and for generating APK files from the AAB. The APK files produced in this way also pass all checks when verifying their CT through *bundletool*.

F5: DEX or SO in Assets. When an application includes DEX or SO files in its assets or other non-standard locations, verifying its CT fails even if the APK file has not been manipulated. This is due to an asymmetry between the implementations of CT generation and verification. While CT generation only considers DEX files in the root of the APK and SO files in the `lib` folder, verification ensures that all such files at any location in the APK have a valid entry in the JWT file.

As a result of this implementation flaw, developers of affected applications can not take advantage of CT to detect manipulations of their APKs. We note that, e.g., the *Facebook Audience Network SDK*[5], the most popular third-party advertisement library for Android [16], ships with a

5. Audience Network SDK. https://developers.facebook.com/docs/audience-network

DEX file in its assets folder. As a result, about 13% of all apps on Google Play are affected by this problem [17].

F6: App Archives. By default, when generating APK files from an AAB, *bundletool* also generates an app archive APK. This is a lightweight placeholder APK sharing the app's package name and app signing key. It may be installed in place of the app's full APK to save storage space while retaining its private data directory. The placeholder APK contains code and resources necessary to redownload the full APK when needed.

The addition of app archiving to *bundletool* in 2022 and Google's integration of the feature in Google Play marked the first time an app store used the power gained through the AAB scheme to actively change app functionality. Given that app archiving is retroactively added during APK generation, the code for redownloading the full APK is not intuitively part of the CT. Indeed, when app archiving was added to *bundletool* in March 2022 (version 1.8.2), placeholder APKs did not contain any CT, even if the AAB did.

This issue was solved in October 2022 (*bundletool* 1.12.0) by adding the hash of the placeholder APK's DEX file to every CT generated. The DEX file can, therefore, be later injected into generated APKs without tripping CT checks. Thus, with every CT generation, developers unknowingly agree a third-party DEX file to be executed in the context of their application. Even worse, Google does not disclose the source code of that DEX file. We conclude that app archiving effectively bypasses CT's integrity guarantees intended to protect against APK manipulations by app distributors.

6. Attacks Against CT

The attacks presented in this section take advantage of the CT flaws we discussed in previous Section 5. We experimentally confirmed all attacks on a Google Pixel 6a running Android 13.0. For simulating an **M1** attacker, we built Magisk[6] modules that installed our proofs of concept as privileged applications. An overview of the attacks, their exploited flaws, respective attacker model and goals may be found in Table 1.

A1: Code Execution by Stripping CT. Given the optionality of the feature, an attacker under any of our models may simply remove the CT of an AAB when generating a manipulated APK file for a victim user. Since no trace of CT is left in the generated APK file, the victim may only detect the manipulation if they have a separate secure channel to the developer. To summarize, a stripping attack works by:

1) Removing the CT JWT from the AAB submitted by the developer.
2) Replacing or adding DEX or SO files in the AAB.
3) Generating and serving APKs from the manipulated AAB.

6. Magisk: The Magic Mask for Android. https://github.com/topjohnwu/Magisk

A2: Code Execution through Library Injection. CT by design covers only executable parts of an application. It does not protect the integrity of the application manifest, which means that any changes an attacker applies to this file go unnoticed. This enables an **M1** or **M2** attacker to inject code into any app's classpath and enforce this payload's execution during app launch. More specifically, injecting code may be accomplished through the undocumented `uses-static-library` manifest entry. During app launch, the system parses this entry and loads all DEX files of the referenced packages into the app's classpath. There are no restrictions on referenced packages other than that they must be installed on the device and that their signing certificate must be known to the dependent application. The application manifest also offers multiple options for an attacker to enforce the invocation of the injected code. The most straightforward approach is adding a `ContentProvider` element that references a class in the injected code. When launching an application, the system automatically invokes the `onCreate()` method of any `ContentProvider`s it exposes. From there, the attacker may then, e.g., inspect files stored in the app's private data folder or manipulate the internal data structures of the ART runtime to alter further app functionality. To exploit this vulnerability, an attacker may follow these steps:

1) Under attacker model **M1**, the attacker uses the privileged `android.permission.INSTALL_PACKAGES` permission to install an additional package containing the malicious code. Under attacker model **M2**, the attacker includes the malicious code in the app store app already installed on the device.
2) The attacker injects the `uses-static-library` entry referencing the malicious code package and a `ContentProvider` into the manifest of the target application. Alternatively, the referenced library may simply contain a class named identical to one of the target application's app components.
3) When launched on the victim device, the attacker's malicious code is executed in the context of the target application.

A3: Code Execution through Debuggable Flag. Modifying its manifest also allows an attacker to mark an application as debuggable without influencing its CT. The `debuggable` manifest flag is usually only set on applications during development. It weakens some of the app isolation security features in the OS to facilitate inspecting the app's runtime behavior. Most notably, the ART runtime opens a Java Debug Wire Protocol (JDWP) debug port for applications carrying this flag. An app developer may then connect the Java Debugger (JDB) to the application through the Android Debug Bridge (ADB). The ADB stack involves a server running on the Android device while the system-wide debug setting is active. ADB clients may connect to this server through USB or TCP/IP. In accordance with the Android Platform Security Model [8], initial connections need explicit consent by the user. For wireless connections,

Attack	Flaw	Attacker Goal	Attacker Models	App Conditions	Android Versions
A1: Stripping CT	**F1**	**G1**	**M1**, **M2**, **M3**	None	All
A2: Library Injection	**F2**	**G1**	**M1**, **M2**	None	8.0+
A3: Debuggable Flag	**F2**	**G1**	**M1**	None	All
A4: Resources / Assets	**F2**	**G1**	**M1**, **M2**, **M3**	Relevant files	All
A5: Backup Opt-In	**F2**	**G2**	**M1**	None	4.0 - 13[a]

a. The vulnerability was fixed in Android 14 after our responsible disclosure

TABLE 1: Summary of the attacks we identified against CT.

this involves supplying the client with a numeric code generated and displayed by the ADB server on the device. Subsequent connections utilize exchanged asymmetric keys for authentication.

Although ADB is intended for connecting to the Android device from an external computer, connections from a local process are possible as well. We also found that a privileged application carrying the `android.permission.MANAGE_DEBUGGING` permission may bypass the user consent requirement described above. These possibilities allow an attacker under model **M1** to gain code execution in any app while the device is connected to a Wifi network by carrying out the following procedure:

1) The attacker sets the debuggable flag in the manifest of the target application delivered to a victim user.
2) The attacker uses the privileged `android.permission.MANAGE_DEBUGGING` permission to enable wireless debugging for the currently connected Wifi SSID. This works by invoking the `allowWirelessDebugging()` and `enabledPairingByPairingCode()` methods on the `AdbManager` system service.
3) The attacker may listen for the `WIRELESS_DEBUG_PAIRING_RESULT_ACTION` broadcast to learn the server port and pairing code that the user normally needs to manually enter into the ADB client.
4) The attacker establishes an ADB connection to localhost, authenticating using the intercepted pairing code.
5) The attacker can now manipulate the target application's execution by establishing a JDB connection via ADB.

A4: Code Execution through Resources or Assets. While CT intends to cover executable parts of an application, it is entirely ignorant of the fact that many applications contain executable code in non-standard forms or locations. CT only covers SO and DEX files in their default locations. However, applications may load DEX files or native code from any location within their APK. Additionally, developers commonly take advantage of cross-platform app frameworks such as *React Native*, *Xamarin*, *Cordova*, or *Ionic*. Many of these frameworks ship the app's functionality in code formats other than DEX or SO. An attacker under any of our models may trivially compromise an affected application by manipulating these custom code formats.

A5: Data Access through Backup Opt-in. System backups on Android usually need to be explicitly started by the user, either through the UI or ADB. Once manually initiated, the system takes a copy of the private data directory of all applications that do not opt out through the `allowBackup` manifest entry.

We identified a vulnerability in the Android OS that allowed privileged applications to bypass user confirmation for backups. Confirmation is implemented by having the `BackupManager` system service generate a random token that is passed to system UI. The backup process is started only if this token is reported back to the `BackupManager` service. Unfortunately, while the token was transported to system UI through an adequately secured channel, it was also leaked to the system's Logcat log. Since privileged applications may obtain the `android.permission.READ_LOGS` permission to gain access to these log messages, they were able to bypass the user's consent for backups. We responsibly disclosed this vulnerability to Google. A fix was implemented and is being deployed to end-user devices in the Android 14 release. For unpatched devices, an **M1** attacker may:

1) Remove the `allowBackup` flag from the manifest of the target application delivered to a victim user.
2) Use the privileged `android.permission.BACKUP` permission for invoking `adbBackup()` on the `BackupManager` system service.
3) Use the privileged `android.permission.READ_LOGS` permission to extract the leaked user confirmation token.
4) Spoof user confirmation to the `BackupManager` by passing the token to `acknowledgeFullBackup()`.
5) Extract the contents of the target application's private data directory from the backup.

7. AAB and CT In Practice

In this section, we evaluate the prevalence and security of CT in real-world applications. We concentrate our analyses on Google Play and Huawei AppGallery. Google Play is the official app store on the Android platform and, therefore, largest in terms of offered apps and active users. Huawei AppGallery was the first third-party Android app store to add support for AAB (in 2020) [18].

7.1. Prevalence of AAB and CT

For determining the prevalence of real-world applications using the AAB distribution format and CT, we carried out large-scale automated analyses on 3.3 million free applications from Google Play and 230k free apps from Huawei AppGallery. It is worth noting that neither platform publishes an official index of available apps. Not even the exact number of listed packages is publicly disclosed, so we are unable to provide any information regarding the completeness of our analysis.

The Google Play dataset was obtained from the *AndroZoo* project [19] in April 2023 and filtered to only include apps listed on Google Play at the time of our analysis. We additionally retrieved metadata directly from Google Play servers for the resulting 3 265 096 apps to detect whether they had been submitted as an AAB file (AndroZoo did not provide app metadata at the time of our study). The dataset from Huawei AppGallery consists of 226 568 free apps indexed through brute-forcing application identifiers on Huawei AppGallery in April 2024. Since no suitable information could be identified in AppGallery's app metadata, we interpreted the presence of the file `META-INF/BNDLTOOL.RSA` in served APKs as an indicator for original submissions as AAB. Unmodified *bundletool* versions store the APK signature at this location. However, since Huawei might have used a modified *bundletool* binary for some of the apps on AppGallery, we note that our results for Huawei AppGallery only represent a lower bound. The presence of CT in an app was assessed by scanning the ZIP central directory in the APK for an entry named `META-INF/code_transparency_signed.jwt`. Since this is the only allowed location for the CT file as per the scheme's specification, this analysis is precise. Carrying out the analyses took about 80 hours for Google Play and 5 hours for Huawei AppGallery.

7.1.1. Android Application Bundle Prevalence. Of the 3 265 096 applications analyzed from Google Play, 1 528 310 (46.8 %) had been submitted in the AAB format. The remaining apps must have been enlisted on Google Play before 2021 (even if they may since have been updated; AAB is only mandatory for *new apps* in general, not for updates). These numbers indicate an increase in AAB adoption compared to official numbers published by Google in 2021, when *"more than 1 million apps"* [20] were reported to have been submitted as AAB. Of the 226 568 analyzed apps from Huawei AppGallery, only 97 (0.04 %) had been submitted in the AAB format. We trace this low number back to the fact that the format is entirely optional for this app store. We note that all the apps that use AAB on any app store are susceptible to the supply chain attacks discussed in Section 6.

7.1.2. CT Prevalence. Only 21 (0.0014 %) of the applications submitted in the AAB format to Google Play contained a CT JWT. A full list of them may be found in Appendix A. Of these 21 applications, 11 are plugins for a geospatial planning app, all listed under the same publisher account on Google Play. 4 more applications were published by a single company. Additionally, three apps are stock portfolio management apps that are so similar in UI and functionality that they likely originate from the same developer, even if listed under different publishers on Google Play (all are French banks; two belong to the same enterprise group). We conclude that within the analyzed dataset, only four developer teams were aware of AAB's security implications and the (limited) countermeasure that exists in CT. None of the analyzed apps from Huawei AppGallery contained a CT JWT. We assume that CT's optionality (design flaw **F1**) and implementation flaw **F5** are major contributing factors to the low adoption of CT on both app stores.

It is worth noting that due to capacity constraints, we chose not to contact the developers of applications that lacked CT JWTs. This would have required us to establish secure channels (e.g., by finding contact information on developer websites; the data published in app stores may not be trusted) to the developers of almost all 3.5 million apps in our dataset. It would have allowed us to learn whether their submitted AABs had contained a JWT in the first place. We, therefore, potentially missed stripping attacks (See Section 6) in our analysis.

7.1.3. Verifying CT. We used *bundletool* to verify the integrity of the 21 samples in our dataset that contained a CT JWT. The tool confirmed that all JWTs contained matching hash entries for the DEX and SO files in the corresponding APKs. We also tried to ascertain that all CT JWTs were signed using the legitimate public key of the respective original developers. For the 11 plugin apps, the legitimate public key could be found in the open-source variant of the host app, which allowed us to completely verify these apps' CT with reasonable certainty. For the 10 remaining apps, we were unable to find trustworthy public information regarding their legitimate public key. We, therefore, contacted the support email addresses recorded in their respective Google Play listings. We could confirm the authenticity for all but one of the provided email addresses through the respective company websites. However, we did not receive any response to our emails within three months. Detailed results for this analysis can be found in Appendix A.

7.1.4. Susceptibility to CT Attacks. All of the apps we found to use CT are susceptible to attacks **A1**, **A2**, **A3**, and **A5**. This simply follows from the fact that these attacks do not impose any requirements on affected apps.

To also evaluate the susceptibility for app-dependent attack **A4** (Code Execution through Resources or Assets), we built a simple static analysis tool. This tool scans for files in an APK's assets or resources that carry the file signature or extension of the ELF, DEX, APK, or DLL formats. Additionally, all files in these folders that contain only printable characters are ran through the Esprima JavaScript syntax validator.

	A4	
Package Name	JS	DLL
com.atakmap.android.bng.plugin		
com.atakmap.android.compassnav.plugin		
com.atakmap.android.datasync.plugin		
com.atakmap.android.firesurvey.plugin		
com.atakmap.android.geocam.plugin		
com.atakmap.android.grgbuilder.plugin		
com.atakmap.android.takchat.plugin		
com.atakmap.android.uastool.plugin		
com.atakmap.android.vns.plugin		
com.atakmap.android.wave.plugin		
com.microsoft.loop	⚠	⚠
com.neuflize.obc.bourse	⚠	
com.oracle.ebs.maintenance	⚠	
com.oracle.ebs.scm.mwa.MSCA10	⚠	
com.oracle.ebsapps.csm	⚠	
com.oracle.events	⚠	
com.portzamparc.bourse	⚠	
com.somewearlabs.swtak.plugin		
net.bnpparibas.bourse	⚠	

TABLE 2: Susceptibility to attack **A4** of apps that use CT. *JS*/*DLL*: Apps integrate JavaScript code or Dynamic Link Library files.

Our analysis indicates that 8 of the 21 (38 %) applications that use CT are susceptible to **A4**. 8 apps implement some of their functionality in JavaScript, while 1 includes code in a Dynamic Link Library (DLL). None of the apps contained DEX, SO, or APK files in their resources or assets. The exact findings for each application are listed in Table 2.

7.1.5. Eligibility for CT and Possible Attacks Among Popular Apps. We also ran our static analysis tool described in Section 7.1.4 against a representative set of the most popular applications from Google Play. Our goal was to determine their eligibility for adopting CT and their susceptibility to our attacks once they do so. The dataset was assembled by collecting the package names of the 200 most popular free applications of all 36 categories on Google Play in December 2023. We again extracted the APKs for these applications from the *AndroZoo* [19] project. Since some apps were not in their collection, our final dataset consisted of 6 648 applications.

Our tool indicates that 3 935 (59.2 %) of the analyzed apps had been submitted to Google Play in the AAB format. One (0.02 %) app (*Microsoft Loop*, which we also found in Section 7.1.2) uses CT. 1 442 (21.7 %) of packages in our dataset are not compatible with *bundletool*'s CT implementation due to containing DEX or SO files in non-standard locations (**F5**). In 1 243 (18.7 %) of the analyzed apps, the integration of the Facebook Audience SDK at least contributed to this incompatibility. If they adopted CT, 3 467 of the analysed apps (52.2 %) would be susceptible to attack **A4**. All of them would be susceptible to attacks **A1**, **A2**, **A3**, and **A5**.

```
$ bundletool check-transparency --mode=apk
↪  --apk-zip=loop-patched-apks.zip
    APK signature is valid. SHA-256
    ↪  fingerprint of the apk signing key
    ↪  certificate (must be compared with
    ↪  the developer's public key
    ↪  manually): 94 29 4F 23 A8 ... 50 6B
    ↪  CD B0 22 AB 1F D8 C9 3C
    Code transparency signature is valid.
    ↪  SHA-256 fingerprint of the code
    ↪  transparency key certificate (must
    ↪  be compared with the developer's
    ↪  public key manually): 52 D4 B1 8E
    ↪  CA 3F 78 62 FF ... 89 54 40 B7 C2
    ↪  02
    Code transparency verified: code
    ↪  related file contents match the
    ↪  code transparency file.
```

Listing 2: The manipulated APK still successfully passes the CT verification in *bundletool*

7.2. Case Study

To demonstrate the practicality of the attacks described in Section 6, we provide a case study on attacking CT in a real-world application.

7.2.1. Microsoft Loop. The Android application (com.microsoft.loop) for this collaborative creation environment has been downloaded from Google Play by more than 100 000 users as of December 2023. The app employs CT, likely in an attempt to thwart the potential for supply chain attacks enabled by the AAB format. However, this attempt proves ineffective against manipulations of the app's runtime behavior. For this case study, we demonstrate how attack **A2** may be used to inject code for stealing login data without invalidating the APK's CT. We implemented a custom stage for the *A2P2* APK patching pipeline [21] that injects a `uses-static-library` element into the application manifest. We also created an additional package exposing the referenced library to the system, which needs to be installed on the victim device (see Section 6). The library code defines a class named identical to the *Microsoft Loop* application's main UI activity. Since static libraries take precedence over app code in Android's class loading, this is enough to intercept the app's launch. Our malicious main activity displays a fake login screen that, in a real-world attack, could, for example, forward all collected credentials to a web server controlled by the attacker. Despite the ability for the attacker to execute arbitrary code in the context of the *Microsoft Loop* application, its CT remains valid, as can be confirmed through the *bundletool* output in Listing 2.

Please note that we do not have access to the private key for the APK's app signing certificate, so we cannot entirely accurately simulate a supply chain attacker. In a real-world attack, the manipulated APK would be signed with the

developer's correct app signing certificate. As explained in Section 3, the AAB scheme requires the developer to share the app signing key with the distributor.

8. Discussion & Future Work

8.1. Improving AAB and CT

Given the number of serious design flaws in CT (see Section 5), we consider a complete secure design out of scope for this work. However, we would like to suggest improvements that mitigate the design problems raised in Section 5. We discuss fixes for the implementation flaws in *bundletool* in Appendix B.

F1: Optionality. The simplest solution for this issue is making CT a mandatory part of the AAB format. This would enable trivial detection of stripping attacks. If for some reason (e.g. backward-compatibility), such a change cannot be realized, we note that the information obtainable to the user through the separate secure channel to the developer (as discussed in the CT documentation [12]) at least needs to include an indication whether the AAB contained a CT when submitted.

F2: Scope. The attacks described in Section 6 highlight the necessity for CT to cover the application manifest, assets, and resources. However, it is unclear how such a solution may be implemented, given that *bundletool* itself modifies these parts of the application during APK generation.

F3: Communication Channel. We argue the need for a standardized registry for legitimate public CT keys. A solution might be akin to Certificate Transparency for TLS certificates [22].

Practicality. Currently, verifying an application's CT requires a deep understanding of the Android OS and development tools. It is, therefore, only accessible to advanced users. Future work needs to identify possibilities for improving the user experience of CT verification.

8.2. Preventing infrastructure-level attacks

Given the complexity of app stores, it is unlikely that the possibility for attacks compromising app store infrastructure (attacker models **M1** and **M2**) can ever be entirely eliminated. For vendors whose devices go through Google's GMS certification, further security could be imposed by requiring security audits of privileged app stores as part of the certification process. However, since Android consciously allows third-party app stores with no ties to Google nor device vendors, attacker model **M2** will always remain well within the boundaries of possibility.

8.3. Server-Side APK Generation

The fundamental proposition of the AAB publishing format is the possibility for distributors such as app stores to generate APK files optimized for end-users' devices. While optimized APKs undoubtedly improve the user experience, we argue that the AAB format is not necessary for serving optimized APKs to end users.

First and foremost, generating APKs on the server would be necessary if the number of possible device configurations to optimize for was unknown at compile time. However, these optimized builds are simply composed of all possible combinations of resource variants already provided by the app developer. Therefore, the optimized APKs could already be packaged during app build by the developer. In fact, functionality for generating a container of all possible optimized APKs is already available in *bundletool*.

Server-side APK generation only possibly offers advantages when new features are to be globally retrofitted into already submitted apps. However, the crude workaround implemented for App Archives (see **F6**) displays that any such efforts drastically interfere with the developer's interest in provable integrity guarantees for their app's behavior.

Given the lack of apparent advantages of server-side APK generation and its negative security repercussions, we argue that app stores should at least offer the option for developers to submit a container of locally built optimized APKs for their apps. This would allow them to sign the APKs themselves without having to hand over the APK signing key. As a result, they could benefit from the strong integrity guarantees of the APK format and app attestation.

Security Impact. Due to the small number of apps that use CT, the practical security impact of the identified flaws in CT alone is relatively small. However, the security impact of our work becomes apparent when considering the bigger picture: Given the security consequences of AAB (without CT), the low prevalence of CT means that a very large number of apps are vulnerable to supply chain attacks. Even worse, due to the current state of CT as discussed in this paper, no effective mitigations for the severe security repercussions of the AAB distribution scheme are available.

9. Related Work

Despite their impact on the security architecture of the Android ecosystem, to the best of our knowledge, our work represents the first scientific publication concerned with Code Transparency and the Android Application Bundle distribution scheme. However, research has been ongoing in the related domains of Android Supply Chain Attacks, App Integrity Checking, and App Misconfiguration. This section provides an overview of these fields.

Android Supply Chain Attacks. To the best of our knowledge, no other studies exist on the security of the global application supply chain *from the developer to the end user* on Android. However, several recent publications deal with various aspects of the software supply chain upstream of the developer, i.e., third-party app components and tools. The work that is most similar to ours in this category is the one by Wang et al. [23], which discusses the possibilities for malicious third-party libraries to stealthily override an application's security policies. A number of further studies elaborate on the perils of malicious app components in a benign app, such as Wang et al. [24], Kim et al. [25] and

Zhang et al. [26]. Several research teams, including Wu et al. [27], Zhan et al. [28], and Backes et al. [16] investigated the prevalence of vulnerable third-party libraries integrated into applications.

App Integrity Checking. Since the introduction of the Android OS, repackaging attacks have been a major focus of research attention. These attacks exploit the fact that APK signing does not provide any authenticity guarantees, allowing anyone to redistribute a manipulated version of a legitimate application. Research in this field has mostly focused on either detecting repackaging attacks or repackage-proofing applications. Works on repackage detection propose approaches based on watermarking [29] or similarities in app resources [30] or code [31]. These approaches scale poorly since they need to compare each sample to a large database of applications. Shi et al. [32] present a solution that does not work through comparison, but only works for repackaging attacks that use virtualization techniques. Suggestions for repackage-proofing include Stochastic Stealthy Networks (SSN) [33] that spread obfuscated signature checks throughout app code, logic bombs [34] that only trigger when executed on end-user devices or a combination of native and self-decrypting code [35]. However, as recently shown by Ma et al. [36], none of these countermeasures are insurmountable by reasonably determined attackers due to being confined to the same sandbox as the malicious code. An effective mitigation for repackaging attacks in server-client scenarios can be found in hardware-assisted app attestation, as discussed by Prünster et al. [10]. Several publications recently investigated the security of hardware attestation implementations. As part of these efforts, Aldoseri et al. [37] and Shakevsky et al. [38] identified severe flaws affecting millions of devices. Complementarily, Ibrahim et al. [39] and Berlato et al. [40] studied the integration of these technologies into real-world applications and discovered they were used by only 0.04 % of analyzed applications. All these protection methods are ineffective against the attacks discussed in our paper because they rely on the app developer having exclusive control over the APK signing key. However, AAB distribution requires the developer to share the APK signing key with the app distributor.

App Misconfigurations. Many of our attacks against CT exploit the powerful role of configuration files in Android apps. Several recent works investigated how mistakes in these configurations can introduce exploitable vulnerabilities. Yang et al. [41] and Jha et al. [42] constructed tools for automatically detecting manifest misconfigurations in compiled apps and found severe security flaws in widely used software. Oltrogge et al. [43] examined the Network Security Configuration in 1.3 million apps and uncovered that it was used for bypassing the default security policy in the overwhelming majority of cases. Lastly, Lenk et al. [44] analyzed the security implications of real-world `ContentProvider` configurations, reporting sensitive data leakage in a considerable number of apps.

10. Conclusion

In this paper, we presented the first comprehensive security analysis of Code Transparency (CT) for Android Application Bundles. We identified 3 design flaws and 3 implementation flaws in CT and its *bundletool* reference implementation. These flaws may be exploited through 7 different attacks for bypassing CT to gain code execution or data access in applications.

We further provided detailed statistics regarding CT in practice. We found that CT is almost non-existent in real-world applications despite the severe security consequences of the AAB format. We were also able to show that more than 20 % of the most popular applications on Google Play are not even able to use CT due to implementation flaws in *bundletool*. For apps that use CT, we provided a case study illustrating the consequences of the attacks we identified. Finally, we discussed possibilities for improvements to CT and AAB.

References

[1] I. Malchev, "Here comes treble: A modular base for android," online, May 2017. [Online]. Available: https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html

[2] A. Ghuloum, "Fresher os with projects treble and mainline," online, May 2019. [Online]. Available: https://android-developers.googleblog.com/2019/05/fresher-os-with-projects-treble-and-mainline.html

[3] K. N. Chris Sells, Benjamin Poiesz, "Use android jetpack to accelerate your app development," online, May 2018. [Online]. Available: https://android-developers.googleblog.com/2018/05/use-android-jetpack-to-accelerate-your.html

[4] T. Lim, "I/o 2018: Everything new in the google play console," online, May 2018. [Online]. Available: https://android-developers.googleblog.com/2018/05/io-2018-everything-new-in-google-play.html

[5] Google, "Android developers: About android app bundles," online, Nov. 2023, accessed 2023-12-05. [Online]. Available: https://developer.android.com/guide/app-bundle

[6] D. Elliott and Y. Becher, "Recent android app bundle improvements and timeline for new apps on google play," online, Aug. 2020. [Online]. Available: https://android-developers.googleblog.com/2020/08/recent-android-app-bundle-improvements.html

[7] J. Wentz, "App bundles for google tv and android tv," online, Nov. 2022. [Online]. Available: https://android-developers.googleblog.com/2022/11/app-bundles-for-google-tv-and-android-tv.html

[8] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, "The android platform security model," *ACM Trans. Priv. Secur.*, vol. 24, no. 3, 2021.

[9] M. Murphy, "Uncomfortable questions about app signing," online, Sep. 2020. [Online]. Available: https://commonsware.com/blog/2020/09/23/uncomfortable-questions-app-signing.html

[10] B. Prünster, G. Palfinger, and C. Kollmann, "Fides: Unleashing the full potential of remote attestation," in *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications ICETE*, 2019.

[11] Google, "Android developers: bundletool," online, Jul. 2023, accessed 2023-12-05. [Online]. Available: https://developer.android.com/tools/bundletool

[12] ——, "Android developers: Code transparency for app bundles," online, Jul. 2021, accessed 2023-12-05. [Online]. Available: https://developer.android.com/guide/app-bundle/code-transparency

[13] ——, "Android developers: Android app bundle frequently asked questions," online, Nov. 2023, accessed 2023-12-05. [Online]. Available: https://developer.android.com/guide/app-bundle/faq

[14] ——, "Google developers: Binary transparency," online, Aug. 2023, accessed 2023-12-05. [Online]. Available: https://developers.google.com/android/binary_transparency/overview#threat_model

[15] S. B. Roosa and S. Schultze, "Trust darknet: Control and compromise in the internet's certificate authority model," *IEEE Internet Comput.*, vol. 17, no. 3, 2013.

[16] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security CCS*, 2016.

[17] AppBrain, "Appbrain statistics: Facebook audience network," 2024. [Online]. Available: https://www.appbrain.com/stats/libraries/details/facebook_ads/facebook-audience-network

[18] Huawei, "Huawei developers: App bundles distribution," online, Dec. 2020, accessed 2024-04-19. [Online]. Available: https://web.archive.org/web/20201205143521/https://developer.huawei.com/consumer/en/agconnect/app-bundle/

[19] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Androzoo: collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016*, 2016.

[20] D. Elliott, "The future of android app bundles is here," online, Jun. 2021. [Online]. Available: https://android-developers.googleblog.com/2021/06/the-future-of-android-app-bundles-is.html

[21] F. Draschbacher, "A2P2 - an android application patching pipeline based on generic changesets," in *Proceedings of the 18th International Conference on Availability, Reliability and Security, ARES 2023*, 2023.

[22] B. Laurie, E. Messeri, and R. Stradling, "Certificate transparency version 2.0," *RFC*, vol. 9162, 2021.

[23] X. Wang, Y. Zhang, X. Wang, Y. Jia, and L. Xing, "Union under duress: Understanding hazards of duplicate resource mismediation in android software supply chain," in *32nd USENIX Security Symposium, USENIX Security*, 2023.

[24] J. Wang, Y. Xiao, X. Wang, Y. Nan, L. Xing, X. Liao, J. Dong, N. Serrano, H. Lu, X. Wang, and Y. Zhang, "Understanding malicious cross-library data harvesting on android," in *30th USENIX Security Symposium, USENIX Security*, 2021.

[25] J. Kim, J. Park, and S. Son, "The abuser inside apps: Finding the culprit committing mobile ad fraud," in *28th Annual Network and Distributed System Security Symposium NDSS*, 2021.

[26] Z. Zhang, W. Diao, C. Hu, S. Guo, C. Zuo, and L. Li, "An empirical study of potentially malicious third-party libraries in android apps," in *WiSec '20: 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020.

[27] Y. Wu, C. Sun, D. Zeng, G. Tan, S. Ma, and P. Wang, "Libscan: Towards more precise third-party library identification for android applications," in *32nd USENIX Security Symposium, USENIX Security*, 2023.

[28] X. Zhan, L. Fan, S. Chen, F. Wu, T. Liu, X. Luo, and Y. Liu, "ATVHUNTER: reliable version detection of third-party libraries for vulnerability identification in android applications," in *43rd IEEE/ACM International Conference on Software Engineering ICSE*, 2021.

[29] W. Zhou, X. Zhang, and X. Jiang, "Appink: watermarking android apps for repackaging deterrence," in *8th ACM Symposium on Information, Computer and Communications Security ASIA CCS*, 2013.

[30] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proceedings of the 30th Annual Computer Security Applications Conference ACSAC*, 2014.

[31] L. Glanz, S. Amann, M. Eichberg, M. Reif, B. Hermann, J. Lerch, and M. Mezini, "Codematch: obfuscation won't conceal your repackaged app," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering ESEC/FSE*, 2017.

[32] L. Shi, J. Ming, J. Fu, G. Peng, D. Xu, K. Gao, and X. Pan, "Vahunt: Warding off new repackaged android malware in app-virtualization's clothing," in *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[33] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu, "Repackage-proofing android apps," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[34] Q. Zeng, L. Luo, Z. Qian, X. Du, Z. Li, C. Huang, and C. Farkas, "Resilient user-side android application repackaging and tampering detection using cryptographically obfuscated logic bombs," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 6, 2021.

[35] C. Ren, K. Chen, and P. Liu, "Droidmarking: resilient software watermarking for impeding android application repackaging," in *ACM/IEEE International Conference on Automated Software Engineering ASE*, 2014.

[36] H. Ma, S. Li, D. Gao, D. Wu, Q. Jia, and C. Jia, "Active warden attack: On the (in)effectiveness of android app repackage-proofing," *IEEE Trans. Dependable Secur. Comput.*, vol. 19, no. 5, 2022.

[37] A. Aldoseri, T. Chothia, J. Moreira, and D. F. Oswald, "Symbolic modelling of remote attestation protocols for device and app integrity on android," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS*, 2023.

[38] A. Shakevsky, E. Ronen, and A. Wool, "Trust dies in darkness: Shedding light on samsung's trustzone keymaster design," in *31st USENIX Security Symposium, USENIX Security 2022*, 2022.

[39] M. Ibrahim, A. Imran, and A. Bianchi, "Safetynot: on the usage of the safetynet attestation API in android," in *MobiSys '21: The 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021.

[40] S. Berlato and M. Ceccato, "A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps," *J. Inf. Secur. Appl.*, 2020.

[41] Y. Yang, M. Elsabagh, C. Zuo, R. Johnson, A. Stavrou, and Z. Lin, "Detecting and measuring misconfigured manifests in android apps," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security CCS*, 2022.

[42] A. K. Jha, S. Lee, and W. J. Lee, "Developer mistakes in writing android manifests: an empirical study of configuration errors," in *Proceedings of the 14th International Conference on Mining Software Repositories MSR*, 2017.

[43] M. Oltrogge, N. Huaman, S. Amft, Y. Acar, M. Backes, and S. Fahl, "Why eve and mallory still love android: Revisiting TLS (in)security in android applications," in *30th USENIX Security Symposium*, 2021.

[44] C. Lenk and J. Kinder, "Poster: Privacy risks from misconfigured android content providers," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023*. ACM, 2023.

Appendix A. Details of apps using CT for AAB

Table 3 lists all apps in our dataset that use CT and their respective verification results.

Appendix B. Fixes to implementation flaws

In this section, we discuss how the implementation flaws in *bundletool* that we identified in Section 5.2.

Package Name	Version	Publisher	Valid CT	Original Key	Unique Key?
com.atakmap.android.bng.plugin	2.0.0-SNAPSHOT [4.8.1]	TAK Product Center	✔	✔	✔
com.atakmap.android.compassnav.plugin	1.0 (8a9728b2) - [4.8.1]	TAK Product Center	✔	✔	✔
com.atakmap.android.datasync.plugin	1.6.6 (1b388a2d) - [4.8.1]	TAK Product Center	✔	✔	✔
com.atakmap.android.firesurvey.plugin	2.9.01.2 (95e2d92a) - [4.8.1]	TAK Product Center	✔	✔	✔
com.atakmap.android.geocam.plugin	1.0 (0ffd4a57) - [4.8.1]	TAK Product Center	✔	✔	✔
com.atakmap.android.grgbuilder.plugin	1.1 (3a1fc26f) - [4.8.1]	TAK Product Center	✔	✔	✔
com.atakmap.android.takchat.plugin	1.0 (a43b4fbb) - [4.8.1]	TAK Product Center	✔	✔	✔
com.atakmap.android.uastool.plugin	12.2 (f2ccbf8d) - [4.8.1]	TAK Product Center	✔	✔	✔
com.atakmap.android.vns.plugin	3.6 (9d26d997) - [4.8.1]	TAK Product Center	✔	✔	✔
com.atakmap.android.wave.plugin	2.0 (4bca9c49) - [4.8.1]	TAK Product Center	✔	✔	✔
com.microsoft.loop	1.0.0321.00	Microsoft Corporation	✔	?	✔
com.neuflize.obc.bourse	5.6.8	Banque Neuflize OBC[a]	✔	?	✔
com.oracle.ebs.maintenance	10.0.0	Oracle America, Inc	✔	?	✔
com.oracle.ebs.scm.mwa.MSCA10	10.0.0	Oracle America, Inc	✔	?	✔
com.oracle.ebsapps.csm	10.0.0	Oracle America, Inc	✔	?	✔
com.oracle.events	1.0	Oracle America, Inc	✔	?	✔
com.portzamparc.bourse	5.6.8	Portzamparc SA[a]	✔	?	✔
com.somewearlabs.swtak.plugin	0.11.9 - [4.8.1]	TAK Product Center	✔	✔	✔
net.bnpparibas.bourse	5.6.11	BNP PARIBAS[a]	✔	?	✔

a. These apps likely are maintained by the same developer

TABLE 3: The apps in our dataset that use CT for AAB and their respective verification results. All apps contained a valid Code Transparency (CT). In column *Original Key*, **?** signifies that the original developer's public CT key could not be obtained. It therefore was not possible to check whether it matched the certificate found in the CT JWT. *Unique Key* denotes that the app signing key was not reused as the CT key.

B.0.1. F4: Certificate Reuse. Checks need to be added to APK generation in *bundletool* to ensure the CT key may not be used as the app signing key. Additionally, the APK verification result should include information on whether the two keys are distinct.

B.0.2. F5: DEX or SO in Assets. This issue may be mitigated by aligning the CT verification policy with its creation policy. The most sensible approach would be including all DEX or SO files in the CT, irrespective of their location in the AAB/APK.

B.0.3. F6: App Archiving. It is unclear whether the CT scheme should allow such radical modifications of an app as carried out through this feature. We believe that all source code of the placeholder APK should at least be available for public inspection.