Mobile App Distribution Transparency (MADT): Design and evaluation of a system to mitigate necessary trust in mobile app distribution systems*

Mario Lins[1], René Mayrhofer[1], Michael Roland[1], and Alastair R. Beresford[2]

[1] Johannes Kepler University Linz, Linz, Austria
{lins,rm,roland}@ins.jku.at
[2] Dept of Computer Science and Technology, University of Cambridge, Cambridge, UK
arb33@cam.ac.uk

Abstract. Current mobile app distribution systems use (asymmetric) digital signatures to ensure integrity and authenticity for their apps. However, there are realistic threat models under which trust in such signatures is compromised. One example is an unconsciously leaked signing key that allows an attacker to distribute malicious updates to an existing app; other examples are intentional key sharing as well as insider attacks. Recent app store policy changes like Google Play Signing (and other similar OEM and free app stores like F-Droid) are a practically relevant case of intentional key sharing: such distribution systems take over key handling and create app signatures themselves, breaking up the previous end-to-end verifiable trust from developer to end-user device. This paper addresses these threats by proposing a system design that incorporates transparency logs and end-to-end verification in mobile app distribution systems to make unauthorized distribution attempts transparent and thus detectable. We analyzed the relevant security considerations with regard to our threat model as well as the security implications in the case where an attacker is able to compromise our proposed system. Finally, we implemented an open-source prototype extending F-Droid, which demonstrates practicability, feasibility, and performance of our proposed system.

Keywords: Mobile app distribution · Transparency logs · Supply-chain security · Verifiable trust · Digital signatures

L. Fritsch et al. (Eds.): NordSec 2023, LNCS 14324, pp. 1–19, 2024.
https://doi.org/10.1007/978-3-031-47748-5_11

1 Introduction

Supply-chain attacks are popular, omnipresent, and effective as evidenced by recent reports about significant attacks and events such as NotPetya, XcodeGhost, or the SolarWind attack [5,13]. Due to their potential severity and automatic distribution to thousands or even millions of trusting users [5], state actors such as China or Russia are actively invested in exploiting software supply chains [13].

According to the MITRE ATT&CK® knowledge base [27], supply chains can be compromised in several ways, like manipulating the software update/distribution mechanisms, replacing legitimate software with modified versions, or selling modified/counterfeit products to legitimate distributors. These examples often involve compromising existing trust anchors like signing keys or certificates [13].

We focused our research on supply chain security of mobile app distribution systems which rely on certain trust anchors, like digital signatures. As these signatures are an integral component in well-known mobile app distributions systems such as Google Play or F-Droid, there is often no alternative but to trust them completely. Although, digital signatures are used to ensure the integrity and authenticity of apps, we have identified certain threats in current mobile app distribution systems that could lead to significant security concerns for a user or the developer of the respective app. These include leaked signing keys that may be used by unauthorized entities, malicious distributors, insider attacks or even attempts to distribute different app versions to specific users.

This paper introduces a novel concept, built on transparency logs, to improve verifiability and discoverability of potential attacks related to digital signatures, with a particular focus on mobile app distribution systems. We concentrate on the digital app signature since it is a key part of ecosystem security.

2 Preliminaries

2.1 App Signing Process

Google Play Store provides an integrated feature, called Play App Signing [12], that manages and protects the private key used for signing the APK file[3]. This approach requires that the private key is managed by Google's Key Management Service and thus it needs to be stored on Google's infrastructure. For Android apps published before August 2021, the Play App Signing approach is optional and developers can still manage app signing keys themselves. However, for newly published apps, the Play App Signing approach is mandatory. This particular policy adaption by Google results in a centralized trust anchor that has to be trusted by both the user and the developer (more details in section 3.2).

F-Droid [9] is an alternative distribution system for free and open source Android apps. If a developer wants to sign an APK file, F-Droid provides two

[3] As developer identities are not directly verified by most Android app distribution systems, authenticity of signing keys is typically only guaranteed in the Trust-on-First-Use (TOFU) model.

possible procedures for that. One approach is to publish two APK versions where one is signed by the developer and the other one is signed by the F-Droid repository (with a key held by F-Droid, comparable to Google Play Signing in this case). This is especially useful for distributing updates for apps that have been installed via different distribution channels (e.g. Play Store) and for apps available through F-Droid. This approach requires including the signature of the developer into the corresponding metadata description of the particular app and that the app is still reproducible by F-Droid. The other approach requires the developer to provide a reference to the signed APK file. If F-Droid is able to reproduce the APK file in a way that it matches the referenced one, F-Droid publishes the signed APK of the developer directly without signing it again with the F-Droid repository key.

2.2 Verifiable Logs

A verifiable log [8,15,16] is a data structure that is based on an append-only ledger that is cryptographically secure. The Merkle tree is a popular example where it is not possible to retroactively insert, delete, or modify any record. One of the main advantages of this data structure is that these properties are auditable, either publicly or at least by its consumers (e.g. when hosted in an internal network). The data stored in a verifiable log is application-specific and is not defined by the log itself. A verifiable log is stored on one or preferably multiple servers that are accessible by clients, which may not necessarily be trusted. Clients do not have to trust the log server as the data structure allows verification of the proper behavior of the log itself.

Merkle Trees We base our design on a verifiable log using a binary Merkle tree [20] to allow efficient auditing and to provide tamper protection due to its append-only property. The Merkle tree consists of leafs and nodes, with the top node called *root node*. The leaves represent data that are managed by the tree. Values are attached to internal nodes and are calculated as a cryptographic hash function (e.g. SHA-256) of their children, recursively, until a value of the root node is reached. Trees do not need to be balanced and therefore can store an arbitrary amount of data.

Inclusion Proofs An *inclusion proof* allows one party to prove to another that a particular leaf exists in a Merkle tree. This proof can be constructed efficiently, as it only requires the so-called Merkle Audit Path [11,15,16] which represents the shortest path from the respective leaf to the root node hash of the tree. The remaining leaves and nodes are not needed for this calculation. This approach means that the calculated root node hash is compared to the expected root node hash. If these hashes are equal, we have proven the particular leaf is part of the tree[4]. The left tree in Fig. 1 highlights the path that is required for such an

[4] We consider a particular tree to be fully represented by its root hash, which can in turn be contained within an updated or larger tree with a different root hash.

inclusion proof for *Record 2*. The required components for the calculation are marked in red. Below is a step-by-step description of validating an inclusion proof for the example given in Fig. 1.

Given an ordered list of node hashes `A` and `C`, the inclusion proof can be verified as followed:

1. Calculate the leaf hash of *Record 2*: `B` = SHA-256(`0x00` || *Record 2*), where `0x00` is used as a prefix for leaf hashes and `0x01` for nodes to provide second preimage resistance [16].
2. Calculate the node hash `E` = SHA-256(`0x01` || `A` || `B`).
3. Calculate the root node hash `root` = SHA-256(`0x01` || `E` || `C`).
4. Compare the calculated `root` hash with the claimed `root` node hash.
5. The inclusion proof is valid if the hashes are equal.

Consistency Proofs A consistency proof [16] can be used to verify if the append-only property of the Merkle tree is valid. The append-only property ensures that it is not possible to insert, modify, or delete a leaf or node in the tree retroactively. Therefore, the consistency proof validates if a previously generated version of the tree is part of the current tree that may have been extended by new entries.

Assuming two Merkle trees, `Tree_Old` and `Tree_New` as shown in Fig. 1, where `Tree_Old` is a previous version of `Tree_New`, a consistency proof provides an ordered list of node hashes in order to perform a verification whether the entries of `Tree_Old` is still equal to the corresponding entries in `Tree_New` or not. Given the root node hash of `Tree_Old`, a root node hash of `Tree_New`, and the corresponding consistency proof [`E`,`C`,`D`,`I`], the verification of that proof can be calculated as followed:

1. Calculate the resulting node hash: `X` = SHA-256(`0x01` || `E` || `C`).
2. Verify that `X` is equal to root hash of `Tree_Old`.
3. Calculate `F` = SHA-256(`0x01` || `C` || `D`).
4. Calculate `J` = SHA-256(`0x01` || `E` || `F`).
5. Calculate root node hash of `Tree_New`: `Y` = SHA-256(`0x01` || `J` || `I`).
6. Compare the calculated root node hash `Y` with the claimed root node hash.
7. The consistency proof is valid if the hashes are equal.

2.3 Split-View Attack

A relevant attack on such verifiable logs that also applies to the design proposal of this paper is called split-view attack [18,24]. A log subject to such an attack would be able to present different log representations to its clients while still maintaining the append-only property given by the Merkle tree. This means that all operations performed by a client on a specific log (e.g. inclusion and

Within the scope of inclusion proofs we thus use the terms 'tree' and 'root hash' interchangeably wrt. the provided security guarantee.

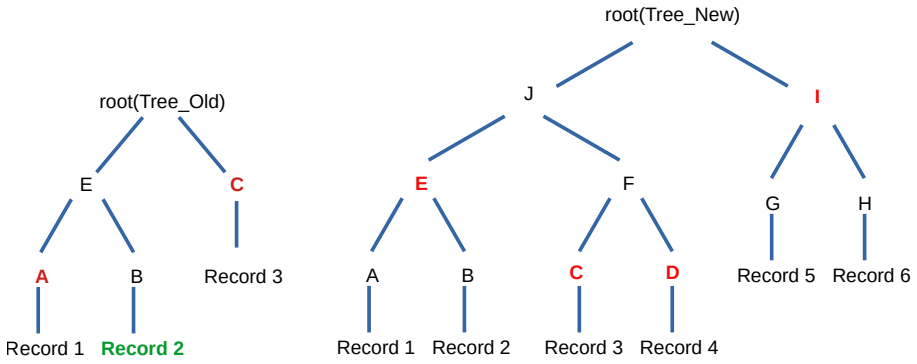

Fig. 1: Merkle Tree proofs

consistency proofs) seem valid, yet receiving different data than seen by other clients. However, once a log carries out a split-view attack, it must consistently maintain different views for each subgroup of clients since doing otherwise is detectable. A security evaluation and suggestions to counter this kind of attack are given in section 5.1.

2.4 Personality

The term *personality* is used by Google Trillian [6] and describes the application-specific interface to access a log server. The main responsibilities of a personality are defining and validating the application-specific data model, optionally providing access control and in case the personality and the log is maintained by different parties, providing auditable information for external verifiers.

2.5 Monitor, Auditor and Witness

One of the main advantages of a verifiable log is that it enables interested parties to detect misconduct up to even malicious behavior regarding certain log entries. It is possible to set up monitors, auditors or witnesses that periodically verify the behavior of the log or notify subscribers in case of suspicious behavior. A monitor may store previous copies of the verifiable log in order to verify the consistency between a new and previous versions. An auditor typically verifies the consistency of only a subset of the log by performing inclusion proofs. A witness [18,26] on the other hand is an independent entity that observes one or more log systems to prevent split-view attacks. Log auditors can thus have more confidence that a log system is truly and globally consistent if multiple independent witnesses have a consensus about the specific state (checkpoint) of the log. The witness cosigns a checkpoint after verifying that an evolution of a previously signed checkpoint is consistent with it. In case the log or the witness are new, the witness uses the trust-on-first-use approach.

3 Threat Model

The focus of our threat model is to identify potential security impacts with regard to the authenticity and integrity of APK files that are distributed by a mobile app distributor (e.g. Google Play, F-Droid). The most important security control that is used to ensure authenticity and integrity are digital signatures. Therefore, most of our threats[5] address scenarios where the signature is compromised or even used by malicious actors. As there are different parties involved in mobile app distribution systems, we first define potential stakeholders.

3.1 Stakeholders

Developer: The developer wants to distribute an app via a mobile app distribution system. From the developer's perspective, it is important that unauthorized entities cannot manipulate the app or even publish app updates on their behalf.
User: The user primarily wants to use the app and may want to verify the authenticity and integrity of the app to be sure that it has not been manipulated.
Distributor: The distributor wants to distribute apps to its users using secure infrastructure. Furthermore, the distributor wants to provide its users with security by incorporating controls such as digital signatures to prevent repository spoofing or malicious app updates[6].

A stakeholder may also take over more than one role, like a developer who is also hosting a distribution system.

3.2 Threats

Threat 1: Signing key is leaked and used by an unauthorized party. The most relevant threat that is addressed by the proposed system is that an unauthorized party uses the app signing key to distribute malicious updates. If the holder of the signing key is not aware that the key has leaked, they may not recognize that it is used by an unauthorized party. This is also relevant even in case that the holder of the signing key monitors certain distribution channels, directly as these could also be untrustworthy or even malicious.
Threat 2: Unauthorized usage of the signing key due to compulsory outsourcing. As mentioned in section 2.1, the current Google Play policies enforce the developer to store the signing key on Google's infrastructure so that the developer is not in control of the signing key anymore and comparable app distributors have similar policies. This restriction requires full trust in this external storage and that no unauthorized entity can access the security relevant signing key(s). In that particular case, the developer or the user cannot verify if the signing key has been compromised.

[5] Most of the threats that we have identified can also be found elsewhere [2,3,17,19,23].
[6] Payment and IP protection mechanisms are already addressed in existing systems and considered out of scope of the threat model in this paper.

Threat 3: Deliberate use of the signing key. The key holder, who may be an outsourced storage provider, may for example be forced to sign the corresponding app update by the respective judicative or due to economic interests. The developer would not have a possibility to detect that the outsourced key has been abused. Reports of government interventions in the mobile world reinforce the associated potential threat. In 2021, the New York Times [22] reported the removal of tens of thousands of apps from Chinese app stores.
Threat 4: A user may get another version of an app than other users. If the signing key is compromised, a user cannot ensure to have the same version of the app as all other users have. A distributor could provide a tampered version only to a subset of users. This threat may also be interesting in terms of censorship, enforced by state actors like the Internet censorship regime of Iran [1]. How can a user be sure to receive the same version in, e.g., the USA and in Iran without any geographical differences? However, distributing different app versions is also done by the app developers themselves, as a recent study [14] revealed 596 apps with geographical differences that may expose a certain security and privacy risk for users in those countries.

4 Architecture of the Verifiable System Design

This section introduces the components used to design our novel concept. To evaluate and to verify the viability of the proposed system, we have implemented a proof-of-concept prototype. Specific implementation parts have been set up by using or adapting available open source components, including F-Droid for distributing Android apps and Google Trillian for the transparency log backend. The first subsection details the individual phases with regards to the previously defined stakeholders. The second subsection lists the involved system components from a more software-centered approach.

4.1 Phases

The proposed system design includes three main phases based on the intended usages of the defined stakeholders: distribution, verification, and monitoring phase. Fig. 2 provides an overview about the phases including the relevant stakeholders and tasks.
Distribution phase: The distribution phase begins as soon as the developer has finished the implementation of the app. At this point, the *developer* wants to distribute the app to its *users* by using the respective infrastructure of the *distributor*. First, the *developer* uploads the app or the source code of the app to the store provided by the *distributor*. Additionally, the *developer* may want to sign the app or allow the *distributor* to sign the final package.

At this point, our system proposal extends the workflow by extracting relevant app metadata that is going to be published and to create a respective log entry via the dedicated transparency log system. This step is performed by the

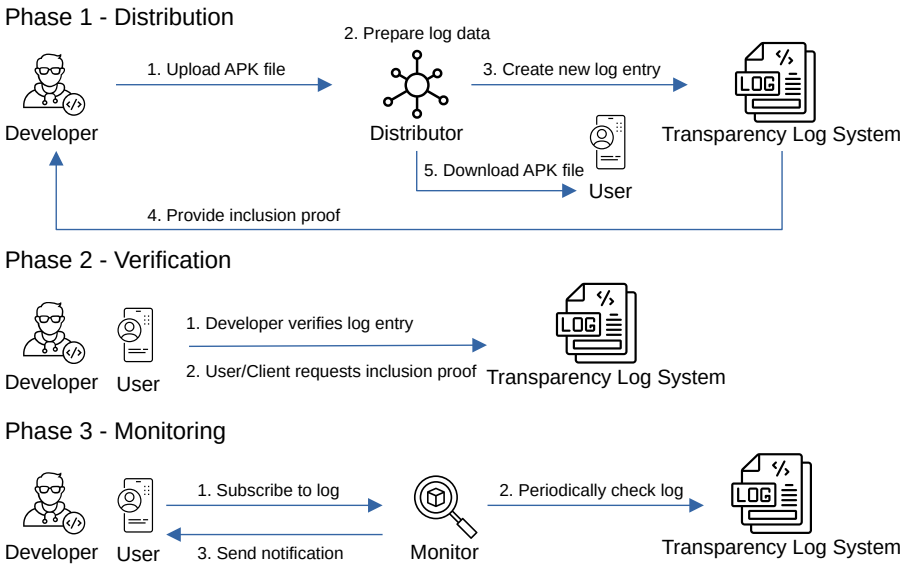

Fig. 2: System flow

distributor. As soon as the log entry has been successfully added to the log, the *distributor* can release the app to its *users*.

Verification phase: Once the app is available through the channel of the distributor, client-side verification can be conducted. Verification can be done by several entities: the developer, the user, and potentially also by existing witnesses. The developer may want to verify if the app has been logged properly. This can be done by requesting an inclusion proof of the log. If the inclusion proof verifies the developer can be sure that the distributor has properly logged the uploaded APK file. Automatic verification on the user side is done by the client of the distribution system (e.g. F-Droid client). The client downloads the requested app, calculates the expected logging information, and requests the corresponding inclusion proof from the personality. If the expected information and the logged information match, there will be no warnings shown to the user. If it does not match, the app can still be installed, but the user will receive a warning. As our transparency log system is publicly available, a user always has the possibility to verify the log entry manually even without trusting the client of the distributor.

Monitoring phase: The monitoring phase may start after the app has been published and verified by the developer. The developer can subscribe to notifications from a monitoring instance that observes the transparency log for new entries based on the application ID and the version that the developer is interested in (e.g. `com.example.sampleapp:v1.0`). When the monitor detects a new log entry with the given namespace, it notifies all subscribers. In case the developer has not published a new update, someone else is trying to publish one.

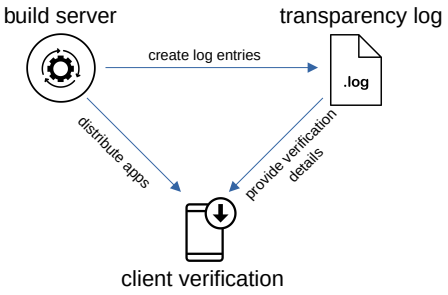

Fig. 3: System overview

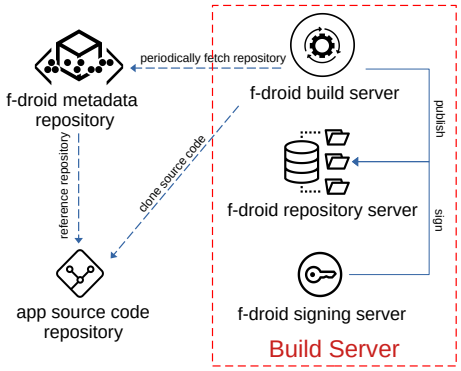

Fig. 4: Build server

4.2 System Components

This section details the involved system components from a more technical point of view including the prototype implementation. Fig. 3 illustrates the three main components and their interactions.

Build Server The build server part includes the relevant components to build, publish (sign) and deploy the given app. The individual tasks of such a build server are to fetch and build the source code, to verify it's reproducibility and to sign and publish the APK to the distribution server. Our prototype implementation is built around F-Droid. The F-Droid build server includes the build environment, a dedicated F-Droid repository, and a signing server as illustrated in Fig. 4. There were no changes needed for the F-Droid repository and signing component. Although, for being used in a production environment, we would recommend to request an inclusion proof before making the APK file available to users.

Prototype implementation: The most relevant changes for our prototype implementation were done in the build server component itself. During the publishing process, the build server signs the APK file and publishes it to the repository. At this point the prototype implementation adds additional steps to the workflow before the signed APK file is finally deployed to the remote web server where it is available for all users:

1. Extract relevant APK metadata (see section 4.2).
2. Select the proper tree ID for the specific repository.
3. Create a JSON object compliant to the personality data.
4. Request an authentication token from the personality.
5. Send tree ID and JSON object to the personality to create new log entry.

Transparency Log The transparency log system consists of three components as described below.

1. *Log Server:* The core element of our transparency log system is the log server that manages one or multiple Merkle trees including the associated functionalities like performing inclusion or consistency proofs. The log server implementation is based on Google Trillian and did not require any adaptions to work with the system as it is designed to be application-independent.
2. *Database:* The database is used to persist the Merkle tree. Our implementation uses a MySQL database.
3. *Personality:* The personality is the application-specific interface in front of the log server. The personality defines and validates the data structure that is used to store the leaf content in the transparency log. Additionally, it exposes an interface to its users to interact with the transparency log. Our prototype implementation does not allow everyone to create new log entries. Therefore, special endpoints of the personality can only be access when properly authenticated and authorized for them.

Prototype implementation: Our prototype implementation includes a dedicated personality, developed as a REST service by using the .NET core framework. By using our build pipeline we are able to build a docker image including a configurable personality instance. The Google Trillian implementation of the log server provides the required **.proto* files to interact with the log server via gRPC. The personality is responsible for defining the data structure that is stored within the Merkle tree and for potential data validation tasks. Furthermore, it also performs proper conversion from the C# object to the byte array that is finally stored in the transparency log. An essential implementation detail that we had to take care of was to use the proper hashing algorithm and dedicated prefixes depending on the type of the tree element (e.g. 0x00 for leaf and 0x01 for node elements), cf. RFC 9162 [16]. To prevent unauthorized write access on the log, we have introduced two roles and implemented a token-based authentication scheme. We are using an admin role responsible for managing trees (e.g. creating a new tree or deleting an existing one) and a build-server role that is authorized to create new log entries. Our docker image is parameterized to allow inclusion of pre-defined credentials for both roles. If the F-Droid build server, e.g., wants to create new log entries, it has to provide the correct credentials to the personality first. If the credentials are valid, the personality provides the F-Droid build server an authentication token that can be used to create new entries in the log.

Data Structure The data structure of the records stored in the verifiable log includes the following information that is required to uniquely identify the package as well as to verify its integrity:

- *applicationId:* The unique APK file package name.
- *version:* The version number of the app release.
- *apkHash:* A cryptographic hash of the APK file to verify its integrity.

Client Verification This component is responsible for verifying that apps are downloaded from a distributor were properly logged. This involves several steps as listed below:

1. The client downloads the APK file, but does not start the installation.
2. The verification library gets relevant metadata (application ID, version, and hash value) of the APK file.
3. An inclusion proof is requested by sending a specifically crafted data object including the metadata to the personality.
4. In case there is an inclusion proof available, the verification library calculates the expected root hash locally.
5. The locally calculated root hash is compared with the claimed root hash of the log server.
6. If the root hashes are equal and therefore the inclusion proof is valid, the client installs the app without further notice.
7. In case the inclusion proof is not valid, the user is notified, but can continue to install the app.

Prototype implementation: We have implemented a dedicated Android library to perform the end-to-end verification of the distributed APK file to verify whether it is properly logged or not. One of the main functionalities that are currently implemented is the end-to-end verification by validating an inclusion proof that is provided by the personality. Besides the verification part, the prototype implementation handles all other kinds of communication to the personality (e.g. requesting available tree IDs). The library has been developed in Java and is publicly available. One main reason why we have decided to do the implementation in a dedicated Android library is that interested parties can easily use it in a separate app or even integrate it into the official clients of the mobile app distributors (e.g. within the official F-Droid client app).

5 Evaluation

5.1 Security Evaluation

This section evaluates our proposed system design to determine whether the identified threats, listed in section 3.2 can be successfully mitigated. Furthermore, the security implications are analyzed in case an attacker is able to compromise one or multiple of the newly added components.

Threat Mitigation Our system design makes any distribution attempt though (or by) a distributor transparent and thus verifiable. In particular, the client verifies the log entry before installing an app and therefore it is not possible to distribute an app without creating a new entry in the append-only and tamper-proofed logging system. As this entry includes the package string, the version, and the hash of the APK file, any interested party can verify if this aligns with the

corresponding log entry and that it is the same APK version that everyone else has (Threat 4). A developer or an app distributor who monitors the log would receive a notification (Threats 1, 2, 3) as soon as the log entry of the particular app is created so that unauthorized distribution attempts can be detected. Our system design also enables independent and verifiable monitoring instances and thus does not rely on trustworthiness of a distributor. Therefore, we can also avoid falsified or missing information (e.g. suppression of publication attempts) compared to monitor specific distribution channels, directly (Threat 1).

Security Implications An attacker who is able to compromise one or multiple of our newly added components could also have a major security impact with regards to authenticity and integrity of the mobile distribution system. Therefore, we also evaluate the security of our approach including the newly added components[7]. The following paragraphs describe the results of our evaluation.

Malicious distributor bypasses the logging system. If a compromised distributor tries to bypass or manipulate the log entry, it avoids creating a new log entry so that a manipulated APK version could be distributed because it cannot be verified. In that case, the client would detect that for this particular APK the log entry is missing and notify the user about this security incident, who can decide how to proceed.

Malicious distributor creates a manipulated log entry. As a client would detect the absence of expected log entries, a malicious distributor may try to create a manipulated log entry. To provide a valid log entry that matches a manipulated APK file, the distributor needs to calculate the hash value of the manipulated version and write the new hash value to the log. A client that verifies the manipulated APK file, calculates the hash of it and verifies the respective log entry. The verification would be successful as a log entry is present and the hash values match. However, this manipulation attempt may be detected by monitors. The developer of the app, for example, may have registered the APK name space on a monitor that observes the log. The monitor would recognize that there is a new log entry for a particular APK file so the developer (as subscriber) will be notified. The developer could then easily compare the real hash value of the original version with the hash value of the log entry and would detect the manipulation attempt. Further steps to be taken in that particular case are out of scope for this paper.

Malicious client bypasses the log verification. An attacker may be successful in tricking the victim into installing a manipulated version of the distribution client that bypasses the logging verification to allow an attacker to distribute malicious APK files via the distribution channel. Besides the fact that an attacker who is able to trick a victim into installing a malicious client could also install other

[7] Note that global passive adversaries may learn which apps are installed by clients by monitoring transmitted inclusion proofs, leaf log entries, and/or the embedded APK metadata. However, as there are many other ways to learn the same information under our threat model, we consider this as out of scope and not a reason for keeping such data confidential.

malicious APK files the same way, there are two efficient countermeasures in place: First, the client app could also be logged in the transparency log so that the client can manually perform an inclusion proof. Second, the user could use a dedicated app that performs the necessary calculations and communication to the personality.

Malicious log server. A log operator may try to manipulate a log entry while maintaining the same root hash. In this scenario the log operator may only manipulate a single leaf, but keeps the root node hash the same so that the cryptographic proofs for the remaining entries are still valid. This attack scenario can be mitigated by running full audits on the Merkle tree. A full audit recalculates the root node hash from the available leaf values. If the full audit results in a root node hash that does not match the claimed one, the suspicious behavior can be detected. From a component point-of-view, a full audit could be performed by monitors.

Unauthorized write access to the log server. The transparency log should be publicly readable by design to allow every interested party to verify log entries. However, when it comes to write permissions, it is essential to consciously decide who is allowed to write to the log. In case arbitrary parties are allowed to write to the log, it is still possible to verify the entries. However, data that has been written to the log can never be removed again due to its append-only property. In regard to the design proposal, it is suggested to only allow authenticated distribution systems to write to such logs.

Split-view attack by a malicious log server. A split-view attack can be mitigated by using witnesses. The log server is independent of the specific application and thus any witness system could be used. However, to make use of the advantages of the consensus of witnesses, the client would need to verify them in addition to inclusion proofs. This functionality is currently not implemented in the prototype.

Orthogonally to the use of witnesses, split-view attacks can be mitigated by querying the personality and log server, e.g. via Tor [28] circuits, as this would make providing consistent split views unrealistic.

Malicious personality. A personality is not necessarily hosted in the same trust zone or operated by the same operator as the log server. Therefore, an external auditor may also want to audit the personality to verify its behavior. The personality can prove correct behavior by additionally monitoring the log server and thus persisting the signed tree heads. If it changes retroactively, the personality can detect that.

5.2 Performance Evaluation

To evaluate the performance of our transparency log system, we used the metadata of all publicly available APK files in the official F-Droid repository and created the corresponding log entries in our system. To perform that evaluation, we wrote a Python script that first fetches the current F-Droid index file[8] of the

[8] https://f-droid.org/repo/index-v2.json (accessed: 2023-02-07)

official repository. The second step is to parse the index file in order to prepare the proper log format required by our personality. Next, the script requests an access token for the buildserver user and starts to send the POST requests to create the new log entries.

For our performance measurements we used a computer with an Intel i7-1185G7 @ 3.00 GHz CPU and 32 GB of RAM for fetching the current F-Droid index file of the official repository, to prepare the log entries and to send the POST requests to our transparency log backend. Our transparency log backend, including the personality and the log itself, is deployed on a virtual machine with 2 cores and 2 GB RAM (Host CPU: Intel E5-2620 v3 @ 2.40 GHz). For the client side end-to-end verification we used an Android emulator running API level 31 with 1536 MB RAM.

There are 9705 APK files in the official F-Droid repository. It took less than 47 minutes to create our log, less than 30 ms on average per APK. The log database required 8 MB of disk storage. Inclusion proofs consist of 14 hashes (825 B) for the first leaf and of 7 hashes (510 B) for the last leaf. Consistency proofs similarly ranged from 14 hashes (821 B) to 8 hashes (533 B). An end-to-end verification with our Android library of the first leaf that requires the maximum amount of intermediate node hashes in that particular tree took 296 ms.

6 Open Research Questions

We have introduced a novel concept to mitigate necessary trust in mobile app distribution systems, especially with focus on digital signatures on APK files. Our current approach includes mitigation techniques, but does not get rid of trust anchors completely. Therefore, we are looking for a solution to remove such trust anchors completely by extending our transparency log system in a way that still meets the same security requirements (e.g. integrity and authenticity checks) as digital signatures. Another open research question exists around third-party libraries in apps. More precisely, we plan to enhance our transparency log system so that it can also detect outdated or compromised third party libraries even in obfuscated APK files.

7 Related Work

7.1 Certificate Transparency

Certificate Transparency (CT) [10] is a process that is part of the web's public key infrastructure. Its main purpose is to detect unauthorized or even maliciously issued TLS certificates for websites by making them transparent and verifiable. Whenever a certificate authority (CA) issues a new certificate, a new entry gets recorded in one of the approved verifiable logs. These logs can be checked for suspicious behavior by independent monitors. As these logs are publicly auditable, interested parties are able to create such a monitor. In that case the browser is one of the possible auditors to verify if the particular certificate is part of the verifiable log.

Difference: Both, CT and our proposed system are based on Merkle trees. Therefore, we can use the same underlying Google Trillian implementation to handle the tree structure. However, data stored within the tree is application-specific as CT needs to store TLS certificate information and our system deals with information about mobile apps. The most relevant difference is the end-to-end verification. In the CT ecosystem, the browser is responsible for verifying if the certificate is properly logged by checking the signed certificate timestamp (SCT), e.g., the X.509v3 certificate extension [21]. Our proposal, on the other hand, does not need additional information, like an SCT on the client side as the end-to-end verification is directly performed with the transparency log system. At this point we do not rely on digital signatures as our end-to-end verification implementation crafts the expected log entry at the beginning of the verification stage and directly verifies if a corresponding log entry is available or not.

7.2 Binary Transparency in F-Droid

F-Droid has already incorporated a module [25] that logs the signed app index metadata files in append-only storage. These files contain information about the available APKs of a specific F-Droid repository so that every update or change also requires a change on the related file. To fulfill the requirement of append-only storage, F-Droid uses a git repository that it claims is tamper proof. This approach allows interested parties to verify if a specific binary was published by the expected publishing entity as only an authorized party is allowed to push to the respective git repository. As of the time of writing this paper, this feature is activated for the Guardian Project repository.

Difference: A git repository is a content-addressable filesystem [4] that is based on a Merkle tree—the same data structure we use in our approach and prototype implementation. If a new or updated file is stored in a git repository, git calculates the SHA-1 hash based on the file's content (called a *blob* object) and stores this information in an internal object database. A blob does not store the filename itself. Instead, we store the names of files in a *tree* object; tree objects may contain tree objects. This approach is analogous to the Unix file system, where a blob object would correspond to the data associated with a file object, while a tree corresponds to the entries found in a directory object.

The logging approach by F-Droid stores the app index metadata file in a specific git repository that is responsible for version control. This approach is not scalable as metadata of all the available apps in the F-Droid ecosystem is stored in one single file. Furthermore, the information required to carry out the verification task is not directly managed by the tree structure, because it is just stored in a file, and thus the verification does not benefit from optimizations of a tree structure like efficient searching. Consequently, if an entity wants to verify if a specific value is part of the tree, the whole file must be downloaded by the client. This file is also larger than necessary for verification since it contains information that is not relevant for the verification task at all (e.g. the applied license). While such an approach may work reasonably well for F-Droid, this approach

would not scale to the distribution scale seen in larger markets. In contrast, our approach is scalable since we use Merkle trees directly and make efficient use of communication and computation effort through the direct provision of consistency proofs over time with snapshots. We do not require the use of a full data structure at once.

7.3 Blockchain

Blockchains are built around a distributed public ledger that provides similar properties to our approach, including an append-only data structure, tamper resistance, and transparent verification. The ledger contains blocks that consist of a hash of the previous block, a timestamp, and the transaction data.

The problems tackled by using a blockchain are orthogonal with regards to authentication, integrity, and non-repudiation [7] that can be addressed by using digital signatures. A digital signature can prove that signed data has not been tampered with afterwards and that it is signed by an entity that possesses the respective signing key. However, for example, the time of signing requires trust in the signing party that is essential for financial transactions or legal contracts. To address this trust dependency, a blockchain uses a distributed trust mechanism, where interested parties can store a list of transactions and thus are able to verify that they have not been tampered with.

Difference: From a security point of view, blockchains fulfill requirements like tamper protection as well. However, their verification procedure is not scalable[9]. For a full end-to-end verification, a blockchain based approach requires the clients to download the whole chain whereas the transparency log approach only requires the hashes of the audit path ($log(n)$) and ideally checkpoints signed by independent witnesses.

8 Conclusion

Current mobile app distribution systems use digital signatures to ensure integrity and authenticity of their apps. However, as shown in this paper, there are realistic threats which may compromise digital signatures. For example, it is currently impossible to detect unauthorized usage of signing keys. A more general perspective on this kind of problem is how to compliment or enhance the trust placed on digital signatures in mobile app distribution systems.

This paper introduces a novel concept to mitigate threats found in mobile app distribution systems by making any distribution attempt transparent and thus verifiable. Additionally, a prototype has been implemented to prove the practicability and feasibility of the design proposal, including a detailed security

[9] In terms of efficiency comparison, we are not even assuming proof-of-work consensus algorithms, but permissioned ledgers comparable to the authentication of submitters performed by the personality.

evaluation of the newly added components as well its performance. Our evaluation shows that an attacker would have to compromise all the involved system components and security controls to successfully distribute a malicious APK file without detection. While the proposed system focuses on mobile app distribution systems, it can also be applied in other scenarios where digital signatures are used and may not be trustworthy.

Acknowledgement. This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World and has partially been supported by the LIT Secure and Correct Systems Lab. We gratefully acknowledge financial support by the Austrian Federal Ministry of Labour and Economy, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, Österreichische Staatsdruckerei GmbH, and the State of Upper Austria.

A Availability

Our prototype implementation consists of the following component repositories and is publicly available.

- **F-Droid server:** The relevant source code segments have been extracted from the fork of the official F-Droid server code. Source Code: `https://github.com/mobilesec/fdroidserver_transparencyextension`
- **Personality:** The personality project contains the code for the application-specific interface between the client library, the F-Droid server, and the Google Trillian logging infrastructure. Source Code: `https://github.com/mobilesec/mobiletransparency-personality`
- **Android library:** The Android library project contains the code for the end-to-end verification of APK files. Source Code: `https://github.com/mobilesec/mobiletransparency-androidlibrary`
- **Evaluation setup:** Contains the test script, configuration file and reference data of our performance evaluation. Source Code: `https://github.com/mobilesec/mobiletransparency-data`

We also provide a running personality with this version of the codebase along with a transparency log running the unmodified Google Trillian case (from `https://github.com/google/trillian`) that has been pre-filled with APK metadata from the index of the official F-Droid repository as well as some of our test apps using the Android library for verification. It is available through a Tor Onion service at `http://madtl6agno7zze4ll66ylxmb4lkmb72attwfhcmfbspyx35v4e6ut5ad.onion/Log/ListTrees`.

References

1. Aryan, S., Aryan, H., Halderman, J.A.: Internet Censorship in Iran: A First Look. In: 3rd USENIX Workshop on Free and Open Communications on the Internet (FOCI '13). USENIX Association, Washington, DC, USA (Aug 2013), https://www.usenix.org/conference/foci13/workshop-program/presentation/aryan
2. Barrera, D., McCarney, D., Clark, J., van Oorschot, P.C.: Baton: Certificate Agility for Android's Decentralized Signing Infrastructure. In: WiSec '14: Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks. pp. 1–12. ACM, Oxford, United Kingdom (Jul 2014). https://doi.org/10.1145/2627393.2627397
3. Basin, D., Cremers, C., Kim, T.H.J., Perrig, A., Sasse, R., Szalachowski, P.: ARPKI: Attack Resilient Public-Key Infrastructure. In: CCS '14: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 382–393. ACM, Scottsdale, AZ, USA (Nov 2014). https://doi.org/10.1145/2660267.2660298
4. Chacon, S., Straub, B.: Pro Git. Apress, Berkeley, CA, USA, second edn. (2022), https://git-scm.com/book/en/v2
5. Coufalíková, A., Klaban, I., Šlajs, T.: Complex strategy against supply chain attacks. In: 2021 International Conference on Military Technologies (ICMT). pp. 1–5. IEEE, Brno, Czech Republic (2021). https://doi.org/10.1109/ICMT52455.2021.9502768
6. Cutter, A., Drysdale, D.: Trillian Personalities (2022), https://github.com/google/trillian/blob/05001d1876f9340e42ba8b839c94e1b79246207b/docs/Personalities.md
7. Di Pierro, M.: What Is the Blockchain? Computing in Science & Engineering **19**(5), 92–95 (2017). https://doi.org/10.1109/MCSE.2017.3421554
8. Eijdenberg, A., Laurie, B., Cutter, A.: Verifiable data structures (2015), https://github.com/google/trillian/blob/30160804ab5203cde4412fe26f55a4149112bd92/docs/papers/VerifiableDataStructures.pdf
9. F-Droid: Docs – F-Droid – Free and Open Source Android App Repository (2023), https://f-droid.org/docs/ (accessed: 2023-01-23)
10. Google: Certificate Transparency (2023), https://certificate.transparency.dev/ (accessed: 2023-01-23)
11. Google: How Log Proofs Work – Certificate Transparency (2023), https://sites.google.com/site/certificatetransparency/log-proofs-work (accessed: 2023-01-23)
12. Google: Use Play App Signing – Play Console Help (2023), https://support.google.com/googleplay/android-developer/answer/9842756 (Accessed: 2023-01-12)
13. Herr, T., Loomis, W., Scott, S., Lee, J., Schroeder, E.: Breaking Trust – Shades of Crisis Across an Insecure Software Supply Chain (Feb 2021), https://www.usenix.org/conference/enigma2021/presentation/herr
14. Kumar, R., Virkud, A., Raman, R.S., Prakash, A., Ensafi, R.: A large-scale investigation into geodifferences in mobile apps. In: Proceedings of the 31st USENIX Security Symposium (USENIX Security '22). pp. 1203–1220. USENIX Association, Boston, MA, USA (Aug 2022), https://www.usenix.org/conference/usenixsecurity22/presentation/kumar
15. Laurie, B., Langley, A., Kasper, E.: RFC 6962: Certificate Transparency (2013). https://doi.org/10.17487/RFC6962
16. Laurie, B., Messeri, E., Stradling, R.: RFC 9162: Certificate Transparency Version 2.0 (2021). https://doi.org/10.17487/RFC9162

17. Mayrhofer, R., Stoep, J.V., Brubaker, C., Kralevich, N.: The Android Platform Security Model. ACM Trans. Priv. Secur. **24**(3) (Apr 2021). https://doi.org/10.1145/3448609
18. Meiklejohn, S., Kalinnikov, P., Lin, C.S., Hutchinson, M., Belvin, G., Raykova, M., Cutter, A.: Think Global, Act Local: Gossip and Client Audits in Verifiable Data Structures. Computing Research Repository (CoRR), arXiv:2011.04551 (2020). https://doi.org/10.48550/ARXIV.2011.04551
19. Melara, M.S., Blankstein, A., Bonneau, J., Felten, E.W., Freedman, M.J.: CONIKS: Bringing Key Transparency to End Users. In: Proceedings of the 24th USENIX Security Symposium (USENIX Security '15). pp. 383–398. USENIX Association, Washington, DC, USA (Aug 2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara
20. Merkle, R.C.: A Digital Signature Based on a Conventional Encryption Function. In: Pomerance, C. (ed.) Advances in Cryptology — CRYPTO '87, LNCS, vol. 293, pp. 369–378. Springer, Berlin, Heidelberg (1988). https://doi.org/10.1007/3-540-48184-2_32
21. Mozilla: Certificate Transparency (2023), https://developer.mozilla.org/en-US/docs/Web/Security/Certificate_Transparency (accessed: 2023-01-23)
22. Nicas, J., Zhong, R., Wakabayashi, D.: Censorship, Surveillance and Profits: A Hard Bargain for Apple in China. The New York Times (May 2021), https://www.nytimes.com/2021/05/17/technology/apple-china-censorship-data.html (accessed: 2023-01-23)
23. Nikitin, K., Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Gasser, L., Khoffi, I., Cappos, J., Ford, B.: CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In: Proceedings of the 26th USENIX Security Symposium (USENIX Security '17). pp. 1271–1287. USENIX Association, Vancouver, BC, Canada (Aug 2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/nikitin
24. Nordberg, L., Gillmor, D.K., Ritter, T.: Gossiping in CT. Internet-Draft draft-ietf-trans-gossip-05, Internet Engineering Task Force (Jan 2018), https://datatracker.ietf.org/doc/draft-ietf-trans-gossip/05/, work in Progress
25. Steiner, H.C.: Binary Transparency Log for https://guardianproject.info/fdroid (2023), https://github.com/guardianproject/binary_transparency_log (accessed: 2023-01-23)
26. Syta, E., Tamas, I., Visher, D., Wolinsky, D.I., Jovanovic, P., Gasser, L., Gailly, N., Khoffi, I., Ford, B.: Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 526–545. IEEE, San Jose, CA, USA (May 2016). https://doi.org/10.1109/SP.2016.38
27. The MITRE Corporation: Supply Chain Compromise (2023), https://attack.mitre.org/techniques/T1195/ (accessed: 2023-01-23)
28. The Tor Project: Tor Project – Anonymity Online (2023), https://www.torproject.org/ (accessed: 2023-02-07)