Firmware Transparency: Strengthening Security Guarantees Against Bootloader-Targeting Attacks

Mario Lins[1[0000−0003−1713−3347]], René Mayrhofer[1,2[0000−0003−1566−4646]], and David Zeuthen[2[0009−0001−1445−0403]]

[1] Johannes Kepler University Linz, Institute of Networks and Security, Linz, Austria
{lins,rm}@ins.jku.at
[2] Google, USA

Abstract. Trust constitutes an essential requirement with regards to security considerations across numerous systems. One critical area where trust plays a significant role is the booting process of mobile devices, which relies on a set of components commonly referred to as the "chain of trust". Most mobile device manufacturers implement a range of security controls, such as tamper-resistant hardware or digital signatures, to protect the trust chain and ensure the overall trustworthiness of the device. Digital signature keys used to verify the chain of trust are typically set by the device manufacturer, with most of them prohibiting users to set a custom root of trust (RoT). Currently, Google is the only major manufacturer that allows users to set a custom RoT on Pixel devices. However, there are legitimate use cases—like using an alternative operating system—where the chain of trust would be broken without setting the proper custom RoT. In this paper, we address compromised security of a broken chain of trust resulting from the necessity of completely disabling key verification on devices that lack support of a custom RoT. To mitigate these lost security guarantees, we propose a new security mechanism, called *Firmware Transparency*. Furthermore, we demonstrate that *Firmware Transparency* enhances verifiability on devices with a locked bootloader. We prove the security properties by formal model and verify our proposed protocol and implement an open-source prototype, demonstrating its practicability and feasibility. Moreover, the proposed concept demonstrates its effectiveness by detecting a malicious modification resulting from the exploitation of a vulnerability that is not detected by current security controls.

Keywords: Supply Chain Security · Mobile Security · Attestation · Transparency log · Android

1 Introduction

"A Digital Prison - Surveillance and the suppression of civil society in Serbia" is a report published by Amnesty International [3], telling the story of an independent journalist who was brought to a police station following a traffic stop. During his time at the station, he was asked to leave his phone at the reception. After his release, the journalist noticed unusual behavior on his phone and

contacted the Amnesty International's Security Lab for assistance. Upon some forensic analysis, the lab discovered that the phone had been unlocked without asking the journalist for any credential to extract sensitive information, and they found traces indicating malicious modifications to the mobile device.

This is just one example highlighting the importance of security and consequently integrity of mobile devices meanwhile used to process sensitive information in our professional and private daily life. Mobile devices are already an integral component providing new opportunities in our interconnected digital landscape for everyone including ordinary citizens, employees, journalists, and many others. This is also evidenced by common development strategies employed by companies and even governments, which involve gradually integrating more mobile services and extending our usage of these devices to process sensitive data. Representative examples are apps for managing digital credentials, like "BC Wallet" [8] (provided by the government of British Columbia), apps for common citizen services, like "Digitales Amt" [9] (provided by the Austrian government), or apps for providing (physical) identity documents like the mobile driver's license app [10] used in California.

One significant aspect of today's mobile-security landscape is the supply chain used for distributing apps and firmware images on our mobile devices. Device manufacturers, software distributors, developers, and researchers around the world are increasingly focusing on utilizing security controls to prevent unauthorized tampering with device and the software running on them. Digital signatures and hardware-backed key stores are two prominent examples of such security measures. Security controls in the mobile ecosystem are implemented across multiple layers, including hardware, firmware, and the application layer, with each layer addressing specific security objectives. Our research focuses on the firmware layer, assuming that the hardware layer is trustworthy. Consequently, the security mechanisms of the hardware layer are considered beyond the scope of this work, as they have been extensively studied in other research. We choose Android due to its open-source nature and widespread adoption across different manufacturers. However, our proposed concept is versatile and could be adapted for use on other embedded devices as well.

One of the most relevant security controls in Android for ensuring device integrity, particularly at the firmware layer, is Android Verified Boot (AVB). AVB ensures the integrity and authenticity of the components involved in booting the OS. To achieve these security objectives, AVB requires the bootloader to operate in a specific state, called *locked* state. However, there are legitimate cases where an unlocked bootloader is intended, such as using a custom OS. Allowing users to flash a (trustworthy) custom OS can even enhance security [33], particularly on older devices where manufacturers no longer provide security updates [1], whereas alternative images, such as LineageOS [38], may incorporate them.

In this paper, we make the following contributions:

1. We present a new security mechanism, called *Firmware Transparency*, built on transparency logs and designed to compensate lost security guarantees

of an unlocked bootloader by introducing new verification capabilities with only minor limitations that we openly address.
2. Furthermore, we argue that *Firmware Transparency* further enhances verifiability on locked devices that maintain a functional chain of trust.
3. We implement an open-source prototype (see Appendix A) to demonstrate the practicability and feasibility.
4. We demonstrate the effectiveness of *Firmware Transparency* by detecting a malicious modification resulting from the exploitation of a vulnerability that was not detected by current security controls.
5. We formally verify the underlying protocol to demonstrates that our concept provides essential security guarantees based on the considered threat model.

2 Preliminaries

The following sections provide an overview about relevant components involved in the Android boot process and a brief analysis of the security considerations that are relevant for our proposed approach.

2.1 Android Bootloader

The boot process of Android involves multiple components [35, 37]. The first component getting active when pressing the power button, is called the *Primary Boot Loader (PBL)*. The PBL is responsible for loading the *eXtensible Boot Loader (XBL)* into memory and typically implemented on a tamper-resistant chip by the chipset manufacturer. Thus, we assume this component to be trustworthy [29].

Compared to the PBL, the XBL (formerly Secondary Boot Loader (SBL)) can be customized by the specific device manufacturer and is responsible to load the next bootloader stage, called *Android Boot Loader (ABOOT)*.

The main purpose of ABOOT is to load the kernel that initializes the OS. Furthermore, it implements an interface to run custom commands directly on the bootloader stage of a device used, for example to flash a new image on a specific partition or perform recovery activities. This bootloader part and especially the implemented interface is different depending on the manufacturer. As an example: Google calls this interface *fastboot* [36], while on Samsung devices this interface is called *ODIN* [39].

2.2 Android Security Controls

Google provides several security controls to mitigate tampering attempts, as outlined by "The Android Platform Security Model" by Mayrhofer et al. [29]. These controls address runtime-security considerations, including *Mandatory Access Control* or the *Trusted Execution Environment* (TEE), but also deal with booting up the OS securely through methods such as *Android Verified Boot* (AVB).

Android Verified Boot (AVB) AVB is used to protect the system against malicious tampering attempts by verifying executable code and relevant partitions (e.g., boot, system, vendor, OEM partitions) used to boot up the OS (see Fig. 1). This security control verifies the chain of trust starting from the hardware-protected PBL stage to the XBL and ABOOT stage ending up with the verification of pertinent partitions. As stated in Section 2.1, we consider the chain of trust starting at the PBL up to (but not including) the ABOOT stage to be trustworthy. The earliest stage at which the chain of trust may be customized is the ABOOT stage by setting a custom RoT. This custom RoT is used to sign the expected hash value of the relevant partitions, that are either stored on the specific partition itself or on a dedicated one called *vbmeta*.

The vbmeta struct contains data, like hashes or hash trees in case of larger file systems, that are used by AVB for verification purposes. For example, it contains the hash of the *boot* partition and hash trees representing the *system* and *vendor* partitions. Furthermore, the vbmeta partition also includes an index used to prevent rollback attacks (e.g., flashing an older and potentially vulnerable system image on a device). The main partition used for storing the vbmeta struct is called *vbmeta*. However, additional *vbmeta* partitions (e.g., *vbmeta_vendor*) used to store vendor specific information are also possible. The term *vbmeta digest* refers to a digest including all vbmeta structs, including the root and vendor specific vbmeta partitions. It can be used to verify the authenticity and integrity of the vbmeta structs and additionally, it is also embedded in the hardware-based attestation data so that it can be queried for verification even if the OS is not trustworthy.

AVB defines four possible device states – *GREEN*, *YELLOW*, *ORANGE*, and *RED*. Device states are used to indicate if a user can flash a custom firmware and whether the verification controls are enforced by AVB. The *GREEN* state

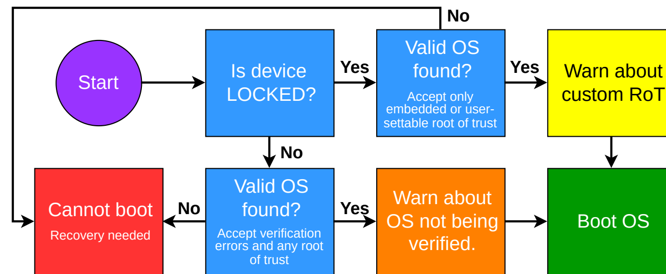


Fig. 1. Simplified Android verified boot flow based on [15]

indicates that the device is locked without a custom RoT. It includes all available verification steps and requires the user to not set a custom RoT, which means that the verification is performed by using original RoT set by the manufac-

turer. The *YELLOW* state indicates a locked device with a custom RoT. The availability of this stage depends on whether the manufacturer permits setting a custom RoT. Currently, Google is the only major manufacturer that allows users to set a custom RoT on Pixel devices. The verification part of the *YELLOW* state is equal to the one used in *GREEN* state as a RoT is present and thus the integrity of the partitions can be verified by the respective kernel module. The most relevant state for this paper is *ORANGE* where the bootloader is *unlocked*. An unlocked bootloader allows the user to flash arbitrary images on the partitions and furthermore it disables certain verification steps. This state does not require the OS to be signed by any RoT. The *RED* state indicates errors that prevent booting the OS.

At the time of writing this paper, there was just one mobile device family allowing to set a custom RoT (*YELLOW* state), namely Google Pixel. Other manufacturers like Motorola, Nokia, Samsung, or Fairphone [13] do not allow setting a custom RoT, but do also not fully prevent users to flash custom firmware on the device (*ORANGE* state). Hence, if a user wants to flash custom firmware on a device other than Google Pixel, they must unlock the bootloader, and once unlocked, they cannot re-lock it again with the custom firmware due to manufacturer restrictions, which do not allow setting a custom RoT. This is a significant disadvantage for this particular user group as evidenced by a security- and privacy-focused alternative Android fork called GrapheneOS [21], which does not even support other devices than Google Pixel due to exactly these reasons. We believe that nowadays, users should have full ownership and, consequently, control over their devices whenever feasible with regards to security, and the freedom of choice to determine which OS they want to run on their hardware. Thus, we focus our research on compensating security controls to overcome lost security guarantees of a device that is in the *ORANGE* state. However, our security concept can also be used in other states to include an additional layer of security in terms of transparency and verifiability from a third-party point of view.

StrongBox Android StrongBox is a dedicated hardware security module (HSM) with its own processor and memory, either implemented through custom hardware or applets on a generic secure element (SE). We consider StrongBox, to be secure and functioning as expected, as it leverages the security guarantees provided by the hardware layer. Furthermore, we assume equal security guarantees when comparing it to a Trusted Platform Module (TPM) that is considered a standard component in server and desktop systems. Thus, we believe that our concept is also applicable to other systems that use a standard TPM.

Android Attestation Android attestation is a foundational trust anchor for AVB, as it produces integrity-protected evidence about a device's state. In particular, the resulting attestation certificate includes, for example, the boot state, the verified boot key, and the verified boot hash—details that are especially relevant to our proposal. When attestation is enforced by the device's TEE

or StrongBox, it is hardware-backed (see [17]): relevant information is gathered or generated by code running in the secure environment that is isolated from the Android platform, making the results non-forgeable by an adversary as long as the secure hardware remains trustworthy—an assumption in our threat model. Consequently, our system requires TEE- or StrongBox-backed attestation; software-only attestation is excluded because it relies on a locked bootloader, which we do not assume. Further details on how we leverage Android attestation are provided in Section 3.5.

2.3 Transparency Logs

A transparency log [12] (also referred to as *verifiable log*) is an append-only data structure that incorporates cryptographic controls to prevent retroactive insertions, deletions, or modifications of its records. Operators mainly control insertion of new entries (i.e., authenticating append operations), but we generally assume global read-only access (i.e., unauthenticated read operations). Therefore, clients can verify proper behavior independently without trusting the operator, though availability remains the operator's responsibility.

An illustrative example [28] of utilizing a transparency log in a mobile device environment is the detection of unauthorized distribution attempts in mobile app distribution systems. An adversary with access to a signing key could try to distribute a malicious update of an app that seems authentic to its users. However, when storing every distributed app version in a transparency log—including the APK hash—the user or the app store client can verify whether a record for that version is available in the log or not. If not, it might indicate an unauthorized distribution attempt. We consider the use of such logs for app distribution orthogonal to the security guarantees addressed in the present paper.

In this work, we use the same building block of a transparency log, which is based on a binary Merkle tree [30]. The verification is performed using two cryptographic proofs: inclusion and consistency proof. The inclusion proof allows the verifying party to prove if a particular leaf in the Merkle tree exists. The consistency proof is used to verify the append-only property. In case an adversary somehow managed to retroactively modify an already existing entry, the consistency proof would reveal that. More details about cryptographic proofs, the structure and components of a transparency log can also be found in [12,16,30].

3 Firmware Transparency

This section presents the threat model, the architecture of *Firmware Transparency* and its components, and key implementation details of our prototype.

3.1 Setting the Stage

We assume the ABOOT bootloader to be trustworthy, given that it is flashed by the device manufacturer and verified by AVB. Our assumption is further supported by the fact that the reliability of the AVB verification process depends

on the security and integrity of the RoT. A common mitigation technique used to secure the RoT is to store the associated private key on a secure element in hardware — and to apply best practices of secure software development to mitigate the risk of security critical bugs in these fundamental components. This makes a tampering attempt unlikely, as an adversary would need to either compromise the HSM or discover a vulnerability on bootloader level (which, while not impossible [23], is rare in practice because of the limited amount of code and slower change rate). Therefore, we put our focus on the last stage of the booting process, where the ABOOT component verifies relevant partitions (e.g., boot). Furthermore, this stage signifies the first encounter with potentially user-specific code as it initializes the OS, which could be an alternative one rather than the stock OS. A possible strategy used to ensure verifiability of the chain of trust in this stage is to allow the user to set a custom RoT. AVB uses this key to continue the verification process by loading related hash values from the *vbmeta* struct, verifying the authenticity with the custom RoT, and comparing the loaded hash values with the actual values of the respective partitions (the *YELLOW* state as described above). This is the point where our research addresses a critical challenge: enabling device owners to flash an arbitrary OS without relying on the manufacturer, who may restrict the ability to set a custom RoT for verification. As a result, we cannot assume a secure device state.

3.2 Stakeholders

We introduce two essential roles: the *auditee* and the *auditor*. The primary target group consists of users who try to make use of long term updates due to sustainability reasons, value their freedom of choice, and require new approaches of end-to-end verification on mobile device with an unlocked bootloader. We refer to this type of devices as *auditee*. On the other hand, we require a trusted party responsible for verifying the integrity of the firmware according to the information provided by the auditee. We refer to this party as the *auditor*.

3.3 Threat Model

In this paper we focus on devices with an unlocked bootloader to allow users to flash a custom image on their device without sacrificing relevant security guarantees. Thus, the main objective of *Firmware Transparency* is to mitigate lost security guarantees on devices with an unlocked bootloader by making tampering attempts transparent and thus verifiable. We do not consider confidentiality nor availability as security objective of *Firmware Transparency* as we focus primarily on the integrity and authenticity of the device.

Adversary Model The following list outlines relevant adversary models to the scope of our research.

- A1 **Physical adversary:** Adversary with physical access to the device and capable of connecting the device to development machines (e.g., via ADB).

A2 **On-path adversary (OPA):** Adversary with access to the distribution network (e.g., via machine-in-the-middle attack) and capability to modify images distributed to their users (typically via HTTPS download).

A3 **Insider adversary:** Insider adversary with access to the OEM infrastructure and potentially also to the signing key(s) of images. Physical attacks (e.g., extracting the signing key from a hard drive within the OEM infrastructure) are also covered by this adversary model.

Threats We present threats relevant to mobile devices with an unlocked bootloader and considered for designing *Firmware Transparency*. Furthermore, we take into account existing verification capabilities that users may have already incorporated in their mobile device.

T1 **Evil maid attack:** An adversary exploiting physical access to a device can flash arbitrary images on the device and—as the bootloader is unlocked—the verification errors are not shown. The user is not aware of image modifications after their own (presumably intentional) unlocking of the device bootloader and flashing of a custom image. This also involves tampering attempts similar to those experienced by Amnesty International journalists [3].

T2 **On-path compromise:** An OPA with access to the distribution channel can tamper the image before it gets flashed to the device. The user could verify the signature and would be able to detect unauthorized modification. However, once a modified image is flashed to the device with an unlocked bootloader, AVB does not verify the authenticity and integrity of the respective partitions. Hence, a user would not detect that the booted image has been modified.

T3 **Insider compromise:** An insider adversary with access to the OEM infrastructure and potentially the signing key could manipulate the source code and sign the artifact or directly replace the artifact and sign it. Unlike the distribution threat, the user cannot detect unauthorized image modifications since it is properly signed with the manufacturer's official key. This threat also applies to the common verification method with a locked bootloader. Since the tampered image would be signed properly, the current verification controls would not detect unauthorized modification even if the bootloader gets locked again and even without using a custom RoT.

T4 **Stealing knowledge factor:** An adversary exploiting physical access, such as an evil maid attack on a device with an unlocked bootloader might gain access to the knowledge factor provided by the user if no verification possibility is in place beforehand.

T5 **Relay attack:** An adversary may relay verification request to a legitimate device and response with these results to the verifier. The verifier requests some form of proof of the user's mobile device, which may already be under the control of the adversary. If the adversary can relay the verification request to a legitimate device to obtain a valid proof, it may appear as an authentic and legitimate audit result.

- T6 **Replay attack:** An adversary may replay an already created legitimate verification result. This threat addresses the freshness property of a potential verification result. If it is possible to replay older verification results a tampered device can easily provide them to the verifying party.
- T7 **Distribute different versions:** A malicious distributor can provide a tampered version of an image only to a subset of users. This threat also addresses distributors who are forced to publish a different version for specific users due to pressure from a nation state actor. An example of such enforcement is the Internet censorship regime of Iran [5] or a study [25] that revealed geographical differences in 596 mobile apps.

3.4 Architecture

Our architecture comprises five primary components as illustrated in Fig. 2: (1) the developer publishing a firmware image, (2) the device of the user with that custom firmware, (3) the firmware distributor, (4) one of more auditors, and (5) the backend for the transparency log infrastructure. As the auditee could already be malicious, the auditor also has to verify the authenticity and integrity of the information given by the auditee. To prevent relay attacks (T5), we incorporate the *Trust on First Use* (TOFU) model. In particular, it requires the auditee to generate a persistent key that is stored in the hardware-backed keystore [3], enabling the auditor to use a strong device binding verification on subsequent audits. As this persistent key is stored in the hardware-backed keystore, the OS does not have access to it even if it had been compromised. The auditor app is assumed to be inherently trustworthy. Since our app can run on an arbitrary device for verification, targeted attacks on the auditor become much less likely to succeed. In scenarios where an adversary compromises the build infrastructure, we recommend utilizing Attestable Builds [24] to ensure strong source-to-binary correlation. Furthermore, in cases where the app's distribution could be compromised (e.g., signing key is lost), we suggest leveraging additional verification mechanisms, such as those described by the authors of *Mobile App Distribution Transparency* [28]. By utilizing a transparency log we mitigate threat T7 as the available version names including their hashes can be publicly verified. We assume that the transparency log is protected against split-view attacks by integrating a sufficient amount of independent witnesses.

3.5 Design and Implementation Details

As the bootloader is unlocked, and we do not have a custom RoT (ORANGE state), AVB will have several verification errors, but continues to boot the custom image because the device is unlocked. Once the device is booted, the user can verify the authenticity and integrity of the firmware using our auditor app.

[3] Our protocol relies on hardware-backed keystores; while Strongbox offers the strongest security guarantees, our protocol also works on devices using TEE.

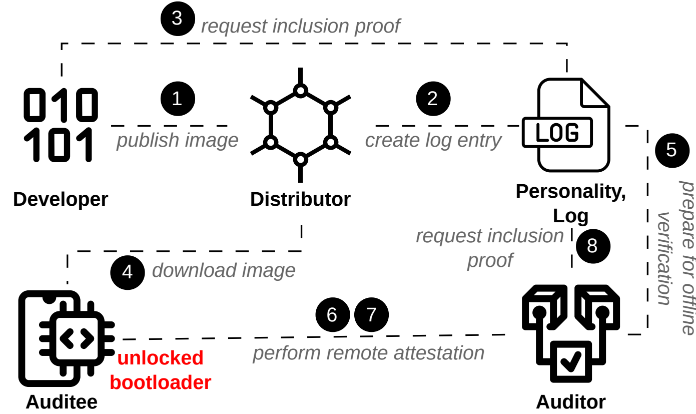


Fig. 2. Architecture of *Firmware Transparency*. **(1)** First, the developer uploads the image to the distributor's infrastructure. **(2)** The next step requires the distributor to create the respective log entry before finally distributing the image to its users. **(3)** To ensure that the distributor is not malicious, the developer verifies whether the log entry is available and that it matches the expected log entry in alignment with the image. **(4)** At this point the user can download the firmware and flash it on the device ending up with an unlocked bootloader and no custom RoT to verify the newly flashed partitions. **(5)** (optional) There are cases (e.g. no internet connection) where the audit itself cannot be performed because the auditor cannot request an inclusion proof from the transparency log. In such cases it would be possible to store only relevant audit paths (rather than the entire Merkle tree) of common firmware images (e.g., LineageOS) on the auditor device to enable offline verification. **(6)** The first audit is crucial as we leverage the TOFU model and store a strong device binding property on the auditor device. This security model is required to prevent T5. **(7)** Subsequent audits can be performed on top of strong device binding verification. **(8)** Verification of the provided data is done by involving our tamper-proofed transparency log, which contains the legitimate information of the current firmware version.

This is possible even on top of lock screen and before first unlock, ensuring protection of the user's knowledge factor and the user data partition. Our prototype implementation includes an application-specific API, called *personality* that is mainly responsible for interacting with the Google Trillian backend, and the auditor app that can be started either in *auditee* or *auditor* mode.

Auditor. The auditor app runs on both the auditee and the auditor side. On the auditee side, the app is responsible for gathering relevant information (e.g., the vbmeta digest) and making it available to the auditor for the verification task. Android includes the vbmeta digest in the hardware key attestation and thus, the auditee side of the app can use Android key attestation to retrieve a signed copy of the digest. The communication between the auditee and the auditor is established via QR code scanning. The auditee generates a QR code

including the attestation details necessary for the verification. The auditor scans the QR code and requests an inclusion proof to verify whether the information has been properly logged. If the verification is successful so that it is proven that the calculated root node hash based on the inclusion proof is equal to the expected root node hash, the auditor shows a green checkmark to the user – mitigating T1, T2, T3. To prevent relay attacks (see threat T5), we use a TOFU model, as in SSH, in which the auditor stores a tamper-resistant device binding to the auditee's device. Subsequent audits must include the TOFU credential.

Personality. The *personality* is the application-specific interface that acts with the transparency log. It is responsible to ensure consistency of newly added log entries. Additionally, it can also be used as authentication and authorization control to prevent unauthorized access to certain tasks, like creating new log entries. For our prototype implementation we have decided to restrict access for creating and deleting entries as we would like to keep the log small. Therefore, we incorporated a token-based authentication scheme. However, theoretically access to a transparency log does not have to be restricted as everyone can verify its content at any time. Based on the authorization model, our *personality* is divided in the following three areas: The **(1)** Public Area that provides the API endpoints for requesting the latest signed root node and cryptographic proofs, like inclusion and consistency proofs. There is no prior authentication required as this is publicly accessible. The authentication itself is also part of this area and provides a bearer access token in case the authentication is successful. The **(2)** Log Builder that includes the endpoint to create new log entries. This endpoint requires prior authentication and authorization for creating new entries. The **(3)** Admin Area that is used for administrative tasks, like creating and initializing the tree. This is only allowed with proper access tokens that contain the admin role. The personality is written in Rust as it provides additional security features (e.g., typestate pattern [7]). The Google Trillian implementation provides **.proto* files that we use to enable interaction with the log server via gRPC. An essential implementation detail with regards to the Merkle tree, is to consider special prefixes depending on the type of the specific tree node. For example, we use 0x00 for leafs and 0x01 for intermediate nodes.

Transparency Log. The incorporation of a transparency log mitigates threat T7 by forcing the distributor to create a log entry and enable the auditor to verify the corresponding inclusion proof. Additionally, it allows a developer to monitor the log to detect if someone else tries to distribute a spoofed firmware version. Typically, a transparency log is publicly accessible, allowing everyone to request the required cryptographic proofs. Furthermore, a log can also be hosted by everyone as it just requires one party to actually create new log entries. However, from a practical standpoint, we believe it is more feasible for the log to be hosted by the distributor – such as LineageOS – that distributes the firmware images. As the transparency log provides sufficient cryptographic protocols, any malicious attempt of the log operator is detectable. We assume that a sufficient number of independent witnesses are available to prevent split-view attacks where a malicious log operator provides different versions of the log to different users. For

our prototype, we base our implementation on Google Trillian, which provides the basic operations required to manage one or even multiple Merkle Trees.
Data structure. The data structure we use to store log entries contains the **version** information and a **hash** value. The version field is based on the Android build number, as it is a unique identifier to represent the current software version running on the device. The hash field stores the digest of the firmware, which is derived from the hashes and hash trees of relevant partitions, such as *boot*. Additionally, it is included in *vbmeta*.

Remote Attestation As already shown in [34], a software-only based attestation for Android is not sufficient to rely on. Therefore, we focus our research on hardware-backed attestation [40]. Android already provides functions to receive a hardware-backed key and the corresponding attestation certificate [19]. To perform key attestation on an Android device, we specify a custom alias of a key pair in order to retrieve the attestation certification chain. To prevent replay attacks (T6), we leverage a cryptographic nonce to ensure that every verification session is unique. This nonce must be included in the attestation results provided by the auditee. As we are focusing on the hardware-backed key attestation feature, the root certificate of that chain is signed using the *attestation root key* stored in the hardware-backed keystore. All relevant information required to verify the authenticity and integrity of the OS is stored in a specific extension called *RootOfTrust* according to an ASN.1 schema. The *RootOfTrust* property contains information about the verified boot key, the lock state of the device, the verified boot state, and most importantly the verified boot hash. The verified boot hash integrates the digest of the vbmeta struct used by AVB to verify the custom partitions. This approach is recommended by Android to perform remote attestation on Android devices in locked state. Since our research focuses on devices in unlocked state, it is important whether security guarantees such as the integrity and authenticity of the attested vbmeta digest are valid or might have been compromised due to skipped security checks. As existing public documentation is not sufficiently detailed on this (so far, not officially supported edge case), we experimentally verify if there are any differences between locked and unlocked devices with regards to remote attestation. We use the auditee app in debug mode to gather relevant data for our experiment. We start the experiment with querying relevant information for the attestation request on a Google Pixel 6 with a locked bootloader. Afterwards, we query the same information, but with an unlocked bootloader. The experiment shows (see Appendix B) that the vbmeta digest stays the same when switching the lock state of a device. Thus, it becomes evident that nothing on the respective partitions has been compromised. An interesting observation is that the *verified boot key* is deleted and set to *0* when unlocking the bootloader. One reason for this is to prevent evil maid attacks, as addressed in section 7. An essential result partly forming the basis of our trust assumptions is the existence of the root public key[4]. That associated

[4] The related root certificate can be found at `https://developer.android.com/privacy-and-security/security-key-attestation#root_certificate`.

private key of the root public key is stored in the HSM and thus we (have to) assume that an adversary is not able to use this key for signing any attestation details. It does also not change when switching from unlocked to locked state or vice versa. Therefore, we can rely on this trust anchor to verify the authenticity of the provided attestation data.

Audit. As we cannot trust the device generating this certificate chain, we have to send the signed certificate to an independent party, namely the auditor. We use QR codes as a communication channel to exchange information between the auditee and the auditor. The auditor verifies basic properties of the certificate, like the revocation list, and parses the ASN.1 part of the certificate extension to extract the attestation information. Afterwards, the auditor generates the expected log entry using the version and the verified boot hash of the *RootOfTrust* property. The final step is to request the inclusion proof of the generated log entry and verify if the log entry is part of the log itself. Additionally, the auditor can store the current root hash of the log tree to perform consistency proofs in future to verify the append-only property of the transparency log.

Securing User Data Although a user can verify the authenticity of the booted firmware image by using our proposed system and thus can detect a tampered firmware image, it still would not prevent unauthorized access to the encrypted user data. A common mitigation technique to protect the user data from unauthorized access, is to delete the decryption key every time when the bootloader gets unlocked. This key gets generated by a key derivation function that also considers the user knowledge factor, like the PIN code. To prevent brute force attacks, current Android systems leverage a security function provided by the TEE. However, our design proposal focuses on devices that have an unlocked bootloader already so we have to assume that the key has not been deleted if an evil maid attack allows an adversary to replace the firmware image because the unlock operation has already been performed. Although the user can verify whether the booted firmware image is authentic, the user has to complete the booting process first and additionally would have to provide the lock screen knowledge factor to start a verification app. Thus, an adversary would also have access to the knowledge factor provided by the user (T4). We mitigate this threat by hooking the *boot-complete* signal within the app to start the verification process before the user has to provide the knowledge factor to unlock the user partition. Our prototype implementation demonstrates that the audit can be performed on top of lock screen as well as before first unlock of the device. This enables the user to verify the firmware image before unlocking the user data partition with the knowledge factor (as shown in Appendix C).

3.6 Security Guarantees on Locked Devices

Our research primarily focuses on mobile devices with an unlocked bootloader. However, we believe that *Firmware Transparency* further enhance verifiability on rare devices already supporting setting a custom RoT and allow the device

to be re-locked (YELLOW state). On such devices, AVB performs verification during the boot process, ensuring that tampering attempts are detected. If an adversary has ample time to replace the firmware image (e.g., if a journalist is forced to provide the device during an interview), they could also replace the custom RoT used for verification. In such cases, AVB would fail to detect manipulation, as the custom RoT would align with the modified firmware. This attack works on devices (e.g., Cubot A1) [26] where the user data partition is not wiped when changing the lock state of the bootloader, regardless of whether the wipe is not performed due to exploiting a vulnerability or the manufacturer specific implementation. However, *Firmware Transparency* does not solely rely on the trustworthiness of that particular key, because the verification allows the user to verify the firmware regardless of the flashed key.

4 Validation and Vulnerability Mitigation

We validated our protocol on various Pixel devices and on a FairPhone 3 with LineageOS 22 (Android 15) and Magisk v28.1. Magisk is used to emulate a malicious modification on the *boot.img* of the device firmware by generating a modified *boot.img* that we flashed on an unlocked device. The first test involves auditing an unlocked device to compare the original LineageOS image with a custom version. We booted the device using the unmodified version, conducted an audit, and received a positive result. Subsequently, we flashed the modified version onto the device, rebooted it, conducted the verification, and received a negative result, thereby making the tampering attempt transparent to the user.

Our second test involved a device in locked state where we exploited a vulnerability [26] to flash a custom firmware image and still get a *GREEN* boot state. We performed the same steps as we've done before with the unlocked device. Finally, *Firmware Transparency* successfully identified the malicious modification, even though AVB, the primary security control, failed to detect any modifications. It took about 30 seconds to replace the modified image and re-lock the bootloader, demonstrating a practical threat.

5 Formal Verification

We formally model and verify the underlying protocol of *Firmware Transparency* using TAMARIN [6], a security protocol verification tool. We use the Dolev-Yao [11] adversary model where the adversary is able to eavesdrop, tamper, or replay messages sent within the network.

5.1 Protocol Flow

This section describes the protocol, as illustrated in Fig. 3, including relevant stakeholders and their interactions.

Generate image (1): The very first step of our protocol starts with developing (A) the firmware image and sending it to the distributor $Image_A(img)$.

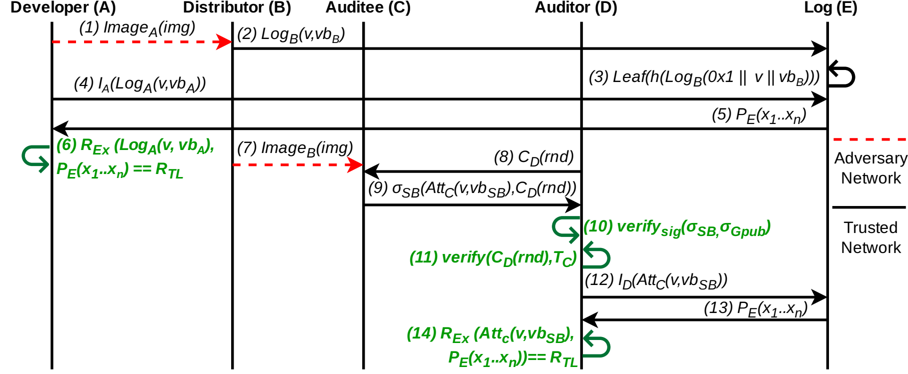


Fig. 3. Protocol of *Firmware Transparency*

Create log entry (2): The distributor (B) then creates the transparency log entry $Log_B(v, vb_B(img)))$ of the firmware image that is going to be shipped to its users. The data structure of the log entries stored in the transparency log system includes information about the firmware version v and the hash value of the image file $vb_B(img)$.
Calculate leaf value (3): Based on the log entry given by the distributor, the personality verifies the input data and calculates the leaf value:

$$\text{Leaf}\,(h\,(\text{Log}_B\,(0x1||v||vb_B)))$$

Verify inclusion proof (4-6): An essential step taken by the developer before the image is distributed is verifying the inclusion proof, such that

$$R_{Ex}(\text{Log}_A(v, vb_A), P_E(x_1..x_n)) == R_{TL},$$

where R_{Ex} is the expected root node hash calculated locally on the developer machine. The calculated root node hash uses x_i which represents the nodes that are part of the audit path. The inclusion proof is valid if the expected root node hash R_{Ex} is equal to the root node hash R_{TL} given by the transparency log. This is important before the distribution as the verification performed by an end-user may fail due to an invalid or missing transparency log entry.
Deploy firmware (7): Once a new entry is added to the transparency log, the distributor can make the image $Image_B(img)$ available to its users. First, the user has to unlock the bootloader to flash a custom firmware. To protect the device from unauthorized modification of the firmware, the bootloader has to be locked again after flashing the custom firmware. However, our proposal ensures sufficient security guarantees even with an unlocked bootloader. Thus, the user neither has to lock the bootloader again nor is able to because there is no way to use a custom RoT, so we can continue by booting up the custom OS.
Initialize audit (8): In this stage the end-user wants to verify if the currently booted firmware image is authentic. The user can start our app (in auditee mode)

and can scan the QR code generated by the app running in auditor mode on a second, independent device. The pairing requires a QR code consisting of a challenge $C_D(rnd)$ used to prevent replay attacks.

Remote attestation (9-10): Next, the auditee requests the attestation and generates a QR code including relevant data and a challenge:

$$\sigma_{SB}(Att_C(v, vb_{SB}), C_D(rnd)),$$

where σ_{SB} is the signature function provided by the HSM, Att_C is the attestation function, vb_{SB} is the provided vbmeta digest by the HSM, and $C_D(rnd)$ is the challenge including the random value provided by the auditor. Once the auditor received the data, the authenticity of the provided information can be verified by checking if the information is signed by a trusted key [5]: $verify_{sig}(\sigma_{SB}, \sigma_{Gpub})$

Verify challenge and TOFU basis (11): Next, the auditor verifies the TOFU basis T_C to prevent relay attacks (T5) and the challenge $C_D(rnd)$ to prevent replay attacks (T6). In case this is the first audit, the auditor stores the TOFU basis for the specific auditee device C.

Request inclusion proof (12): To verify whether the corresponding log entry has been properly logged, the auditor requests the inclusion proof of the attestation details Att_C: $I_D(Att_C(v, vb_{SB}))$,

Providing inclusion proof (13): In case the requested log entry can be found in the Merkle tree, the personality responds with the audit path $P_E(x_1..x_n)$ that includes the necessary tree nodes to calculate the expected root hash.

Verify inclusion proof (14): The final step is to calculate the expected root hash with the information available from the inclusion proof combined with the generated expected log entry: $R_{Ex}(Att_C(v, vb_{SB}), P_E(x_1..x_n)) == R_{TL}$ If the locally calculated hash (also referred to as *expected root hash*) matches the *claimed root hash* provided by the log the auditor confirms that the verification has been successful and that the image is part of the transparency log.

5.2 Tamarin

Our security protocol includes several parties, such as the *personality*, the *auditor*, or the *auditee* that somehow need to interact with each other. Every party has a specific state depending on the flow of the protocol represented as *fact* in Tamarin. A *fact* can be represented as $F(t_1..t_n)$, where F is the name of the fact and t_i a value. A fact in Tamarin can either be linear – so it can be consumed only once – or persistent so that it can be consumed arbitrary times. We also utilize some built-in facts of Tamarin, like $Fr(x)$, $In(..)$, and $Out(..)$. $Fr(x)$ can be used to generate a fresh random value and the $In(..)$ and $Out(..)$ facts can be used to communicate via the adversary network.

[5] In case of Google Pixel, the public key used by StrongBox can be found on https://developer.android.com/privacy-and-security/security-key-attestation (last accessed on 23.05.2025)

The individual protocol steps indicating a state transition are modeled by using multiset rewriting rules (MSR) with three parts. The first part on the left-hand side describes the required fact to apply the MSR. The middle part, also called *action fact* is used to label the transition so that we can later validate our security properties. The right-hand side represents the result of the state transition. To verify the actual protocol behavior including the security objectives, Tamarin uses *lemmas* and *action facts*. Tamarin tries to find a possible trace using first-order logic that contradicts the defined security property.¸ In case Tamarin does not find any trace we can assume that the defined security properties are satisfied.

5.3 Assumptions

We assume the transparency log to be tamper resistant so that an adversary is not able to retrospectively insert, delete, or modify any record once it is stored in the log. Additionally, we assume that the authentication and authorization mechanism incorporated in our personality prevents the adversary from any action except reading log entries. Therefore, we do not except communication between the distributor and the personality via the adversary network. Our model also assumes that the attestation information generated from the auditee and sent to the auditor is trustworthy [2]. This assumption is based on the fact that we assume to have an HSM available that performs the attestation and signs the generated data with a verifiable key. This includes the information transfer between the auditee and the auditor as the signature can still be verified by the auditor. We do not consider the verification of the signature itself within our model, because we assume signatures to be secure. To maintain simplicity our model focuses on the protocol behavior itself by abstracting standard defense mechanisms. We do also consider the usage of a transparency log as already proven defense mechanism against threat T7.

5.4 Security Properties

This section outlines the security properties based on our threat model in section 3.3. Our model defines two types of security properties: one verifies whether there is a trace where an adversary could successfully perform the specific attack without applying our mitigation concept, and the other ensures that there is no trace of success when our verification system is applied. Each security property is associated with an adversary model A (see section 3.3) and a threat T (see section 3.3). We model the security properties as formulas in a first-order logic using *lemmas* within Tamarin. To express the execution of an *action fact* in Tamarin, we use the notation $F(x_1..x_n)@i$ where F denotes the name, x_i the variable, and $@i$ the time variable for the execution.
Evil maid attack and on-path compromise (A1,A2,T1,T2,T4). This security properties are used to verify whether an adversary with physical access to the device or with access to the distribution network is able to compromise the firmware. The first proof we perform using Tamarin is to verify whether there

is a trace where an adversary can successfully manipulate a firmware image f in case the auditor does not verify the attestation at and the inclusion proof ip. Specifically, this lemma proves that $\exists\ f, at, ip : f = at \wedge \neg(h(at) = ip)$. Thus, we can ensure that an attack would be possible with regards to our formal model in case *Firmware Transparency* is not used. Conversely, we prove that TAMARIN does not find any trace where an adversary can compromise the firmware image without detection. Thus, our second lemma verifies $\forall\ f, at, ip : h(f) = ip \wedge f = at$.
Insider compromise (A3,T3). This property verifies if an insider adversary with access to the signing key can compromise a firmware image f without detection. The first lemma determines whether this attack is undetected if the developer does not verify the inclusion proof ip and the attestation at. Specifically, it proves that $\exists\ f, ip : \neg(h(f) = ip)$. Conversely, we prove that TAMARIN does not find any trace such that $\forall\ at, ip : ip = h(at)$ if our protocol is used.
Relay and replay attack (A1,T5,T6). These properties address both relay- and replay attacks, where an adversary either relays a valid attestation result (at_r based on at) to the auditor or replays an existing, but still valid attestation. The first relay- and replay lemmas determines whether these attacks are possible without auditor verification. Specifically, the first lemma for relay attacks verifies $\exists\ at, at_r, t, t_s : at = at_r \wedge \neg(t = t_s)$ and for replay attacks $\exists\ f, at, at_a, c, c_t : \neg(f = at) \wedge at = at_a \wedge \neg(c = c_t)$. Conversely, we prove that TAMARIN does not find any trace for such an attack if our protocol is used. The lemma for relay attacks verifies $\forall\ at, at_r, t, t_s : at = at_r \wedge t = t_s$ and for replay attacks $\forall\ cs, ct : cs = ct$.

6 Related work

GrapheneOS. GrapheneOS [21] is an alternative mobile OS focused on security and privacy. From a security perspective it contains various layers of defense such as disabling certain functionalities, hardened SELinux policies for improved sandboxing, and enhancing verified boot. According to their FAQ section [22], GrapheneOS only supports Google Pixel devices due to security and update aspects. The underlying implementation enforces verified boot during startup by using a custom RoT for the verification process. This is one reason why they only support Google Pixel devices because Google allows flashing a custom RoT on Pixel devices. One of the listed requirement which is also relevant for our paper is having access to full hardware security functionalities. Additionally, GrapheneOS provides a hardware-based attestation app for Android devices [20] used to verify the authenticity and integrity of the OS.

However, GrapheneOS does not provide a secure solution for devices where it is not possible to use a custom RoT. The question with regards to GrapheneOS is not how it differs from our proposal, but rather how our proposal can complement GrapheneOS, enabling secure booting of devices without flashing a custom RoT. The auditor app provided by GrapheneOS can be used to ensure that the mobile device is running an authentic version of the OS. However, this app is only supported on devices with a locked bootloader – *YELLOW* state. Hence, it is not suitable for the use case addressed by our paper. Furthermore, the auditor

of GrapheneOS necessitates that users provide their knowledge factor (e.g., after rebooting the device) before they can perform any form of verification.

Pixel Binary Transparency. Google introduced Pixel Binary Transparency [14] to verify the authenticity and integrity of Pixel factory images. One verification method involves downloading a stock firmware image from the official website and requesting an inclusion proof. Although this method provides a full verification approach, it is only applicable on a downloaded image before flashing it on the device. Additionally, the currently deployed image can be verified by requesting relevant metadata via Android Debug Bridge (adb). However, using adb does not provide high security confidence as the deployed image could already be malicious and thus could tamper the requested values (e.g., the vbmeta digest). At the time of writing this paper, the Android Binary Transparency website [14] only illustrates the adb approach, but with an additional note that they will also provide an example using Key Attestation for better security confidence. Therefore, we conclude that Pixel Binary Transparency may not be suitable for security-sensitive individuals (e.g., journalists) who wish to verify the device's current state before submitting sensitive information on that device.

Both, our proposed system and Android Binary Transparency are based on Merkle trees which are used to store information about firmware images. The similarities of the data structure enable us to use the same underlying Google Trillian implementation for managing the Merkle tree. However, there are significant differences in scope, integration, and operability. Our proposal is not limited to only store information about Pixel firmware images as we focus on mobile devices which do not allow using a custom RoT for locking the bootloader. The use case of the transparency log within our system affects the integration of the log significantly. While the Android Binary Transparency log must be queried by the user with metadata derived from the device, we make use of a separate auditor app which gets relevant information through remote attestation. This approach ensures a full end-to-end verification of an already flashed firmware which is at the time of writing this paper not available for Android Binary Transparency.

Sigstore. The Sigstore ecosystem [32] is an open source project based on available open source tools aiming to address security considerations in software supply chains. The primary focus of Sigstore is trust in signatures on any kind of artifacts. First, the Sigstore client requests a certificate from the Sigstore CA using OpenID Connect to authenticate the developer. During this request, the certificate, together with the artifact hash and the signature, is added to the transparency log. The provided certificate can then be used to sign the artifact that is published to its users (e.g., via web download). Finally, the end user can verify the signature and if it has been properly logged.

The difference with regards to the Sigstore project is that it uses a transparency log to store metadata about an artifact. However, this project focuses on signatures while our proposed system bases the verification on metadata of the firmware image itself. Therefore, Sigstore does not address our threat model as we do not focus on the downloaded firmware image itself, but on the flashed partitions. Thus, Sigstore cannot be used to mitigate lost security guarantees due

to an unlocked bootloader, but would be useful for verifying the downloaded firmware images before flashing it on the device.

7 Deployment Consideration

Rethinking trust on first use. Our proposal still relies on the TOFU model to prevent threats like T5. Thus, an adversary who compromise the device before the first audit could manipulate the device binding parameter to carry out relay attacks on future audits. To strengthen the security guarantees of our proposed solution, we are actively pursuing a follow-up research project focused on achieving even stronger protections without relying on the TOFU model.
Incorporate verification in early boot stage. To strengthen our mitigation against evil maid attacks we suggest incorporating our proposed system in the early bootloader stage. Currently, our solution is designed for sensitive users (e.g., journalists) who want to use a custom firmware image and want to have a possibility to verify the authenticity of the firmware image before providing their knowledge factor and unlocking the user data partition. However, a compromised firmware image could just deactivate the verification or try to trick the user to believe a wrong verification result. Although a sensitive user might detect such attempts and wants to verify the authenticity of the firmware, the overall security could be strengthened by OEMs incorporating our verification control in an early bootloader stage. We are aware that the usability of *Firmware Transparency* in its current state is not practical for the majority of Android users, but strongly believe that it is certainly a mitigation, useful to a selected group of users based on their threat model (e.g., journalists). This matches with the fact that most Android users do not use an alternative firmware. Our proposal needs to be seen in the context of the subgroup of users who are ready to install an alternative firmware.
Convince other vendors. First and foremost, we should convince other vendors to support custom firmware images by allowing users to make use of AVB with a custom RoT. However, we believe that our proposal still provides an additional layer of security especially to prevent attacks on the image distributor.
Relying on RoT. Although our current approach can be used to verify the authenticity of a firmware flashed on a device, it cannot be used on devices without a trusted hardware security module as our design requires a RoT. To our best knowledge, there is currently no solution that sufficiently mitigates the lost security guarantees due to an unlocked bootloader on devices without that root of trust. This is an orthogonal research question that remains open and requires further exploration in the area of replacing hardware RoT in general.
Performance and Scalability Aspects We evaluated the scalability and performance of the transparency log by generating 1,306 log entries, reflecting an unofficial estimate of LineageOS builds obtained from a publicly available crawler dataset [4]. Creating all 1,306 entries took 24,090 ms ($\approx$18.45 ms per entry). An inclusion proof can be generated and verified efficiently in logarithmic time, $O(\log n)$ [31]. Requesting the inclusion proof for the first entry—covering the

maximum audit path for that Merkle tree—took 18 ms, and the proof size was 864 bytes. These results indicate that the transparency log scales well for firmware images and provides sufficient capacity before scalability becomes a concern. If needed, further scalability could be achieved by adopting new deployment strategies [18,27] demonstrated by various providers hosting certificate transparency, where logs handle orders of magnitude more entries. The experiments were conducted in a Docker container on a machine with an AMD Ryzen 5 7640U and 16 GiB of RAM.

We did not observe any significant performance impact on the auditor app on either the auditee or auditor side. On the auditee side, collecting attestation data and generating the QR code took 687 ms on average over five trials. On the auditor side, the end-to-end verification—from receiving the attestation data to validating the corresponding inclusion proof—took on average 302.8 ms over five trials. All experiments were performed on a Google Pixel 7a over a local network within the same broadcast domain as the transparency log, which contained 1,306 entries.

8 Conclusion

We presented a new security mechanism, built on transparency logs, to compensate lost security guarantees of devices with an unlocked bootloader and to enhance verifiability on locked devices that maintain a functional chain of trust. One of the primary goals of our approach is to provide new opportunities for applying updates to mobile devices that are no longer maintained by the manufacturer, while preserving important security primitives. We also acknowledge that most mobile device manufacturers prioritize their own software stack over freedom-of-choice, limiting support for using a custom RoT and thereby restricting usage of important security features. Thus, we aim to support long-term updates due to sustainability aspects and freedom-of-choice without sacrificing security. Additionally, we argue that *Firmware Transparency* further enhances verifiability on devices that already allow users to set a custom RoT. Our concept successfully mitigates realistic threats that we have elaborated within our threat model. To demonstrate the practical feasibility of our concept, we designed and implemented a prototype. The verification of our prototype shows that malicious modifications can be successfully detected, even if a vulnerability enables evasion of current security mechanisms. Finally, we have proven that relevant security properties are satisfied by formally verifying the underlying protocol design based on the considered threat model.

Acknowledgments. We would like to thank Sofia Giampietro, Xenia Hofmeier, and Felix Linker from ETH Zürich for supporting us getting familiar with Tamarin and Adrian Rebola-Pardo from Johannes Kepler University Linz for valuable discussions about formal verification.

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World. We gratefully acknowledge financial support by the Austrian Federal Ministry of Economy, Energy

and Tourism, 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, and Österreichische Staatsdruckerei GmbH.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article. Two of the authors were/are employees of Google and familiar with internal design documents, but have no competing interests regarding the core matter of this paper.

References

1. Acar, A., Tuncay, G.S., Luques, E., Oz, H., Aris, A., Uluagac, S.: 50 Shades of Support: A Device-Centric Analysis of Android Security Updates. In: Proceedings of the USENIX Network and Distributed System Security (NDSS) Symposium (2024)
2. Aldoseri, A., Chothia, T., Moreira, J., Oswald, D.: 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. p. 218–231. ASIA CCS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3579856.3582812, https://doi.org/10.1145/3579856.3582812
3. Amnesty International: A digital prison (2024), https://www.amnesty.org/en/wp-content/uploads/2024/12/EUR7088132024ENGLISH.pdf. Last accessed April 2025
4. Amos Batto: LineageOS Stats (2025), https://github.com/amosbatto/lineageos_stats/. Last accessed November 2025
5. 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
6. Basin, David and Cremers, Cas and Dreier, Jannik and Meier, Simon and Sasse, Ralf and Schmidt, Benedikt: Tamarin Prover (2025), https://tamarin-prover.com/. Last accessed September 2025
7. Biffle, C.L.: The typestate pattern in Rust (2025), http://cliffle.com/blog/rust-typestate/. Last accessed May 2025
8. of British Columbia, G.: BC Wallet (2025), https://www2.gov.bc.ca/gov/content/governments/government-id/bc-wallet. Last accessed September 2025
9. Bundesministerium für Finanzen: Digitales Amt (2025), https://www.oesterreich.gv.at/ueber-oesterreichgvat/faq/app_digitales_amt.html. Last accessed April 2025
10. of California, S.: California DMV (2025), https://www.dmv.ca.gov/portal/ca-dmv-wallet/. Last accessed September 2025
11. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory **29**(2), 198–208 (1983). https://doi.org/10.1109/TIT.1983.1056650
12. Eijdenberg, A., Laurie, B., Cutter, A.: Verifiable data structures (2015), https://github.com/google/trillian/blob/30160804ab5203cde4412fe26f55a4149112bd92/docs/papers/VerifiableDataStructures.pdf. Last accessed April 2025

13. Fairphone: Manage the Bootloader (2025), https://support.fairphone.com/hc/en-us/articles/10492476238865-Manage-the-Bootloader. Last accessed September 2025
14. Google: Android Binary Transparency (2025), https://developers.google.com/android/binary_transparency/overview. Last accessed April 2025
15. Google: Boot flow (2025), https://source.android.com/docs/security/features/verifiedboot/boot-flow. Last accessed May 2025
16. Google: Certificate Transparency (2025), https://certificate.transparency.dev/. Last accessed September 2025
17. Google: Key and ID attestation (2025), https://source.android.com/docs/security/features/keystore/attestation. Last accessed November 2025
18. Google: Tile-Based Transparency Logs (2025), https://transparency.dev/articles/tile-based-logs/. Last accessed November 2025
19. Google: Verify hardware-backed key pairs with Key Attestation (2025), https://developer.android.com/privacy-and-security/security-key-attestation. Last accessed September 2025
20. GrapheneOS: Device integrity monitoring (2025), https://attestation.app/about. Last accessed September 2025
21. GrapheneOS: GrapheneOS (2025), https://grapheneos.org/. Last accessed September 2025
22. GrapheneOS: GrapheneOS - Frequently Asked Questions (2025), https://grapheneos.org/faq. Last accessed September 2025
23. Hay, R.: fastboot oem vuln: Android Bootloader Vulnerabilities in Vendor Customizations. In: 11th USENIX Workshop on Offensive Technologies (WOOT 17). pp. 383–398. USENIX Association, Vancouver, BC (Aug 2017), https://www.usenix.org/system/files/conference/woot17/woot17-paper-hay.pdf
24. Hugenroth, D., Lins, M., Mayrhofer, R., Beresford, A.R.: Attestable Builds: Compiling Verifiable Binaries on Untrusted Systems using Trusted Execution Environments. In: 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25) (2025). https://doi.org/10.1145/3719027.3765128
25. 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
26. Leierzopf, E., Kempinger, S., Mayrhofer, R., Roland, M.: AVBTestKeyInTheWild: Bypassing Android Verified Boot Using A Firmware Supply Chain Vulnerability on Locked Devices. In: SPICES 2025 co-located with EWSN '25: Proceedings of the 2025 International Conference on embedded Wireless Systems and Networks (2025)
27. Let's Encrypt: Introducing Sunlight, a CT implementation built for scalability, ease of operation, and reduced cost (2024), https://letsencrypt.org/2024/03/14/introducing-sunlight. Last accessed November 2025
28. Lins, M., Mayrhofer, R., Roland, M., Beresford, A.R.: Mobile App Distribution Transparency (MADT): Design and evaluation of a system to mitigate necessary trust in mobile app distribution systems. In: Secure IT Systems. 28th Nordic Conference, NordSec 2023. LNCS, vol. 14324/2024, pp. 1–19. Springer (Nov 2023). https://doi.org/10.1007/978-3-031-47748-5_11
29. 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, https://doi.org/10.1145/3448609

30. 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
31. Mozilla Corporation: Certificate Transparency (2025), https://developer.mozilla.org/en-US/docs/Web/Security/Certificate_Transparency. Last accessed November 2025
32. Newman, Z., Meyers, J.S., Torres-Arias, S.: Sigstore: Software signing for everybody. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. p. 2353–2367. CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3548606.3560596, https://doi.org/10.1145/3548606.3560596
33. Possemato, A., Aonzo, S., Balzarotti, D., Fratantonio, Y.: Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 87–102 (2021). https://doi.org/10.1109/SP40001.2021.00074
34. Prünster., B., Palfinger., G., Kollmann., C.: Fides: Unleashing the full potential of remote attestation. In: Proceedings of the 16th International Joint Conference on e-Business and Telecommunications - SECRYPT. pp. 314–321. INSTICC, SciTePress (2019). https://doi.org/10.5220/0008121003140321
35. Qualcomm Technologies, Inc.: Secure Boot and Image Authentication (2019), https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/secure-boot-and-image-authentication-version_final.pdf. Last accessed April 2025
36. Redini, N., Machiry, A., Das, D., Fratantonio, Y., Bianchi, A., Gustafson, E., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: BootStomp: On the security of bootloaders in mobile devices. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 781–798. USENIX Association, Vancouver, BC (Aug 2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/redini
37. The LineageOS Project: Qualcomm's Chain of Trust (2018), https://lineageos.org/engineering/Qualcomm-Firmware/. Last accessed September 2025
38. The LineageOS Project: LineageOS Android Distribution (2025), https://lineageos.org/. Last accessed March 2025
39. Tian, D.J., Hernandez, G., Choi, J.I., Frost, V., Raules, C., Traynor, P., Vijayakumar, H., Harrison, L., Rahmati, A., Grace, M., Butler, K.R.B.: ATtention spanned: Comprehensive vulnerability analysis of AT commands within the android ecosystem. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 273–290. USENIX Association, Baltimore, MD (Aug 2018), https://www.usenix.org/conference/usenixsecurity18/presentation/tian
40. Zhou, Z., Xiao, X., Hou, T., Hu, Y., Gu, D.: On the (In)Security of Manufacturer-Provided Remote Attestation Frameworks in Android. In: Tsudik, G., Conti, M., Liang, K., Smaragdakis, G. (eds.) Computer Security – ESORICS 2023. pp. 250–270. Springer Nature Switzerland (2024)

A Availability

We provide our prototype implementation though three repositories, all publicly available: **(1)** The personality contains the code for the application-specific inter-

face between the auditor app, the firmware distributor, and the logging backend. Source Code:

`https://github.com/linsm/firmwaretransparency-personality`

(2) The auditor project contains the code for the end-to-end verification of the firmware. Source Code:

`https://github.com/linsm/firmwaretransparency-auditor`

(3) The formal verification project contains the Tamarin model and proof of our protocol. Source Code:

`https://github.com/linsm/firmwaretransparency-formalverification`

B Experiment

The following results were obtained from our experiment, where we requested an attestation of a device in both locked and unlocked state to compare the responses.

```
Locked state:
  vbmeta digest:
  AEFEE75C50C79.....6EA6C8005D39EE6C40BC
  verified boot key:
  0F6E75C80183B.....B0819DE1F150BA0FF9D7
  root public key:
  MIICIjANBgkqh.....i8WEg5UmAGMCAwEAAQ==

Unlocked state:
  vbmeta digest:
  AEFEE75C50C79.....6EA6C8005D39EE6C40BC
  verified boot key:
  0000000000000.....00000000000000000000
  root public key:
  MIICIjANBgkqh.....i8WEg5UmAGMCAwEAAQ==
```

C Auditor App

Figure 4 present some impressions of the user interface and design of the *Firmware Transparency Auditor App*.

D Formal Verification

D.1 Security Properties

This section provides a more detailed definition of selected security properties as representative examples.

The following security properties are used to verify whether an adversary with physical access to the device or with access to the distribution network

Table 1. Notations used in Tamarin lemmas

Notation	Description
A, B, C, D, E	Participants: A(Developer), B(Distributor), C(Auditee), D(Auditor), E(Personality)
f, at, ip	Data: f(Image), at(Attestation), ip(Inclusion Proof)
$\#$	Time variable
$h()$	Hash function
$Audit_D()$	Auditor function to verify combination of attestation data and inclusion proof
$InitVerify_A()$	Developer check to verify if image is properly logged
$Publish_A()$	Developer publishes new firmware image
$Distribute_B()$	Distributor distributes firmware image or creates inclusion proof
$GetImage_C()$	Device downloads image
$SendIP_E()$	Personality sends inclusion proof
$GetAtt_D()$	Auditor receives attestation data from device
$GetIP_D()$	Auditor receives inclusion proof from personality

is able to compromise the firmware. The first equation verifies if the adversary would be successful without using the auditor to validate the firmware image. The notation of the following lemmas is described in Table 1.

$$\exists\, C,\, D,\, E,\, at,\, ip,\, \#i,\, \#k.\ ((GetAtt_D(D,\, C,\, at)\ @\, \#i).$$
$$\wedge(GetIP_D(D,\, E,\, ip)\ @\, \#j)) \wedge (\neg(\mathrm{h}(at) = ip))$$

The second lemma verifies all possible traces if a tampered firmware image can be detected.

$$\forall\, A,\, B,\, C,\, D,\, E,\, f,\, at,\, ip,\, \#v,\, \#w,\, \#x,\, \#y,\, \#z.$$
$$(((((InitVerify(A,\, f,\, ip)\ @\, \#v) \wedge (GetImage_C(C,\, B,\, f)\ @\, \#w))\wedge$$
$$(GetIP_D(D,\, E,\, ip)\ @\, \#x)) \wedge (GetAtt_D(D,\, C,\, at)\ @\, \#y))\wedge$$
$$(Audit_D(D,\, ip,\, at)\ @\, \#z))$$
$$\Rightarrow ((h(f) = ip)\ \wedge\ (f = at))$$

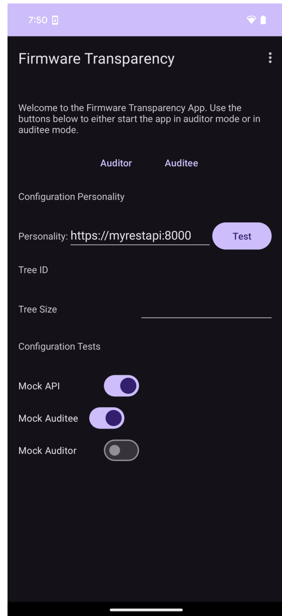
$$\exists\, A,\, B,\, C,\, E,\, f,\, ip,\, \#i,\, \#j,\, \#k.\ (((\,$$
$$Publish_A(A,\, B,\, f)\ @\, \#i) \wedge (Distribute_B(B,\, E,\, f)\ @\, \#j))\wedge$$
$$(SendIP_E(E,\, C,\, ip)\ @\, \#k)) \wedge (\neg(\mathrm{h}(f) = ip))$$

The lemma below proofs that an insider attack gets detected when using *Firmware Transparency*.

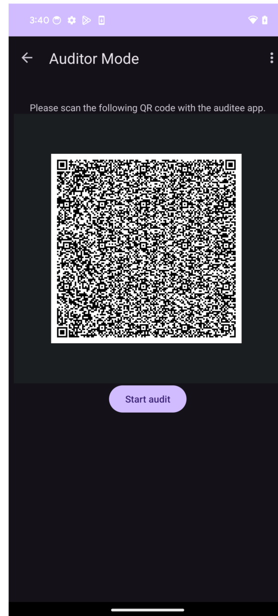
$$\forall\, A,\, D,\, f,\, at,\, ip,\, \#x,\, \#y.\ ((Audit_D(D,\, ip,\, at)\ @\, \#x)$$
$$\wedge(InitVerify_A(A,\, f,\, ip)\ @\, \#y)) \Rightarrow (ip = h(at))$$

D.2 Attack Path Examples

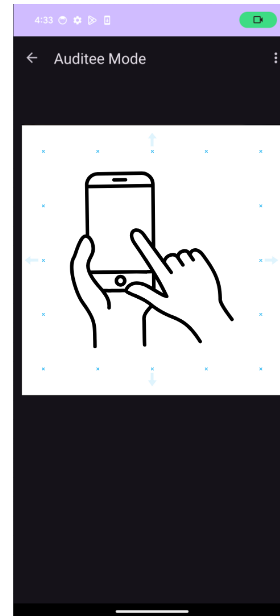
Figure 5 and 6 are examples of the identified traces by Tamarin used to verify physical, on-path, and insider attacks on devices with an unlocked bootloader.



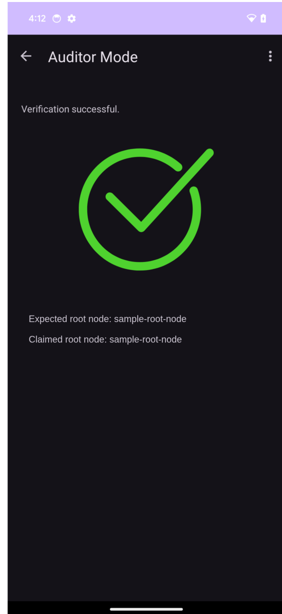
(a) Starting our auditor app.



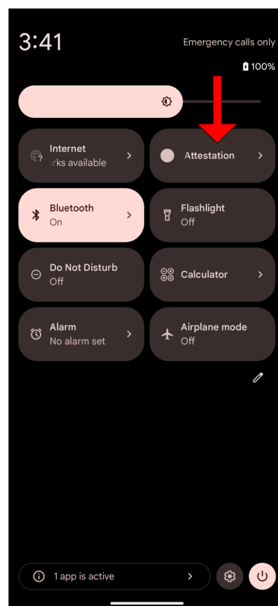
(b) App in auditor mode providing the challenge to the auditee.



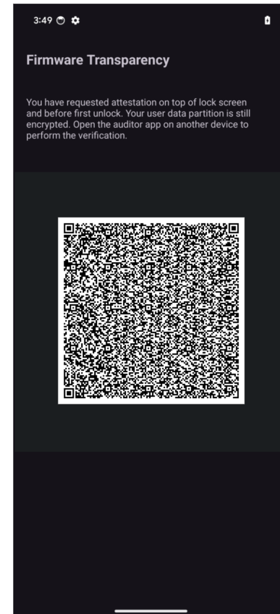
(c) App in auditee mode scanning the challenge provided by the auditor.



(d) Verification confirms that the expected root node hash matches the claimed root node hash.



(e) Tile used to start verification on top of lockscreen.



(f) Attestation data provided to the auditor on top of lock screen.

Fig. 4. Screenshots showcasing some examples of our Auditor app

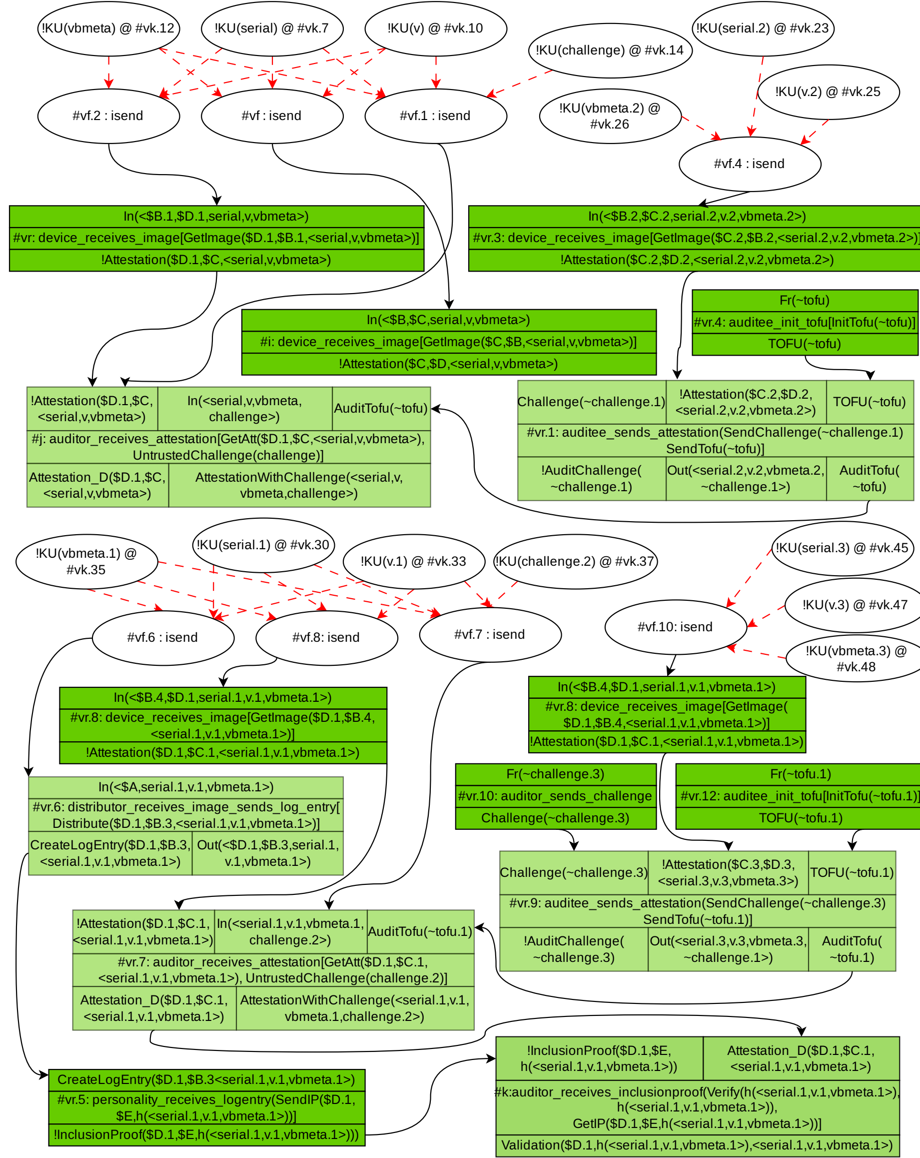


Fig. 5. This figure shows attack traces found by TAMARIN for physical and on-path attacks. We use them to verify whether the specific attack would be successful if *Firmware Transparency* is not used. Once *Firmware Transparency* is used, TAMARIN no longer detects any traces.

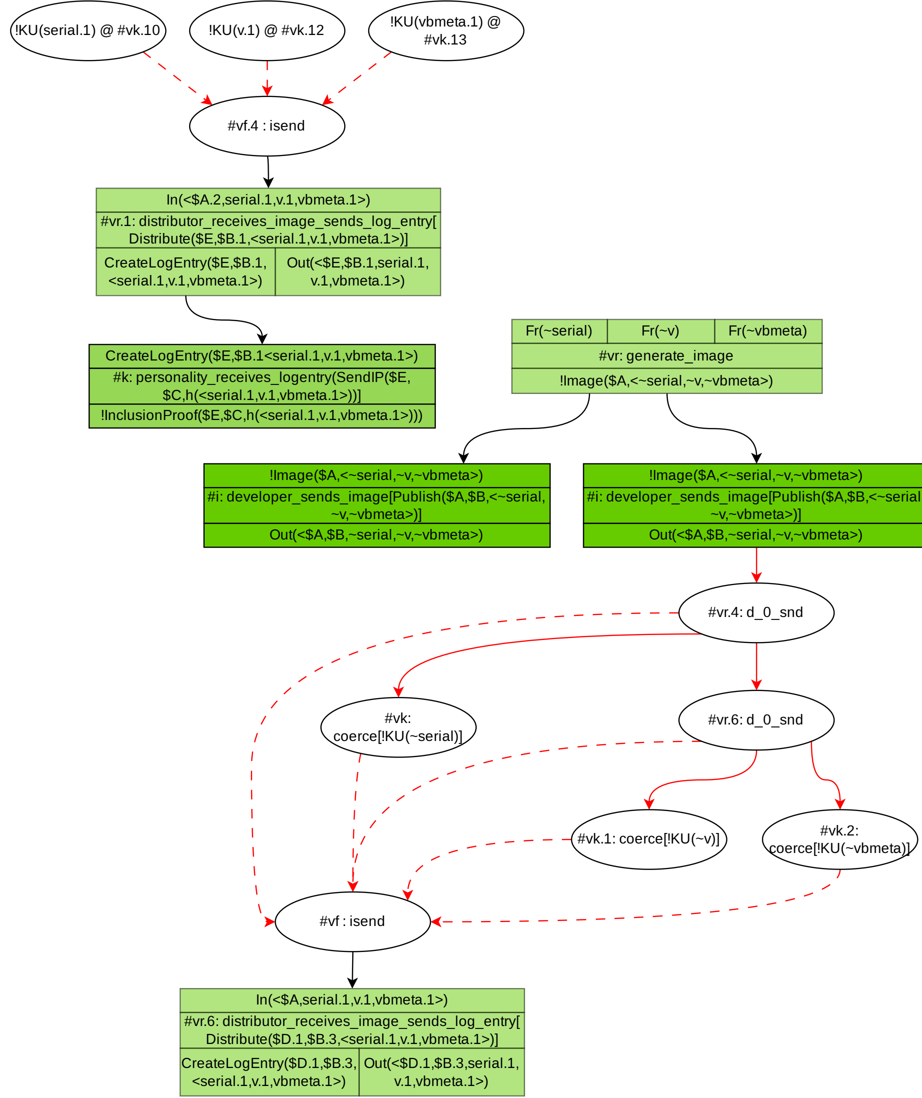


Fig. 6. This figure shows attack traces identified by TAMARIN for insider attacks. Once *Firmware Transparency* is used, TAMARIN no longer detects any traces.