AVBTestKeyInTheWild: Bypassing Android Verified Boot Using A Firmware Supply Chain Vulnerability on Locked Devices

Ernst Leierzopf
Johannes Kepler University Linz
Linz, Austria
ernst.leierzopf@ins.jku.at

Stefan Kempinger
Johannes Kepler University Linz
Linz, Austria
stefan.kempinger@ins.jku.at

René Mayrhofer
Johannes Kepler University Linz
Linz, Austria
rene.mayrhofer@ins.jku.at

Michael Roland
Johannes Kepler University Linz
Linz, Austria
michael.roland@ins.jku.at

Abstract

The fragmented nature of the Android market makes it difficult to enforce consistent security practices across all devices and vendors. Low-level and hardware-bound components such as the bootloader, Fastboot, kernel, device drivers, and the recovery partition are vendor-specific. Each vendor must implement the standard Hardware Abstraction Layer interface for Android to communicate with the hardware. These deviations are typically located in the `vendor` partition of the device.

We identify a critical vulnerability, which we name *AVBTestKeyInTheWild*, in the Android firmware supply chain that enables attackers to flash modified firmware images onto locked devices without wiping the `userdata` partition. By exploiting weak signing practices, such as the use of Android Open Source Project test keys in production firmware, attackers can bypass bootloader integrity checks, retain user data, and compromise device security without user interaction. The vulnerability affects multiple manufacturers and devices, posing significant risks to user privacy, device integrity, and the Android ecosystem as a whole. We provide a detailed analysis of the attack path, demonstrate exploitation on devices from different System-on-Chip vendors, and highlight the limitations of current integrity verification mechanisms, such as Android Verified Boot and key attestation. Because the vulnerability impacts multiple vendors, we decided to work with Google, in addition to the impacted OEMs, on the coordinated disclosure process to inform all involved parties properly. The vulnerability was reported privately with an ethical vendor response window of 90 days before public disclosure. While, to the best of our knowledge, impacted devices cannot be fixed, we suggest detection mechanisms and mitigation strategies. These include stricter firmware signing protocols, enhanced attestation processes, and improved testing frameworks, to prevent the production of vulnerable devices in the future.

CCS Concepts

• **Security and privacy** → *Malware and its mitigation*; *Software security engineering*; **Mobile platform security**; **Trusted computing**.

Keywords

Android, Supply Chain Security, Firmware Integrity, Vulnerability, Exploit

1 Introduction

The Android ecosystem relies heavily on the integrity of its firmware and a secure supply chain. Android devices are organized into multiple partitions, each serving a specific purpose. The `system` partition contains the core operating system files, while the `vendor` partition houses hardware-specific drivers and configurations. The `boot` partition includes the kernel and ramdisk, which are essential for initializing the operating system. User-specific data, such as app settings and personal files, are stored in the `userdata` partition. This partition is encrypted with a `synthetic password` to protect sensitive information, and it is wiped during a factory reset to restore the device to its default state [4]. The `synthetic password` is encrypted with a key derived from the Lock Screen Knowledge Factor (LSKF) and stored on disk, whereas the LSKF is stored in the hardware-backed keystore. Other partitions, often vendor-specific, may also exist to support additional functionality.

The `vbmeta` partition, a cornerstone of Android Verified Boot (AVB), contains cryptographic signatures and hashes that are used to validate the integrity of other partitions. As an orthogonal side note, not all partitions are included in `vbmeta`, and major vendors, including Google, have partitions excluded from the AVB process. These excluded partitions must be verified through different means —usually on a different processor such as the radio/modem. If an attacker possesses signing keys for these excluded partitions, they can modify and re-sign them without detection through the (AVB-based) device integrity attestation. However, partitions holding code not executed on the main CPU (a.k.a. Application Processor, AP) are outside the scope of this paper.

Android Open Source Project (AOSP) includes, among others, private test key(s)[1] to sign the `vbmeta` partition disk image. This allows for an easier development process, without the need to generate new public-key pairs. While this allows for easier, automated testing of the code base, it poses serious issues when used in production-grade firmware images. With the possession of the signing

[1] https://android.googlesource.com/platform/external/avb/+/refs/heads/main/test/data/

key, attackers are able to rebuild the full `vbmeta` hashtree and modify all included partitions, while replacing the signing keys of the modified partitions.

Existing mitigations against such insider attack threats include the Gatekeeper and Weaver components which implement functionality to protect existing data from exfiltration using the LSKF and transparency logs that contain public tamper-evident records of release firmware versions to detect targeted attacks [13].

In this work we identify *AVBTestKeyInTheWild*, a critical vulnerability in the supply chain of multiple Android device manufacturers that allows attackers to bypass bootloader integrity checks and modify firmware on locked devices. We verified the vulnerability by exploiting devices actively sold in retail stores within the European Union at the time of writing. Devices from different System-on-Chip (SoC) vendors including Fairphone 3 (Qualcomm), Tecno Spark 10 Pro (MediaTek), and Cubot A1 (Unisoc) were successfully flashed with modified firmware and locked bootloader, without wiping the `userdata` partition and still passing all security checks. Google Play Integrity, as well as tests from the Compatibility Test Suite (CTS), currently do not detect tampered-with devices, which means that apps successfully verify the integrity of susceptible devices with modified firmware. The vulnerability exploits the usage of public AOSP signing test keys[1] found in production-grade firmware images from five different vendors so far, including but not exclusive to Fairphone, Cubot, Tecno, Itel, and Infinix.

The implications of this vulnerability are far-reaching, impacting device security, user privacy, and the trustworthiness of the affected devices. We show that, under certain conditions, if signing keys to partitions not present in the `vbmeta` structure—or even worse, the VBMeta-signing key itself—get leaked, Android device integrity attestation fails on affected devices.

Initially the widespread usage of AOSP signing test keys in firmware was discovered using android_universal[2], an open source tool that was last updated in July 2021, about four years ago. This indicates that the vulnerability existed for a long time and was known by some individuals.

To allow for better analysis of the firmware images, we adapted `avbtool` and upstreamed our changes[3]. The main contributions are that instead of stopping verification on failure, all partitions are verified, while the verification still fails. This helps users to see which partitions can be verified successfully, even when one partition fails for any reason such as not being included in the firmware image. It is common that not all partitions are distributed through a firmware update. The new `--allow_missing_partitions` argument lets the verification succeed, even when some partitions are missing. A new method `print_signature_key` to print the public key of `vbmeta` and to check against known test keys in a subfolder was also added.

A mass scan of available first-party Android firmware images revealed that AOSP test keys, which are intended to sign partition disk images during development, are found in production environments, making this vulnerability exploitable in a viable low-cost manner in real-world scenarios. Attackers can leverage this weakness for actions such as data extraction, rooting, and infecting devices of targeted adversaries.

As our main contributions, we

- provide a detailed analysis of the vulnerability, its attack path, vulnerable devices, and the impact on the Android ecosystem;
- demonstrate exploitation of three Android devices from different SoC vendors to highlight this impact;
- outline indicators of firmware modifications and challenges in verifying device integrity;
- recommend mitigations for detecting and preventing test key usage in production firmware; and
- propose improvements to Android key attestation for enhanced security.

2 Background

The Android ecosystem is a collaborative framework involving multiple stakeholders, including AOSP, Google, device vendors, and SoC manufacturers. AOSP provides the foundational codebase, while Google defines compatibility standards through the Android Compatibility Definition Document (CDD)[4]. Device vendors customize and distribute Android for their devices, often relying on SoC manufacturers for hardware components and low-level firmware [13].

Compatibility and Integrity Tools. To enforce compatibility and security, Android employs several testing and validation tools:

Compatibility Test Suite (CTS)[5] is a suite of automated tests designed to verify a device's compatibility with the CDD. Its main objective is to verify functional and security requirements of devices, enabling a consistent user experience across devices. Part of the CTS is the Security Test Suite (STS), which includes tests for known security issues.

Vendor Test Suite (VTS)[6] focuses on testing hardware-specific components and their integration with the Android platform. These tests ensure that vendor-specific implementations do not compromise system security and compatibility.

Google Play Integrity API[7] provides runtime integrity checks for apps, allowing developers to verify the authenticity of the device and installed apps using hardware-backed key attestation. This security layer helps to identify tampering, unauthorized modifications, and the use of compromised devices, while also assisting Google in detecting statistical inconsistencies and evaluating the security of devices in the field.

Android Verified Boot. When locked, the Android bootloader enforces strict integrity checks, ensuring that only verified firmware can be loaded. Unlocking the bootloader bypasses these checks, allowing custom firmware to be installed. This bootloader unlocking process typically requires user interaction in the form of triggering the `OEM Unlocking` setting in the developer options while the device is running and unlocked with the LSKF. Another option is to use the `fastboot flashing [unlock | lock]` command. All

[2] https://github.com/bkerler/android_universal
[3] https://android-review.googlesource.com/c/platform/external/avb/+/3604872
[4] https://source.android.com/docs/compatibility/cdd
[5] https://source.android.com/docs/compatibility/cts
[6] https://source.android.com/docs/core/tests/vts
[7] https://developer.android.com/google/play/integrity

state transitions result in wiping of the data partitions [2] to prevent unauthorized access to sensitive information and device tampering. It is worth noting that this behavior depends on the device- and vendor-specific bootloader implementation, which might not comply with the standard. Android devices communicate their integrity status through boot states displayed during startup [1]. A green state indicates that all partitions are verified using manufacturer-provided keys, while a yellow state signifies a locked state in which a user-installed root of trust is present. This verification process is handled via the AVB[8] framework, which checks the integrity of partitions against their cryptographic signatures. An orange state represents an unlocked bootloader, and a red state warns of corruption or the absence of a valid operating system. In our case, we always achieved a green boot state, even with our modified partitions, as the test keys used to sign these images were accepted by the device bootloader.

Attesting Device Integrity. Keystore attestation is a critical feature enabling (remote) verification of cryptographic keys and their associated properties, including the overall security state of the device on which they are held. This mechanism ensures that keys are securely generated and stored within hardware-backed environments, such as the Trusted Execution Environment (TEE) or StrongBox. Attestation provides cryptographic proof that a key was generated on a certified device and, assuming hardware security guarantees hold, has not been tampered with or exported.

The attestation process involves generating an attestation certificate, which includes metadata about the key, such as its origin, purpose, and security properties. This certificate is signed by a hardware-protected attestation key, ensuring its authenticity. Neither the generated private key, which can optionally be secured using the LSKF, nor the device-bound private key used for signing the attestation certificate can be exported or used outside the secure hardware environment. Applications and backend services can use this certificate to remotely verify the integrity of the key and device security state. In particular, the attestation certificate includes information about the Verified Boot state, such as whether the device bootloader is locked and running verified firmware.

We explicitly assume all related secure hardware environments—including TEE hardware and software and, if available, StrongBox components—to remain secure and the attestation certificates to accurately represent all information passed on by the earlier stages of AVB, specifically its lock state and the top-level hash of the `vbmeta` partition itself.

3 The AVBTestKeyInTheWild Vulnerability

The vulnerability we found, *AVBTestKeyInTheWild*, takes advantage of smartphone vendors using a master key, often an AOSP test key, to sign the `vbmeta` structure.

3.1 General Attack Path

Step 1: Obtaining Firmware Files. In some cases, the original firmware files can be extracted directly from the target device using tools such as SPD Flash Tool[9] for Unisoc-based devices or MTKClient[10] for MediaTek-based devices. These tools are widely available online and require only a USB connection to the device. Alternatively, firmware images can often be downloaded from external sources.

Step 2: Unpacking, Analyzing, and Modifying Firmware. Once acquired, the firmware can be unpacked using utilities like `AndScanner`[11], which allows for detailed inspection and modification of the firmware components. In our proof of concept, we modified the `boot` partition to install `Magisk`, which enables root access and other customizations. This simple modification is sufficient to demonstrate the vulnerability, as it shows the attacker running code with the highest Android privileges on the device.

Step 3: Re-signing and Verifying Images. Using the AOSP test key(s), we re-signed the modified `boot.img` and `vbmeta.img` files that are subsequently flashed as the contents of the respective on-device partitions. This process includes updating the `vbmeta-footer` within `boot.img` to ensure consistency with the new signatures. After modifying and re-signing the relevant images, the `vbmeta.img` must be regenerated to reflect the changes. The `avbtool` can be used to generate and verify the correctness of the updated `vbmeta.img` by executing, e.g.,

```
avbtool.py verify_image --image vbmeta.img.
```

This step ensures that the cryptographic metadata aligns with all modified partitions, allowing the device to boot without triggering integrity warnings [3].

Step 4: Deploying the Modified Firmware. Depending on the device, the final step involves either unlocking the bootloader or force-overwriting existing partitions. While unlocking and re-locking the bootloader requires force-erasing of the `userdata` partition, most devices offer other means to flash partitions with locked bootloaders. However, such functionality is publicly available for only a minority of vendors and devices.
A more detailed description of the proof-of-concept exploit can be found in our Coordinated Vulnerability Disclosure (CVD)[12].

3.2 Vulnerable Devices

Our analysis of Android firmware images in the wild revealed that many devices are vulnerable to the *AVBTestKeyInTheWild* vulnerability. We use an automated pipeline that receives Android firmware images—previously downloaded with webscrapers from the manufacturers' websites in their vendor-specific formats—extracts data from the included disk images, and analyzes the firmware for potentially security-relevant findings. Through this system, we identified 69 potentially vulnerable devices. Since we only used publicly available first-party firmware images, this list is not exhaustive.

Fairphone 3. The `vbmeta` partition for the Fairphone 3 includes the hashtree for multiple partitions but does not sign them with separate keys. By exploiting this vulnerability, we were able to

[8] https://source.android.com/docs/security/features/verifiedboot

[9] https://spdflashtool.com/

[10] https://github.com/bkerler/mtkclient

[11] https://github.com/ernstleierzopf/AndScanner/tree/main

[12] https://issuetracker.google.com/issues/416187987

Table 1: Vulnerable devices

Cubot J5	Cubot MAX 5	Cubot TAB 50	Cubot KingKong AX
Cubot J7	Cubot X30	Cubot Note 20	Cubot KingKong X70
Cubot J8	Fairphone 3	Cubot Note 30	Cubot KingKong Star
Cubot J9	Fairphone 4	Cubot Pocket 3	Cubot KingKong Star 2
Cubot A1	Redmagic 3S	Cubot Quest Lite	Cubot KingKong Power
Cubot C20	Redmagic 5G	Cubot X30 P	Cubot KingKong Power 3
Cubot J10	Redmagic 5S	Cubot Ace 3	Cubot KingKong 5 Pro
Cubot R15	Redmagic 6R	Cubot R15 Pro	Cubot KingKong ACE 2
Cubot P50	Redmagic 6	Redmagic Mars	Cubot KingKong ACE 3
Cubot P60	Cubot Note 7	Redmagic 6 Pro	Cubot KingKong Mini 3
Cubot P40	Cubot Note 8	Redmagic 6S Pro	Cubot TAB KingKong
Cubot J20	Cubot Note 9	Cubot KingKong 5	Cubot TAB KingKong 2
Cubot X50	Cubot Pocket	Cubot KingKong 6	Cubot KingKong Mini 2 Pro
Cubot P80	Cubot Quest	Cubot KingKong 7	Asus ZenFone Max (M2)
Cubot C30	Cubot TAB 10	Cubot KingKong 8	Tecno Spark 10 Pro
Cubot MAX 2	Cubot TAB 30	Cubot KingKong 9	Infinix Hot 50 Pro 8
Cubot MAX 3	Cubot TAB 40	Cubot KingKong X	

flash a `Magisk`-modified `boot.img` while maintaining a green boot state and passing all security checks, including the `MEETS_STRONG_INTEGRITY` flag. Partitions can be flashed without requiring a wipe of the `userdata` partition due to an additional vulnerability in the handling of the bootloader lock state. In an older version of the Fairphone 3 firmware, the vendor included a special `devinfo` partition, which was used to unlock the bootloader. This unlocking process did not automatically wipe the `userdata` partition, instead relying on a setup script to wipe the `userdata` partition after unlocking. After the device was unlocked, we were able to flash our modified `boot.img` and `vbmeta.img` without wiping the `userdata` partition, but a re-lock of the bootloader was still handled by issuing the appropriate `fastboot flashing lock` command, wiping the `userdata` partition in the process When further testing the handling of the bootloader lock state, we found that in a locked state, this `devinfo` partition changes by only two bits, and by flashing the locked `devinfo` partition, we could re-lock the bootloader without wiping the `userdata` partition at all.

Cubot A1. Using the `Unisoc Upgrade Download` tool, we flashed modified firmware onto the Cubot A1 without unlocking the bootloader or wiping the `userdata` partition.

Tecno Spark 10 Pro. The `MTKClient` tool was used to dump partitions on the Tecno Spark 10 Pro and to flash modified partitions without wiping the `userdata` partition. While a security patch in later firmware versions mitigated the exploit that allowed force-flashing on locked devices, attackers can still install modified firmware on unlocked devices and relock the bootloader.

4 Related Work & Discussion

The security of the Android ecosystem has been extensively studied, particularly in the context of firmware integrity. Prior research has highlighted vulnerabilities in Android bootloaders and firmware [9, 15], which should be detected by verified boot mechanisms in Android and in Linux systems more generally [6, 13]. For instance, BootStomp [15] identified memory corruption and unlock-bypass vulnerabilities in commercial Android devices, while FirmAlice [16] demonstrated authentication bypass vulnerabilities in other types of firmware. These studies underscore the importance of robust bootloader and firmware integrity mechanisms, which are directly relevant to the exploitation path of the *AVBTestKeyInTheWild* vulnerability.

Original Equipment Manufacturer (OEM) practices of system customization have also been shown to introduce security issues, such as the inclusion of vulnerable apps [5, 7, 19, 20] or unsafe modifications to access control policies [10, 14]. In addition, vendors often weaken security by failing to promptly update the software components they ship, sometimes including firmware or libraries that are significantly outdated [8, 11, 12, 17, 18].

By definition, certain vendor-provided elements, including radio chipsets or hardware-backed security modules (in Android these are the TEE or StrongBox) are explicitly out of scope for AVB [13]. The platform is still open to OEM customization there, which, as our results and related work show, lead to weakened or in some parts invalidated security guarantees [18]. Some of those elements rely on trust anchors baked into the chip, and thus further fragmenting the security guarantees of Android on a per-device basis.

Previous vulnerabilities [CVE-2024-20865, CVE-2018-1000205, CVE-2018-9567] have shown the importance of a secure supply chain to verify the integrity of firmware. A very similar vulnerability to *AVBTestKeyInTheWild* was also discovered in the past [CVE-2023-45779], which takes advantage of the use of public AOSP test keys APEX modules and seven OEMs were found to be at risk.

4.1 Impact on the Android Ecosystem

At its core, the *AVBTestKeyInTheWild* vulnerability allows attackers to modify signed partitions on affected devices, fundamentally undermining the guarantees provided by AVB. This breach enables adversaries to bypass critical integrity checks and permits the undetected installation and launch of modified firmware.

Vulnerable devices permit attackers to alter system partitions, sometimes even on devices with locked bootloaders. In typical scenarios, flashing modified partitions requires unlocking the bootloader, which triggers a wipe of the `userdata` partition. By leveraging the update tools mentioned in Section 3.1, attackers can flash modified firmware images on affected devices, potentially even without wiping the `userdata` partition. This creates a significant supply chain risk, as compromised devices can be prepared by flashing modified partitions and be distributed to unsuspecting users, who have no indication that their device has been tampered with. Due to several additional vulnerabilities, we were able to force-flash partitions without triggering a wipe, allowing us to retain user data and settings on all tested devices (Fairphone 3, Cubot A1, and Tecno Spark 10 Pro). On these devices, attackers can access sensitive user data without requiring any immediate interaction from the user, by flashing a modified firmware image and waiting for the entry of the LSKF.

Furthermore, the ability to bypass Android device integrity checks renders existing mechanisms for detecting firmware modifications ineffective. Applications that rely on these checks, such as banking or enterprise security apps, are unable to distinguish between genuine and compromised devices. Attackers can leverage this vulnerability to gain persistent root access, embed malicious code (such as keyloggers or spyware) into system partitions, and maintain control over the device even after factory resets. The exploit also enables large-scale attacks, as a single modified firmware image can be deployed across multiple identical devices, and

the modification process is often similar to other devices from the same vendor.

Depending on the device, exploitation of this vulnerability is usually straightforward and fast. If an attacker has a prepared firmware image, they can flash it onto a vulnerable device in a matter of seconds, switching the device from a secure state to a compromised state without any user interaction.

Impact on Vendors. The use of insecure signing keys in production firmware indicates lapses in supply chain security and key management practices. Vendors may be subject to regulatory scrutiny, potential lawsuits, and loss of consumer trust. Additionally, the need to remediate affected devices in the field can incur significant costs, including firmware updates, customer support, and possible device recalls.

Impact on Device Owners. Owners of affected devices may be unaware that their device has been compromised, as standard integrity checks and bootloader states do not indicate tampering. This can lead to unauthorized access to personal data, financial loss, and exposure to further attacks. The inability to reliably detect or fix compromised devices undermines user confidence in the security of the Android platform.

4.2 Detection of Modified Firmware

The integrity of an Android device's firmware can typically be verified using the boot state indicator. For instance, executing the command `adb shell getprop ro.boot.verifiedbootstate` will return `green` if the device is in a verified boot state, indicating that all critical partitions have passed integrity checks.

Partitions on Android devices are protected using cryptographic signatures, which are stored as `vbmeta header` and `footer` sections within the partition disk images. The `vbmeta.img` file plays a central role in this process, as it contains a hashtree that represents the integrity of all included partitions. This hashtree is signed with a key which, in our case, is a test key included for development or testing purposes. A `Verified Boot Hash` is computed over all partitions and included in the `vbmeta.img`. This hash serves as a reference for detecting any unauthorized changes to the partitions. If a partition is modified, the hash will no longer match, signaling a potential compromise. However, not all partitions are covered by the `vbmeta.img`. All major vendors tested, including Google, have some partitions that are not included in the `vbmeta.img`. If an attacker possesses the key used to sign `vbmeta.img`-excluded partitions, they can modify these partitions and re-sign them with the same signing key. In such cases, the system will not detect the changes, as the modified partitions will appear valid during verification, and the `Verified Boot Hash` will remain unchanged. This limitation highlights a potential vulnerability in the firmware integrity verification process.

To detect changes in the firmware, even when the hashtree is signed correctly (i.e. the `vbmeta.img` is signed with a leaked or test key), it is necessary to know the exact hashes of all partitions published by the vendor. One way to achieve this is through transparency logs. Google already offers transparency logs within their Android Binary Transparency[13] project, which at the time of writing includes Pixel Firmware Transparency[14] and Google System APK Transparency[15] logs.

4.3 Mitigation Recommendations

As a first step, we recommend that at least all mentioned Android device manufacturers immediately review their firmware and APK signing practices to ensure that no AOSP test keys are used in future production-grade firmware.

We recommend that the Android build process either treats all AOSP test keys as placeholders that cannot be used to compile production firmware, or, at the very least, adds a test at the end of the build process that checks for the presence of test keys anywhere in the firmware. Android's testing frameworks, such as CTS or STS, should be enhanced to identify and flag firmware that uses weak or insecure signing keys, possibly using databases of leaked and insecure private keys. These testing frameworks play a critical role in maintaining platform security by verifying established standards, and by missing checks for the presence of test keys in production firmware, highlighting gaps in the testing process.

The mentioned recommendations can all be implemented in the AOSP code base and therefore we therefore consider them feasible.

The completeness of partition hashes in `vbmeta` is not enforced at the moment and integrity of firmware for other CPUs is out of scope for AVB [13]. The Android keystore attestation process should also be updated to include all cryptographic hashes and signatures involved in the firmware boot process, not just the partitions included in the `vbmeta.img`. This would provide a more complete picture of the device's security state, allowing for better verification of its integrity. One simple solution would be to enable apps to read the full `vbmeta` structure without requiring elevated system privileges. Integrity of these structures can be verified through the existing keystore attestation certificates and their intended content is publicly known from official device images, so these read requests would not seem to require further authentication. However, in combination, this would allow detection of modifications through remote attestation. We argue that it is not feasible to include tests for the completeness of partitions in `vbmeta`, however, including other CPUs cryptographic hashes of the firmware and public keys in the `vbmeta.img` and the attestation certificate would improve the integrity of Android devices. Due to various signature formats of different SoC vendors, this would be challenging to implement in `libavb`.

5 Coordinated Disclosure

Due to the severity of this vulnerability and its impact on multiple vendors—namely Fairphone, Transsion (Tecno, Itel, Infinix), Cubot, and Redmagic—we decided to report our findings to Google using their Bug Hunters program[16], as well as to all affected vendors, within an ethical disclosure window of at least 90 days before disclosure. Our Coordinated Vulnerability Disclosure (CVD) with the ID 416187987 received a High severity and High quality rating from Google's security team. We informed Google of our

[13] https://developers.google.com/android/binary_transparency/overview
[14] https://developers.google.com/android/binary_transparency/pixel_overview
[15] https://developers.google.com/android/binary_transparency/google1p/overview
[16] https://bughunters.google.com/

recommendations mentioned in section 4.3 to be implemented in AOSP. The Tecno Security Response Center[17] responded promptly, acknowledging that the vulnerability was known and has been fixed in new devices since 2023. Fairphone also responded in a timely manner, indicating they would investigate the report. Similar to Tecno, Fairphone uses secure signing keys for the Fairphone 5, which indicates that they are also aware of this issue and fixed it. Cubot and Redmagic did not respond to our report.

6 Conclusion

This paper highlights a critical vulnerability in the Android firmware supply chain that undermines device security and user trust. By exploiting weak signing practices, attackers can bypass integrity checks, modify firmware, and retain user data without detection. Our analysis demonstrates the widespread impact of this issue, affecting multiple manufacturers and device models. We showcased how compromised firmware can pass security checks and appear as bootloader-locked devices, posing significant risks to user privacy and the Android ecosystem.

These findings emphasize the need for stricter firmware signing protocols and enhanced verification mechanisms. Current tools, such as the Compatibility Test Suite and Verified Boot, fail to detect these exploits, leaving devices vulnerable. We recommend immediate action, including eliminating test keys in production firmware, improving keystore attestation, and expanding the scope of integrity checks to cover all partitions.

This work underscores the importance of robust supply chain security and calls for collaborative efforts among stakeholders to address these weaknesses. By implementing the proposed mitigations, the Android ecosystem can strengthen its defenses and restore confidence in device integrity.

Acknowledgments

This work has been carried out within the scope of the LIT Secure and Correct Systems Lab and has partially been supported by Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World. We gratefully acknowledge financial support by the State of Upper Austria, the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association.

References

[1] Android Open Source Project. 2025. Android Verified Boot (AVB) Boot flow. https://source.android.com/docs/security/features/verifiedboot/boot-flow Accessed: 2025-06-10.

[2] Android Open Source Project. 2025. Android Verified Boot (AVB) Device state. https://source.android.com/docs/security/features/verifiedboot/device-state Accessed: 2025-06-10.

[3] Android Open Source Project. 2025. Android Verified Boot (AVB) README. https://android.googlesource.com/platform/external/avb/+/761178607206f4cb2af79ed9eec52d8cbd814adb/README.md Accessed: 2025-06-10.

[4] Android Open Source Project. 2025. File-based encryption. https://source.android.com/docs/security/features/encryption/file-based Accessed: 2025-06-10.

[5] Nguyen Tan Cam, Van-Hau Pham, and Tuan Nguyen. 2017. Sensitive Data Leakage Detection in Pre-Installed Applications of Custom Android Firmware. In *2017 18th IEEE International Conference on Mobile Data Management (MDM)*. 340–343. doi:10.1109/MDM.2017.56

[6] Ronny Chevalier, Stefano Cristalli, Christophe Hauser, Yan Shoshitaishvili, Ruoyu Wang, Christopher Kruegel, Giovanni Vigna, Danilo Bruschi, and Andrea Lanzi. 2019. BootKeeper: Validating Software Integrity Properties on Boot Firmware Images. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy* (Richardson, Texas, USA) *(CODASPY '19)*. Association for Computing Machinery, New York, NY, USA, 315–325. doi:10.1145/3292006.3300026

[7] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2379–2396. https://www.usenix.org/conference/usenixsecurity20/presentation/elsabagh

[8] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. 2019. Hey Google, What Exactly Do Your Security Patches Tell Us? A Large-Scale Empirical Study on Android Patched Vulnerabilities. *CoRR* abs/1905.09352 (2019). arXiv:1905.09352 https://doi.org/10.48550/arXiv.1905.09352

[9] Roee Hay. 2017. fastboot oem vuln: android bootloader vulnerabilities in vendor customizations. In *Proceedings of the 11th USENIX Conference on Offensive Technologies* (Vancouver, BC, Canada) *(WOOT'17)*. USENIX Association, USA, 22. https://dl.acm.org/doi/10.5555/3154768.3154790

[10] Grant Hernandez, Dave (Jing) Tian, Anurag Swarnim Yadav, Byron J. Williams, and Kevin R.B. Butler. 2020. BigMAC: Fine-Grained Policy Analysis of Android Firmware. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 271–287. https://www.usenix.org/conference/usenixsecurity20/presentation/hernandez

[11] Qinsheng Hou, Wenrui Diao, Yanhao Wang, Xiaofeng Liu, Song Liu, Lingyun Ying, Shanqing Guo, Yuanzhi Li, Meining Nie, and Haixin Duan. 2022. Large-scale security measurements on the android firmware ecosystem. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1257–1268. doi:10.1145/3510003.3510072

[12] Qinsheng Hou, Wenrui Diao, Yanhao Wang, Chenglin Mao, Lingyun Ying, Song Liu, Xiaofeng Liu, Yuanzhi Li, Shanqing Guo, Meining Nie, and Haixin Duan. 2023. Can We Trust the Phone Vendors? Comprehensive Security Measurements on the Android Firmware Ecosystem. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3901–3921. doi:10.1109/TSE.2023.3275655

[13] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. 2021. The Android Platform Security Model. *ACM Transactions on Privacy and Security* 24, 3 (April 2021), 1–35. doi:10.1145/3448609

[14] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. 2021. Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization. In *2021 IEEE Symposium on Security and Privacy (SP)*. 87–102. doi:10.1109/SP40001.2021.00074

[15] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2017. BootStomp: On the Security of Bootloaders in Mobile Devices. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 781–798. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/redini

[16] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. San Diego, CA. https://doi.org/10.14722/ndss.2015.23294

[17] Daniel R. Thomas, Alastair R. Beresford, and Andrew Rice. 2015. Security Metrics for the Android Ecosystem. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '15)* (Denver, CO, USA) *(SPSM '15)*. ACM, New York, NY, USA, 87–98. doi:10.1145/2808117.2808118

[18] Elliott Wen, Jiaxing Shen, and Burkhard Wuensche. 2024. Keep Me Updated: An Empirical Study of Proprietary Vendor Blobs in Android Firmware. In *2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*. 116–125. doi:10.1109/ICPADS63350.2024.00025

[19] Yifan Yu, Ruoyan Lin, Shuang Li, Qinsheng Hou, Peng Tang, and Wenrui Diao. 2024. Security Assessment of Customizations in Android Smartwatch Firmware. In *2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 1141–1148. doi:10.1109/TrustCom63139.2024.00162

[20] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, USA, 409–423. doi:10.1109/SP.2014.33

[17] https://security.tecno.com