Exploiting RPMB authentication in a closed source TEE implementation

Aya Fukami[a,b], Richard Buurke[a], Zeno Geradts[a,b]

[a]Netherlands Forensic Institute, Laan van Ypenburg 6, The Hague, 2497 GB, The Netherlands
[b]University of Amsterdam, Science Park 904, Amsterdam, 1098 XH, The Netherlands

Abstract

Embedded Multimedia Cards (eMMCs) provide a protected memory area called the Replay Protected Memory Block (RPMB). eMMCs are commonly used as storage media in modern smartphones. In order to protect these devices from unauthorized access, important data is stored in the RPMB area in an authenticated manner. Modification of the RPMB data requires a pre-shared authentication key. An unauthorized user cannot change the stored data. On modern devices, this pre-shared key is generated and used exclusively within a Trusted Execution Environment (TEE) preventing attackers from access. In this paper, we investigate how the authentication key for RPMB is programmed on the eMMC. We found that this key can be extracted directly from the target memory chip. Once obtained, the authentication key can be used to manipulate stored data. In addition, poor implementation of certain security features, aimed at preventing replay attacks using RPMB on the host system can be broken by an attacker. We show how the authentication key can be extracted and how it can be used to break the anti-rollback protection to enable data restoration even after a data wipe operation has been completed. Our findings show that non-secure RPMB implementations can enable forensic investigators to break security features implemented on modern smartphones.

Keywords:
RPMB, replay attack protection, TEE, mobile forensics

1. Introduction

With the implementation of multiple types of security features on smartphones, extracting user data from them at a digital forensics lab is becoming more challenging day by day. The details of these features are not always disclosed, requiring digital forensic investigators to reverse engineer the device both on the software and hardware side. The data on modern smartphones is encrypted by default, preventing data carving through physical data acquisition. Additionally, unlocking the device requires knowledge of a user secret, such as a passcode or password. On top of those security features, data wiping routines are common on modern smartphones, which are triggered when a certain threshold of failed password attempts is reached.

To keep track of the state of the device, smartphone manufacturers often make use of the Replay Protected Memory Block (RPMB) in an embedded Multimedia Card (eMMC). eMMC is a popular storage memory in smartphones. Writing to the RPMB partition requires authentication, therefore a pre-shared secret key is needed every time the host device wants to modify information. To prevent an attacker from rolling back system data to an older version, the smartphone System on Chip (SoC) stores information in the RPMB area where data integrity is guaranteed. The use of RPMB has been suggested to prevent replay attacks on Android-based devices [1] [2], and other Linux based systems [3]. The security of data stored in RPMB relies on the secure storage of the pre-shared secret key.

While the use of RPMB for device protection against unauthorized access has been discussed in literature, the actual security of the RPMB implementation on memory devices has, to the best of authors' knowledge, not been widely researched. By using a real world example, whereby we accidentally triggered the wipe routine, we show how to recover a wiped-state smartphone which implements RPMB anti-rollback protection. Through hardware reverse engineering, we identified that the RPMB authentication key is stored in flash memory in an accessible manner, making it possible for the key to be extracted. Once the key is extracted, the RPMB data becomes editable, thereby losing its integrity. We demonstrate how the RPMB authentication key can be extracted, and how we can rollback a device in a wiped state to a working state, by restoring a flash data backup and modifying data stored in the RPMB.

This paper mainly makes the following contributions:

- We show the RPMB authentication key can be extracted from an eMMC
- We provide a detailed description of a closed source TEE implementation that utilizes the RPMB area for full-disk encryption
- We experimentally demonstrate that a smartphone, of which the wipe routine has been triggered, can be recovered by recovering the RPMB authentication key, and restoring a flash data backup

The rest of the paper is organized as follows. In section 2, we provide the necessary background on the RPMB area and its

Email address: a.fukami@uva.nl (Aya Fukami)

use in smartphones. We then explain the implementation of the trusted execution environment in detail, focusing on the usage of the RPMB, in an actual target device in section 3. In section 4, we demonstrate the actual procedures to restore a wiped-state smartphone by exploiting the RPMB data. We discuss the impact of our attack in 5 followed by the related work in section 6, before concluding in section 7.

2. Background

2.1. Replay Protected Memory Block

Replay Protected Memory Block (RPMB) is a memory block implemented in JEDEC standards for modern storage devices, such as embedded Multi Media Card (eMMC) and Universal Flash Storage (UFS) [4, 5]. In this paper, since the target device has an eMMC-based non-volatile memory chip, we define the RPMB as a memory block in an eMMC, unless otherwise specified. The RPMB area is provided to store data in an *"authenticated and replay protected manner"* [5]. Authentication is performed by utilizing an authentication key, which is the shared secret between the eMMC and the host system. At the time of manufacturing a smartphone, the authentication key is programmed to the device and to the eMMC in a secure environment before it is shipped from the factory.

When the host system wants to write the RPMB, the host must calculate a SHA-256 hash based message authentication code (HMAC) over the message to be sent, using this authentication key. The message includes the data to write, and the write counter. The write counter of the RPMB represents the total number of successful authenticated write operations. This value is incremented by one every time an authenticated data write operation is performed. The counter value is stored in the eMMC in an area inaccessible via the external interface to prevent it from being reset. The write counter provides protection against replay attacks. If it is not implemented, an attacker can monitor the communication between the host and the eMMC, and then reproduce the same communication at a later time, allowing modification of data stored in the eMMC.

Once a message is received, the eMMC calculates the HMAC over the received message using its own stored authentication key to check if the message is from the authenticated host. Only when the calculated HMAC matches with the received one, and the provided counter matches the stored value, a data write operation to the RPMB is authorized. Since a RPMB data write operation requires the correct authentication key and write counter value, the stored data can be tamper-resistant and can be protected against replay attacks.

2.2. RPMB Use Cases in Digital Devices

Since the RPMB area is a tamper-resistant memory block, it is often used in modern digital devices to store information that can help prevent unauthorized access to the system [2, 6, 7, 8]. Common use cases include anti-rollback protection, unlock protection, and secure data storage.

2.2.1. Anti-Rollback Protection

Smartphone manufacturers are constantly upgrading software running on their products to patch reported vulnerabilities. Once software is upgraded, the device is generally not allowed to downgrade to the previous version. This is to protect user data from unauthorized access by exploiting known vulnerabilities. To keep track of the latest software version, the RPMB area can be used to store the version-related data. When a user attempts to install a piece of software to the target device, the host system checks the current version of the software using the data stored in the RPMB. If the stored software version is higher than the one to be installed, the system rejects the software installation.

2.2.2. Unauthorized Device Unlocking Prevention

Another use case of RPMB is unauthorized unlock prevention. To prevent unauthorized access, a user can lock the device with a unique passcode or password. The passcode/password is stored in the device at creation time. Once getting physical access to the device, an attacker might try all possible combinations to unlock it. This type of attack can be automated through the use of software. To prevent brute-forcing, RPMB can be used to store a counter that keeps track of the failed number of password attempts. If the counter exceeds a certain threshold, the system can initiate a security measure, such as wiping user data or enforcing an increasingly long wait time. Since data in RPMB cannot be overwritten without the authentication key, the attacker cannot reset or decrement the failed attempts counter.

2.2.3. Secure Data Protection

In modern smartphones, the integrity of all critical software components is enforced by secure boot. Different variations exist, but the general approach is to store the hash of a certificate, containing a public key, in immutable memory. The stored hash serves as a root of trust, since it authenticates the public key that verifies the signature of the next component in the boot chain.

RPMB can also be used as a root of trust, since it is considered authenticated storage. As an example, the Android Verified Boot (AVB) public key, which authenticates the bootloader, can be stored in RPMB by the Trusty Trusted Execution Environment (TEE) [9].

2.3. Target Device

During our forensic analysis at the forensic lab, we accidentally triggered the wipe routine on a Blackphone 2 while developing a brute-force method. Restoring a backup of the flash data did not return the device to a working state, therefore extensive research on the implemented anti-rollback protection was needed. The Blackphone 2 is a smartphone focused on security and privacy, first produced in 2015. The device itself is built around the Qualcomm Snapdragon 615 (MSM8939) SoC and runs a modified version of the Android operating system called "Silent OS". In 2016 Silent Circle released "Silent OS 3" based on Android 6.0.1 (Marshmallow), which was the last version available for the Blackphone 2. The user data of the Blackphone 2 is protected using full disk encryption (FDE). The FDE

implementation on Android devices is vendor specific, but they commonly require the user password and a hardware bound key to derive the correct decryption key. On top of the encryption, the Blackphone 2 seemed to use an anti-rollback counter stored in the RPMB area to prevent an attacker from restoring a data backup after a device had been wiped. Because we needed to break this security feature, and had to restore the device to a working condition, we selected the Blackphone 2 as our target for researching RPMB authentication.

3. Dissecting the Use of RPMB on the Target Device

We applied various software and hardware reverse engineering techniques to determine the use of RPMB on the target device. The software reverse engineering effort focused on the RPMB implementation in the TEE and FDE key derivation scheme. We also analyzed the physical architecture of the eMMC chip to recover the RPMB pre-shared key to enable write access to the RPMB partition.

3.1. Software

The implemented FDE scheme on the target device, including the use of an anti-rollback counter in the RPMB partition, is vendor specific. We used static and dynamic analysis techniques to determine the key derivation process and the role of the counter value. The key derivation process is implemented almost exclusively in the Qualcomm Secure Execution Environment (QSEE) for which, to our knowledge, no source code is currently publicly available.

3.1.1. Qualcomm Secure Execution Environment (QSEE)

Qualcomm TrustZone technology enables the separation of a non-secure operating system (e.g. Android) and a secure operating system such as QSEE on the same device. TrustZone technology is implemented according to the “Advanced Trusted Environment: OMTP TR1” [10] standard, and therefore has mitigations against software and hardware attacks. The non-secure operating system is said to be running in the Rich Execution Environment (REE) or “normal world”. The secure operating system runs in the Trusted Execution Environment (TEE) or “secure world”. This separation ensures that certain operations can be performed securely even when an attacker compromised the Android operating system. The secure operating system has full control over the device, while the normal operating system can only access non-secure memory assigned to it. This separation is not only enforced by the memory management unit (MMU) in the application processor (AP) but also on the data bus itself by the TrustZone Address Space Controller (TZASC) [11].

Code executed on the AP runs at different privilege levels, defined by ARM as “exception levels”. Those levels are used both in the REE and the TEE as follows:

- EL0: User space
- EL1: Supervisor
- EL2: Hypervisor
- EL3: Secure channel monitor

The secure channel monitor is used to relay messages between the REE and the TEE. Only the Linux kernel running at EL1 (or the hypervisor) is permitted to send messages to QSEE through the secure monitor. Therefore, normal applications can only communicate with QSEE through the Linux kernel. Fig. 1 shows the flow of message from the Linux kernel in the REE to QSEE in TEE. QSEE can also run trusted applications (TA’s), shared libraries and drivers (TD’s). We refrain from going into details of those features since it is outside the scope of this paper.

The QSEE kernel implements a handler that executes a function according to the secure monitor call (SMC) identifier sent by the Linux kernel. There is also a separate command handler for processing requests originating from TA’s. On our target device all relevant functionality, concerning key derivation and user authentication, is implemented within the QSEE kernel itself.

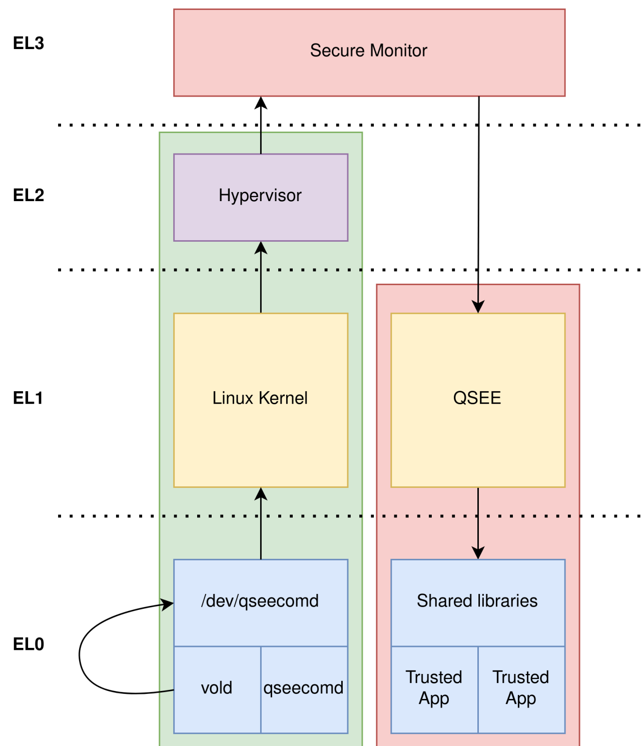


Figure 1: A simplified overview of the TZ architecture. Left: REE (Green) Right: TEE (Red)

3.1.2. Android Volume Daemon

The Android Volume Daemon (*vold*) is responsible for mounting storage media, including the *userdata* partition. The operating system on the target device shared many similarities with CyanogenMod 13, of which multiple relevant code repositories are available on GitHub [12, 13, 14].

By analyzing the available source code, we determined that the first thing *vold* does after the user entered their password

is to check the encryption type stored in the footer partition. When set to *aes-xts*, the Qualcomm specific implementation is used exclusively, instead of including the Android Keymaster Hardware Abstraction Layer (HAL).

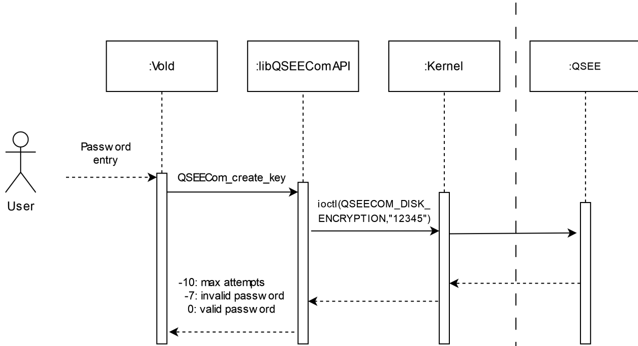


Figure 2: A simplified overview of the vold function.

The *vold* process is linked to the *libQSEEComAPI.so* shared library which provides communication with QSEE through the Linux kernel. It uses the */dev/qseecom* special device which is exposed by the Linux kernel for communication with QSEE. Once communication between the *vold* process and QSEE is established through */dev/qseecom*, the *QSEECOM_IOCTL_CREATE_KEY_REQ ioctl()* call is executed with the disk encryption type and user password as the arguments, as shown in Fig. 2. The kernel function handling this request forwards it to QSEE by executing an SMC. As shown in Section 3.1.1, the secure channel monitor, which runs at the highest possible privilege level (EL3), handles this communication between the Linux kernel and QSEE. Once the request is received by QSEE, one of the values shown in Table 1 is returned through *ioctl()* depending on the password attempt status.

Table 1: Returned values from ioctl() after password attempt

Return value	Decimal	Meaning
0xFFFFFFF6	-10	Max. password attempts reached
0xFFFFFFF9	-7	Invalid password attempt
0	0	Correct password

When the correct password has been entered, the decryption key is set in the crypto engine (CE) of the SoC. This ensures the AP can never read the key used in the decryption process. Communication with the CE happens exclusively within QSEE. The *dm-crypt* device mapper can only decrypt the *userdata* partition after the correct key is set in the CE.

3.1.3. RPMB Usage in User Authentication and Key Derivation

QSEE uses a data structure called the *keystore* for securely storing security sensitive information, including the FDE key and the number of failed password attempts. Fig. 3 shows the communication flow between the Android and QSEE during user authentication procedures. The keystore itself is saved in the partition named *SSD* on flash. This is a proprietary partition specified by Qualcomm. The partition data is encrypted using a key derived from a hardware bound key (HBK). The *tz_ks_ns_generate_key* function residing in QSEE is responsible for adding the FDE encryption key to the keystore if it does not already exist. If the keystore has not yet been loaded in memory, it is read from flash storage first. QSEE uses a combination of listeners and shared memory buffers to transfer data from flash memory and the RPMB partition by communicating with the *qseecomd* user space process. The encrypted keystore consists of one or more entries, starting with a header including an HMAC used to authenticate the encrypted data. The HMAC is calculated over the encrypted data and also includes the anti-rollback counter stored in the RPMB partition of the target eMMC. Authentication will fail if the anti-rollback counter cannot be read from the eMMC, or if the read counter value is incorrect. Once authentication succeeds, the encrypted entry data is decrypted using a key derived from the HBK.

When the keystore entries are authenticated and decrypted, the kernel tries to load the FDE key into the CE by calling the *tz_ce_pipe_key_select_ns* function in QSEE. This function calls *tz_ks_dy_get_key*, which performs the actual key derivation. If the user enters thirty wrong passwords, the FDE key is removed from the keystore. The keystore is then re-encrypted and written back to flash. The anti-rollback counter is also incremented and written back to the RPMB partition. This prevents an attacker from writing back an earlier version of the keystore since the entries it holds can no longer be authenticated.

When QSEE is loaded, it checks if the RPMB authentication key is programmed in the eMMC. If this it not the case, it starts a provisioning routine, which first checks an eFuse value to determine if an authentication key has been programmed before. If the authentication key has never been programmed on the eMMC, the SoC executes the key programming procedure. This procedure is performed only once in the device's lifetime.

The reason for this behavior is most likely because the authentication key is derived from a number of static values, encrypted by the CE using the AES algorithm with a hardware bound key. This means that the authentication key is always the same. The authentication key is dynamically generated when needed and never stored. If the key were to be programmed multiple times, an attacker might be able to extract the authentication key by replacing the eMMC whose RPMB partition has never been programmed, and monitor the key programming procedure.

The QSEE initialization routine also reads the current write counter value directly from the eMMC, and stores it in volatile memory. The counter value does not appear to be stored in any type of non-volatile memory for later use. If an RPMB write request executed by QSEE fails, the write counter is read again from the eMMC and updated in memory accordingly. We did not find any evidence that the write counter is ever checked against a stored value to detect tampering with RPMB data.

3.2. Hardware

The target device uses a Hynix embedded Multichip Package (eMCP) H9TQ26ADFTMCUR as its storage memory. The eMCP chip has the eMMC and DRAM in one package. In this section, we describe how we identified the hardware structure

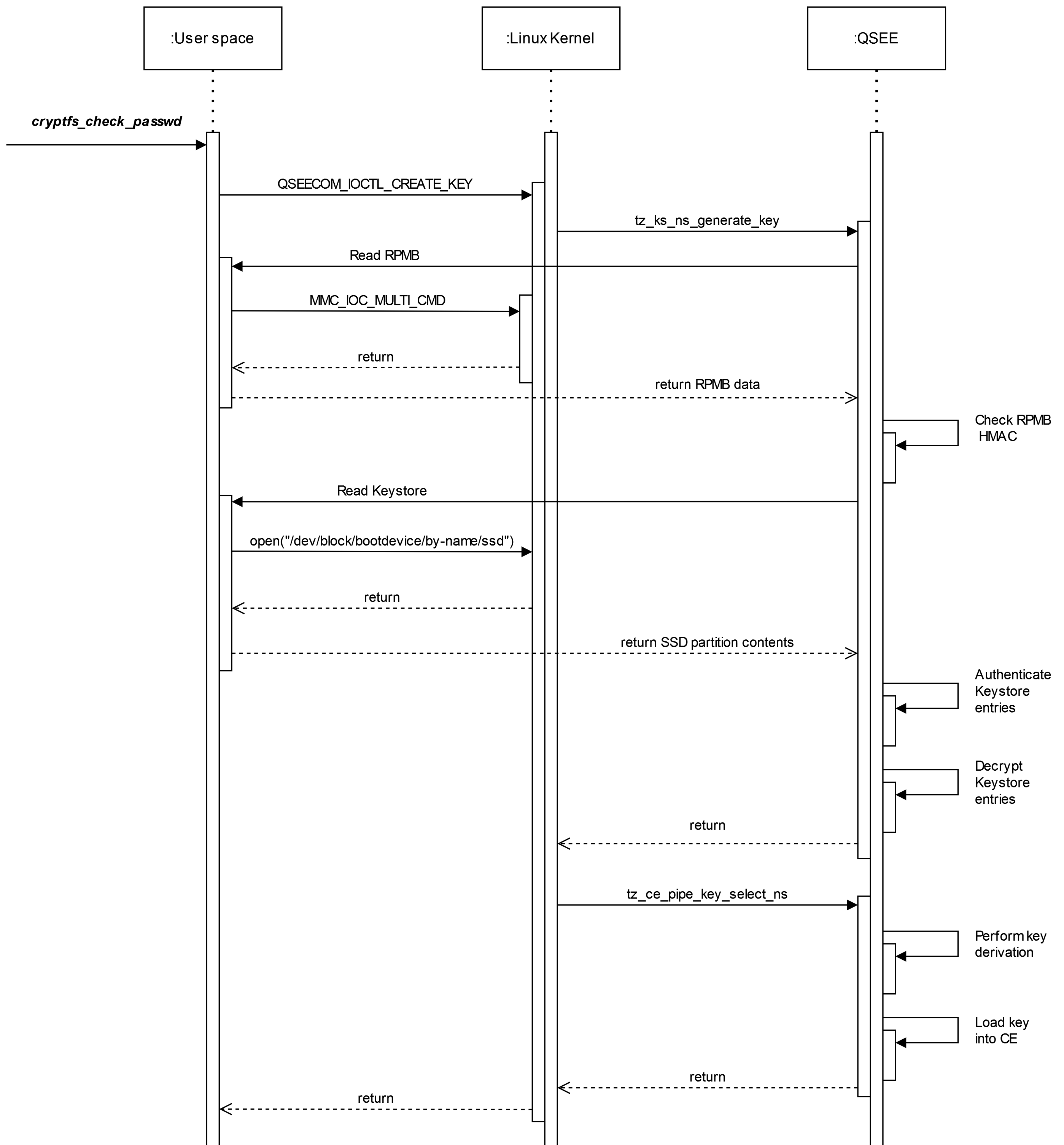


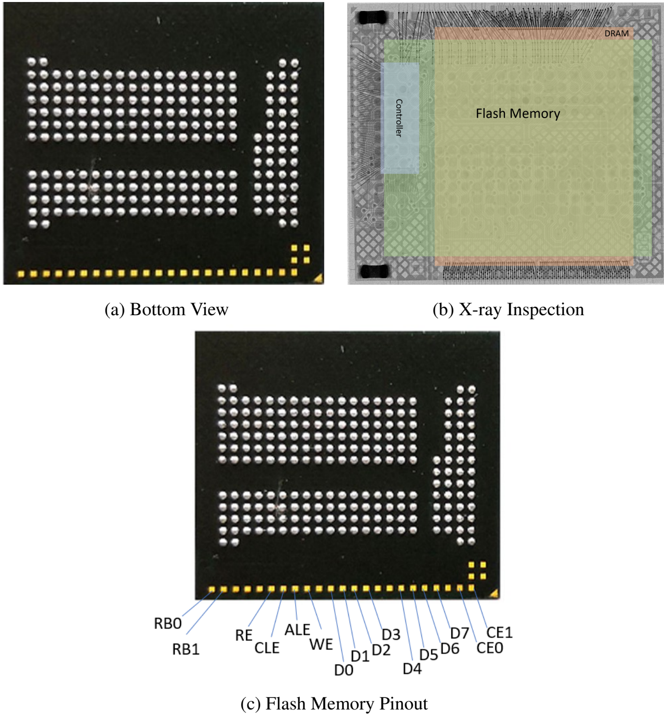
Figure 3: Communication between Android and QSEE for user authentication

of the chip and identified how the RPMB authentication key is stored in flash memory.

3.2.1. eMMC Structure

The eMMC embedded in an eMCP consists of managed NAND flash memory. The dedicated flash memory controller is embedded in the chip together with flash memory and provides an external interface following the JEDEC eMMC standard [4]. Therefore the SoC can access the storage memory by using JEDEC standardized commands. Since the internal flash controller manages wear leveling, error correction, and other operations required for interfacing with flash memory, the SoC does not need to implement flash memory vendor specific commands. Instead it can use the JEDEC interface as a hardware abstraction layer to flash memory.

We started observing the internal structure of the chip through radiographic inspection. We obtained a few eMCPs which have the same part number as the target device, and worked on those reference chips to perform reverse engineering. Fig. 4b shows the X-ray image of the target chip. The location of each silicon die is annotated. By tracing the electrical paths of each die, one can find that the flash memory controller and DRAM are connected to the standard interface. The standard interface is established through the silver pads shown in Fig. 4a. On the other hand, flash memory dies are not exposed to those pins. Instead they are traced to the *technical pins* [3]. The pinout of the technical pins was identified as shown in Fig. 4c.



(a) Bottom View

(b) X-ray Inspection

(c) Flash Memory Pinout

Figure 4: Hynix H9TQ26ADFTMCUR visual observation

To monitor the controller behavior while the RPMB authentication key is being programmed, we connect the technical pins to a logic analyzer. This setup is shown in Fig. 5. The communication between the flash memory controller and flash memory, while programming the authentication key, was captured using this method. Using the *mmc-util* package with the *rpmb write-key* option [15], we programmed the key "1234567890 1234567890123456789ABC" (in ASCII) to a reference eMCP chip. Meanwhile, the logic analyzer is configured to capture the communication once the Command Latch Enable (CLE) pin of the flash memory is enabled.

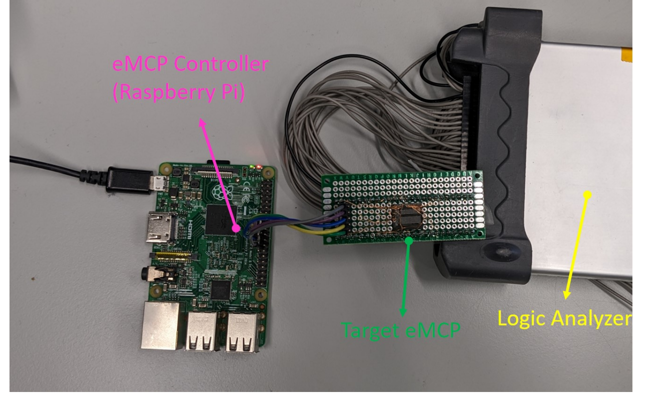
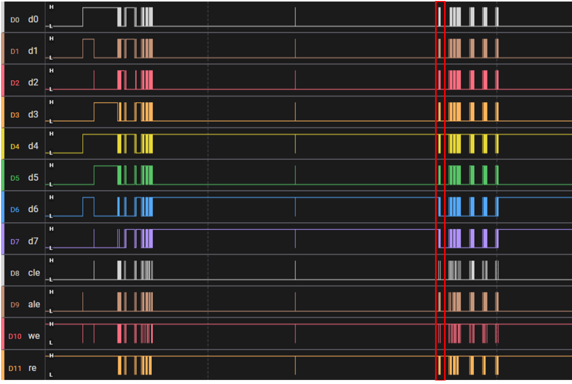


Figure 5: Hardware setup to monitor communications between flash memory and the flash memory controller.

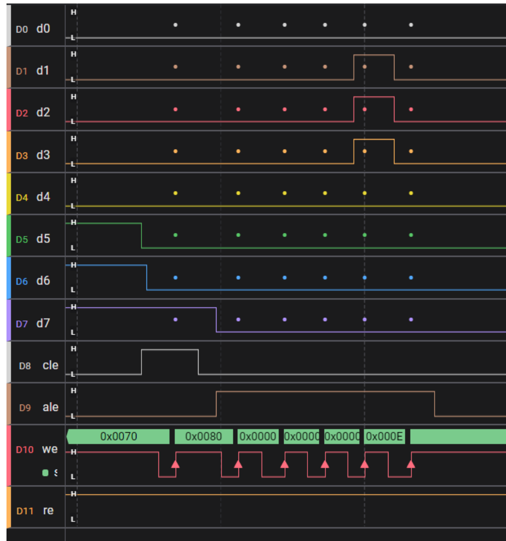
The captured result is shown in Fig. 6a. Multiple communications are captured between the controller and flash memory. During the first half of the monitored communications, multiple read commands are issued from the controller to various addresses in flash memory. We assume that the controller is checking the current status of the RPMB during this phase. After reading the values from flash memory, the controller starts writing values to flash memory. For example, if we zoom in the red square shown in Fig. 6a, we find the command shown in Fig. 6b. Command 0x80 is issued followed by the address data while the Address Latch Enable (ALE) pin is pulled high. According to the ONFI standard [16], command code 0x80 is defined as a page program operation, as shown in Fig. 6c. In the captured communication, the page program operation is issued to the flash memory page address 0x0E00.

3.2.2. Physical Memory Dump and Authentication Key Extraction

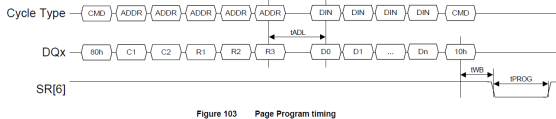
Now with the purpose of identifying what data is written to the flash memory while programming the authentication key, we connect the technical pins to a flash memory reader. The read operation was performed using single-level-cell (SLC) mode provided by Rusolut Visual NAND Reconstructor [17]. Given the reliability issues with multi-level-cell (MLC) flash memory, which stores more than 2 bits of data per flash memory cell, flash memory manufacturers offer SLC mode in MLC flash memory. When set to SLC mode, the flash memory cell is forced to store only 1 bit per cell at the specified address. This way, the host can store important data, such as system data, to the SLC area in a reliable manner. The SLC mode data read/write operation is implemented as a proprietary command, therefore the implementation details differ per manufacturer.



(a) Communication between the flash memory controller and flash memory in the eMMC



(b) Zoomed in flash memory command captured during the authentication key programming



(c) Data write command timing defined by ONFI [16]

Figure 6: Captured communication between the flash memory controller and flash memory

The extracted raw flash memory data is XORed with a scrambling pattern. The scrambling pattern is extracted from the reference device, where all plain-text data is 0x00. Reverse-engineering of the scrambling pattern can also be performed [18, 19], however for our experiment, identifying one page of the pattern was enough to extract the required key information. Using the scrambling pattern from the empty sectors, the plain text data was successfully extracted. By analyzing the de-scrambled data, we found that the authentication key data is stored in plain text after the flag `[PASS]` as shown in Fig. 7.

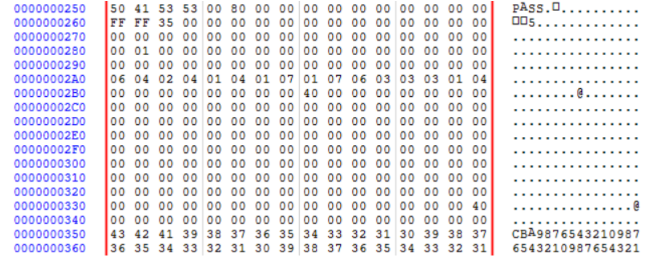


Figure 7: RPMB key stored plain-text in flash memory.

4. Restoring the Wiped-state Smartphone by Exploiting the RPMB

Based on the results we obtained through reverse engineering, it is clear that the RPMB data is used to store the anti-rollback counter. Once the investigator extracts the RPMB authentication key from the eMMC, it allows the investigator to modify the RPMB content, which makes the anti-rollback protection compromised. In order to evaluate our findings and apply them to the case devices, we executed an arbitrary data-wipe operation and experimentally performed a data restore operation on a Blackphone 2. To reproduce our accidental data wipe incident, we performed wrong password attempts in an automated manner until the data wiping operation was initiated, while monitoring how the data was modified on the target device.

4.1. Device Setup

4.1.1. Hardware Modification

To perform the required eMMC read and write operations, some hardware modifications were required on the target device. First, the eMCP chip was detached from the PCB by using a heat gun and melting the underlying solder between the eMCP chip and the circuit board of the target device. The eMCP was then connected to a Linux based computer through an eMCP-SD adapter in order to check its contents. The result of the *dmesg* command showed that the target eMMC is recognized as *mmc0*, and it contained the following four physical partitions:

- *mmcblk0*
  The main partition, and had a size of 29GB with Android related data, which is partitioned into 32 partitions.
- *mmcblk0boot0*
  4KB in size, all bytes were 0x00.
- *mmcblk0boot1*
  4KB in size, all bytes were 0x00.
- *mmcblk0rpmb*
  RPMB partition, 4KB in size.

Running *mmc-utils* with the *rpmb read-counter* option showed that the RPMB had been written 155 times. Following the NIST guideline [20], we performed image acquisition from all partitions, and saved the images as our baseline data.

The flash memory interface (technical pins) of the target chip was then connected to a flash memory reader to extract the RPMB authentication key. The key is stored at the same address as the reference eMCP as discussed in Section 3.2.2. In order to keep track of data modification on the RPMB while triggering the wipe routine, header pins for In System Programming (ISP) were installed at the same time as re-mounting the eMCP onto the PCB. Specifically, the CLK, CMD, and D0 lines of the eMMC were extended using thin wires. With this setup, the eMMC data can be accessed directly without detaching the chip from the PCB. This way we can read and write the RPMB partition directly.

4.1.2. Software

As discussed in Section 3.1.1, root privileges are required to enable communication with QSEE through the secure channel monitor. Therefore, we first need to find a way to run unsigned code. On our target, including other old Android devices, part of the device configuration is stored in the *devinfo* partition, including the unlock state of the bootloader. As reported by Hay [21], by modifying the values at offsets 0x10 and 0x18 in this partition to 0x01, the bootloader is considered unlocked. Through this modification, an attacker can run arbitrary code on the target device with EL0 or EL1 privileges.

Qualcomm devices can boot into "Emergency Download Mode" (EDL) either by holding a specific button combination or automatically when the initialization of certain hardware components fail. This mode is used for diagnostic purposes by uploading a signed secondary boot stage, also called a "programmer" or "loader" enabling custom code execution. We used a leaked programmer [22] to read/write to eMMC memory and replace the *devinfo* partition with our modified version. This operation can also be performed physically since the eMMC is connected to ISP pins.

Next, we created a modified boot image using Magisk, which starts an Android Debug Bridge (ADB) shell with root privileges without booting the Android operating system. The *fastboot* mode provided by the Android bootloader enables the user to run a custom boot image. Using this method, we acquired root privileges on the device and could communicate with QSEE from user space through *ioctl()* calls.

4.2. Initiating Data Wipe

After establishing a baseline for our target device, we then unlocked the bootloader and started a custom boot image using the method described in Section 4.1.2. By connecting to the device using the ADB, we were able to perform multiple password attempts using the command `vdc cryptfs checkpw <password>`. After 30 failed attempts, the wipe routine in QSEE was executed, the device rebooted and started erasing data.

After the target device completed the wipe routine, we made another physical image of the eMMC including the RPMB partition through the use of ISP. Fig. 8 shows the difference in data stored in the RPMB partition before and after the data wiping routine. The value stored at offset 0x20C has been incremented by 1 from 0x10 to 0x11 after the device was wiped.

After observing this data modification triggered by the data wipe routine, we restored the whole contents of the eMMC main partition, which was 29GB in size, as mentioned in Section 4.1.1. This hardware partition contains the entire filesystem including the *SSD* partition, containing the encrypted keystore. The device booted normally. However, after entering the correct password, the device showed a notification that the password was correct, but the data could not be decrypted, as shown Fig. 9.

This behaviour validates our reverse engineering findings, since we cannot boot the device even after restoring the original keystore data. At this point, only the incremented anti-rollback counter stored in the RPMB partition differed from the original eMMC data.

4.3. Data Restore

To recover the target device, we again restored the whole eMMC contents to its original state. This time, we also restored the RPMB partition data to the one acquired in section 4.1.1. In case of the RPMB data, only the anti-rollback counter at offset 0x20C was effectively changed to 0x10. It is worth mentioning that the value at offset 0x20C was incremented again by 1 to 0x12 by booting the device with the restored eMMC data and the wrong RPMB partition data, as explained in Section 4.2. By rewriting the RPMB partition data with the RPMB authentication key extracted from the eMMC, the write counter value stored in the eMMC was also incremented. Nevertheless, our tampering remained undetected by the SoC, enabling us restore the target device to the state it was before the wipe routine was performed. The target device booted successfully, with its original data intact.

5. Discussion

5.1. RPMB Authentication Key Storage in the eMMC

As shown in section 3.2.2, the RPMB authentication key can be extracted by reading the internal flash memory of the target eMMC. Since the key is stored in plain text with no obfuscation or read protection, the key data is essentially accessible by attackers. While the flash memory interface is not exposed on eMMCs, it is still accessible through *chip-off* analysis, which has been popularly performed in digital forensic analysis [18, 3]. The authentication key of the target device was stored at the same location as the reference device. We also found that the key data is duplicated at multiple locations. Using a similar technique, the authors have successfully extracted RPMB authentication keys from a Samsung KLMAG2GE4A-A001 and Sandisk SDIN8DE4-16G, which were used in other smartphones. Each model uses different address to store the authentication key information. Therefore sniffing the flash write operation during the key programming is necessary. In an example where the RPMB is used to prevent a software downgrade, an attacker can edit the version information stored in the RPMB to a lower value, following our procedure. Then an attacker can downgrade the running software with the purpose of exploiting a known vulnerability. Once exploited, the target

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
000001E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000200  01 00 00 00 01 00 00 00 50 54 42 4C 10 00 00 00  ........PTBL....
00000210  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000220  D3 C2 0D 30 84 1E 5F 55 00 00 00 00 00 00 00 00  ÓÂ.0„._U........
```

(a) RPMB data before user data being wiped.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
000001E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000200  01 00 00 00 01 00 00 00 50 54 42 4C 11 00 00 00  ........PTBL....
00000210  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000220  D3 C2 0D 30 84 1E 5F 55 00 00 00 00 00 00 00 00  ÓÂ.0„._U........
```

(b) RPMB data after user data being wiped.

Figure 8: RPMB data comparison

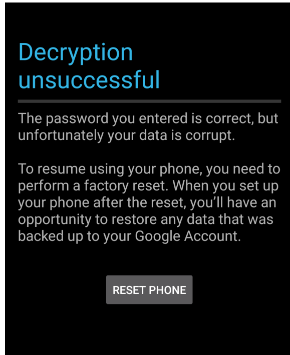


Figure 9: Decryption unsuccessful

device is no longer secured against unauthorized access. The attack itself requires de-soldering of the eMMC chip with hot air, and microsoldering with thin wires, which can be done by a skilled engineer. Therefore the attack is feasible on other digital devices containing eMMCs with an RPMB.

In JEDEC Standard [4], it is defined that the authentication key should be stored in a "one time programmable" authentication key register, which cannot be overwritten, erased or read. Nevertheless, there exists commercially available products where the RPMB key can be deleted and the write counter can be cleared. Therefore the hardware implementation of the RPMB in eMMCs is not always secure.

5.2. *Use of the RPMB Write Counter*

As discussed in Section 2, authentication of RPMB write is performed by computing the HMAC over the message which includes the RPMB write counter. The RPMB write counter is implemented as one of the security features to prevent attackers from performing a replay attack. Therefore, even if the RPMB authentication key is leaked, once the write counter value does not match, we would expect the authentication to fail. After we edited the RPMB partition, the RPMB write counter of the eMMC was incremented, as mentioned in Section 4.3. Therefore we expected that the QSEE would detect tampering of the RPMB partition data. However, it turned out that QSEE always uses the write counter provided by the eMMC, and even re-requests it from the eMMC when an RPMB write fails. In the end, the RPMB write counter is not used as part of the authentication scheme, allowing us to arbitrarily edit the RPMB data. This design decision might be made because it would not lock out the user when a non-malicious mismatch occurs, for example because of data corruption. Further research is required to determine if this is a valid concern.

5.3. *RPMB Authentication Key Generation*

The way the RPMB authentication key is currently generated on the target device, does not prevent an attacker from swapping the eMMC chip. Since the authentication key is hardware bound, only written once, and assumed to be inaccessible, it is supposed to be tied to a single eMMC. Meaning that if the eMMC fails, it cannot be replaced. However, even if the SoC stores the write counter value and detected our tampering with the RPMB data, it would be possible to reprogram another chip with the correct authentication key, and increment its write counter to the desired value. This would work since currently the SoC cannot distinguish different eMMC chips. One way to make the device more resilient against the proposed attack might be to derive the RPMB authentication key from the eMMC device-specific information such as Card Identification (CID) register, which is an unique identification number assigned to each eMMC chip. The SoC may also keep track of eMMCs that it programmed before, to prevent leaking the key. Ultimately, however, the authentication key should be stored in immutable non-volatile memory (such as *eFuses*) within the controller to prevent it from being read directly. Without this measure, all other mitigations onOly add an additional level of complexity but with only a trivial increase in security.

5.4. *Responsible Disclosure*

The authors have reached out to Silent Circle and SK Hynix in June 2023 and July 2023, respectively, to report our findings and to share the possible vulnerabilities.

6. Related Work

Western Digital published a white paper on vulnerabilities in the eMMC RPMB [23] in 2020, and suggested that by performing a "man-in-the-middle" attack, one can trick the host system to make it behave as the intended data has never been written to the RPMB area. This attack would only work to prevent the anti-rollback counter from being updated. It is not feasible in our scenario since the device is already in a wiped state.

Skorobogatov showed that password brute-forcing on a smartphone is possible by mirroring the whole content of the storage memory and restoring its state to reset the number of failed password attempts [24]. Theoretically, the same attack works on our target. However, since our target was in a wiped-state protected with RPMB anti-rollback authentication, an additional RPMB exploit is required to restore the device to a working state to enable the reported brute-force attack.

Multiple use cases of RPMB in smartphones are suggested through literature. [25, 3, 2]. However, none of them have

looked at the actual hardware implementation of the RPMB key storage on the eMMC side.

7. Conclusion

We show that the current RPMB implementation on a smartphone can be exploited due to a non-secure implementation of the RPMB on the hardware and its use in software. We successfully extracted the RPMB authentication key from the target device. By writing back the original data and modifying the RPMB data, we were able to restore the wiped-state smartphone back to a working state. Our proposed attack can be expanded to other smartphones using the RPMB for anti-rollback protection. Given that the anti-rollback protection is a generic implementation used by the SoC manufacturers, we expect that our method is applicable on a wider range of smartphones. We have observed that other manufacturers make use of the RPMB partition on newer devices (e.g. Google/Samsung/Xiaomi), which can be the subject of further research.

References

[1] Android Open Source Project, Keymaster functions (2022). URL https://source.android.com/docs/security/features/keystore/implementer-ref

[2] A. K. Reddy, P. Paramasivam, P. B. Vemula, Mobile secure data protection using emmc rpmb partition, in: 2015 International Conference on Computing and Network Communications (CoCoNet), IEEE, 2015, pp. 946–950.

[3] D. Giese, G. Noubir, Amazon echo dot or the reverberating secrets of iot devices, WiSec '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 13–24. doi:10.1145/3448300.3467820. URL https://doi.org/10.1145/3448300.3467820

[4] JEDEC Solid State Technology Association, Embedded multi-media card (e·MMC) electrical standard (5.1), JEDEC Standard JESD84-B51 (February 2015). URL https://www.jedec.org/system/files/docs/JESD84-B51.pdf

[5] JEDEC Solid State Technology Association, Universal flash storage (ufs) version 2.2, JEDEC Standard JESD220C-2.2 (August 2020). URL https://www.jedec.org/system/files/docs/JESD220C-2_2.pdf

[6] Einav Zilberstein and Adi Klein, e.mmc security methods, White paper (2021).

[7] J. Wiklander, Secure storage in op-tee, linaro Connect (2017). URL https://static.linaro.org/connect/sfo17/Presentations/SFO17-309\%20Secure\%20storage\%20updates.pdf

[8] J. Yao, V. Zimmer, Configuration, in: Building Secure Firmware: Armoring the Foundation of the Platform, Apress, Berkeley, CA, 2020, pp. 383–431. doi:10.1007/978-1-4842-6106-4_11. URL https://doi.org/10.1007/978-1-4842-6106-4_11

[9] Digi International Inc., Connectcore 8x documentation portal (2023). URL https://www.digi.com/resources/documentation/digidocs/embedded/android/dea11/cc8x/android-trustfence_c_key-summary.html#avb-keys

[10] Omtp advanced trusted environment tr1 v1.1 (May 2009).

[11] A. Limited, Arm corelink tzc-400 trustzone address space controller technical reference manual (2014). URL https://developer.arm.com/documentation/ddi0504/c/?lang=en

[12] LineageOS (2012). [link]. URL https://github.com/LineageOS/android_kernel_oppo_msm8939/tree/cm-13.0

[13] LineageOS (2014). [link]. URL https://github.com/LineageOS/android_vendor_qcom_opensource_cryptfs_hw/tree/cm-13.0

[14] LineageOS (2015). [link]. URL https://github.com/LineageOS/android_system_vold/tree/cm-13.0

[15] A. Altman, U. Hansson, Mmc tools (mmc-utils) (2023). URL https://git.kernel.org/pub/scm/utils/mmc/mmc-utils.git

[16] ONFI, Open nand flash interface specification, Tech. rep. (2021). URL http://www.onfi.org/

[17] ruSolut, VNR software. URL https://rusolut.com/visual-nand-reconstructor/vnr-software/

[18] A. Fukami, S. Sheremetov, F. Regazzoni, Z. Geradts, C. De Laat, Experimental evaluation of e.mmc data recovery, IEEE Transactions on Information Forensics and Security 17 (2022) 2074–2083. doi:10.1109/TIFS.2022.3176187.

[19] J. P. van Zandwijk, A Mathematical Approach to NAND Flash-Memory Descrambling and Decoding, Digital Investigation (2015).

[20] T. G. Karen Kent, Suzanne Chevalier, H. Dang, Guide to integrating forensic techniques into incident response (August 2006). URL https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-86.pdf

[21] R. Hay, Exploiting qualcomm edl programmers (2): Storage-based attacks rooting (January 2018). URL https://alephsecurity.com/2018/01/22/qualcomm-edl-2/

[22] B. Kerler (April 2021). [link]. URL https://github.com/bkerler/Loaders/tree/a59f9f765ee1d3fae83eea814a917d4b8d8d8cd9/blackphone

[23] Western Digital, Replay protected memory block (rpmb) - protocol vulnerabilities, White paper (2020).

[24] S. Skorobogatov, The bumpy road towards iphone 5c nand mirroring (2016). arXiv:1609.04327.

[25] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, et al., ftpm: A firmware-based tpm 2.0 implementation, Microsoft Research (2015) 0–23.