When *Samsung* meets *MediaTek*: the story of a small bug chain

Maxime Rossi Bellom, Raphael Neveu, and Gabrielle Viala
mrossibellom@quarkslab.com - rneveu@quarkslab.com
gviala@quarkslab.com

Quarkslab

Abstract. Last year, we saw a resurgence of vulnerabilities impacting the logo parsers of various boot chains, leading to complete secure boot bypasses. While these researches, such as LogoFail [7], impacted mainly desktop environments, mobile platforms are not immune to this type of issue. During our past research analyzing the Android Data Encryption Scheme, we dived into the boot chain of Samsung low-end mobile devices, which are based on MediaTek System-on-Chips. Some parts of the implementation, including the JPEG logo parsing of the bootloader, quickly raised our interest as they had a good potential for bugs.

In this paper, we present a small bug chain that can be used by an attacker with physical access to the device to bypass the secure boot, execute code on the chip, reach persistency, and ultimately leak the secret keys protected by the hardware-backed keystore.

This article brings together two important concepts of modern mobile architecture: the secure boot and the Trusted Execution Environment. It gives a comprehensive view of how these features work and how they can be targeted by security researchers, focusing on the offensive approach.

1 Introduction

During our previous researches, we have studied the boot chain of some Samsung devices based on MediaTek system on chips. Our objective was to exploit a known boot ROM vulnerability to bypass the secure boot and ultimately retrieve the required ingredients to bruteforce the user credentials [10]. Once we became familiar with this boot chain, we decided to take a closer look at a component coming later in the process: the Little Kernel bootloader (LK, also called BL3-3).

We begin our bug-hunting journey in LK from a JPEG parser that was introduced by the vendor. Then we show how, thanks to reverse engineering, we discovered a vulnerability leading to code execution in the context of the bootloader, and how it can be used to bypass the secure boot and take full control over the Android system.

In order to trigger this vulnerability, we need a way to flash our JPEGs on the flash memory of the device. We will dive into the implementation of Odin, the Samsung recovery protocol and present a second vulnerability we discovered, allowing us to write anything on the flash memory without authentication.

In the last part, we focus on the ARM Trusted Firmware (also known as the Secure Monitor), which runs with the highest privileges on the device. We present two critical vulnerabilities we discovered and show how they allowed us to break the last security barrier of this device to leak the secrets hidden in the Secure World.

1.1 The MediaTek boot chain

In MediaTek boot chain, Little Kernel is the third bootloader, coming after the boot ROM and the preloader (as shown in Figure 1). It is executed in the Normal World with the Exception Level EL1 (more details about the Exception Levels can be found in section 3.1), which is the same as the Android kernel. This means that a vulnerability in this bootloader may lead to full control of the Normal World and so of the Android system. However, this can't impact the Secure World and its secrets which is why we targeted this second component: the ARM Trusted Firmware.

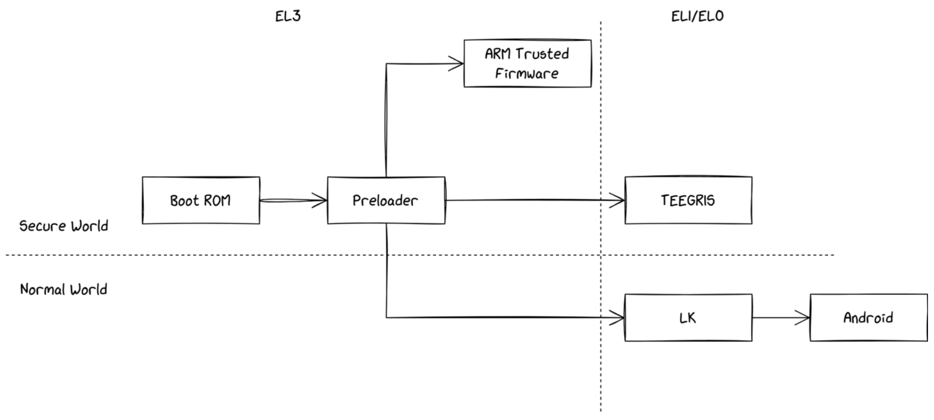
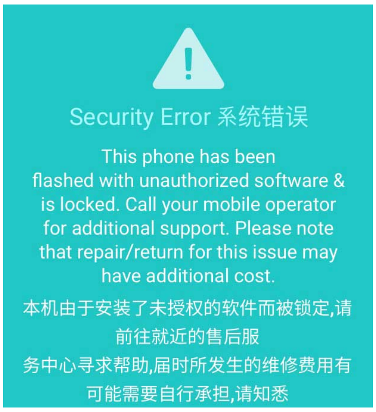


Fig. 1. The MediaTek Secure Boot

2 Little Kernel

Little Kernel[1] is an open-source operating system commonly used in the Android world, which primarily serves as a bootloader. It usually implements the fastboot protocol, which is used to perform diagnostic and recovery operations over USB (such as reflashing the partitions). More importantly, LK also implements *Android Verified Boot* [9], which is part of the secure boot and verifies Android-related partitions.

The implementation we found in the devices we studied differs from the open source one. Indeed, we noticed that a few modifications were introduced by both MediaTek and by Samsung. For instance, Samsung added its own diagnostic and recovery protocol: Odin, in place of fastboot. Another change introduced by Samsung is the ability to show pictures on the screen representing logos and error messages in JPEG format.



Security Error 系统错误

This phone has been
flashed with unauthorized software &
is locked. Call your mobile operator
for additional support. Please note
that repair/return for this issue may
have additional cost.

本机由于安装了未授权的软件而被锁定,请
前往就近的售后服
务中心寻求帮助,届时所发生的维修费用有
可能需要自行承担,请知悉

Fig. 2. Example of picture present in the `up_param` partition

It is also interesting to note the absence of mitigations in this bootloader: there is no ASLR, no bound checks in the heap and the heap is executable…

2.1 JPEG Loading During the Boot Process

These JPEG files are present in a partition called `up_param`, which consists of a tar file including the various images to be rendered on the screen depending on the boot state (e.g., secure boot verification failed) and the bootloader mode (e.g., locked/unlocked, Odin mode).

[1] `https://github.com/littlekernel/lk`

Even though this partition contains a signature, it is not verified at boot time. Which means that one with the ability to write something on the flash storage, through a vulnerability in the recovery protocols or with enough privileges on the Android system, can replace these pictures. From an attacker's point of view, this makes an interesting attack surface as it is possible to target the tar and the JPEG parsers for vulnerability research.

The Heap Overflow We started our analysis by reverse engineering the code responsible for providing the JPEG files to the parser after reading them from the flash memory. During this step, we discovered the first heap overflow vulnerability.

Listing 1: Code snippet of the function `drawimg` extracted from Ghidra

```
_JPEG_BUF = alloc(0x100000);
if (_JPEG_BUF == 0) {
  log("%s: img buf alloc fail\n","drawimg");
  uVar2 = 0xffffffff;
}
else {
  memset(_JPEG_BUF,0,0x100000);
  iVar1 = read_jpeg_file(file_name,_JPEG_BUF,0);
  if (iVar1 == 0) {
    log("%s: read %s from up_param as 0
↪  size\n","drawimg",file_name);
    uVar2 = 0xffffffff;
  }
// ...

pimage(*(undefined4 *)(&DAT_4c5107fc + param_1 * 0x3c),
       *(undefined4 *)(&DAT_4c510800 + param_1 *
↪  0x3c),0x2d0,0x640,1,_JPEG_BUF,iVar1);
```

Here, the buffer in which the JPEGs are copied is dynamically allocated with a fixed size of `0x100000` and is initialized to `0` with `memset`. It is then passed in argument to the function we called `read_jpeg_file`. This function takes the name of the JPEG to read as first argument, then reads it and stores its content into the buffer, previously allocated and provided as argument. The last argument corresponds to the maximum number of bytes to be read. However, the function behaves in a rather surprising way when this argument is set to `0` (which is the case here): no checks are performed on the size of the JPEG file. As a result, it is possible to overflow the JPEG buffer by placing a file bigger than `0x100000` bytes.

What can be done using this overflow depends on the allocator and on the state of the heap. The dynamic allocation algorithm used here seems to be an old version of *miniheap*,[2] that is relying on a doubly linked list.

Listing 2: Structure of the *miniheap* doubly linked list of free chunks

```
struct free_heap_chunk {
    struct list_node *prev;
    struct list_node *next;
    size_t len;
}
```

When a chunk is allocated, the free chunk is turned into an allocated one with a header (defined in Listing 3) followed by the data.

Listing 3: Structure of the *miniheap* allocated chunk header

```
struct alloc_struct_header {
    unsigned int magic; // Always 0x48454150
    void *ptr;          // Points to the header
    size_t size;        // Size of the data + header
}
```

The memory chunks are following each other, and an overflow happening in one of them may overwrite the metadata of the next chunk in memory (either free or allocated). What we can do from there depends on the state of the heap when the overflow happens. For example, if the chunk that overflows is followed by another one that contains function pointers, it is possible to simply overwrite these pointers to achieve code execution when they are used. However, in our case, it seems that nothing particularly interesting is coming after our JPEG chunk.

We found a method to take advantage of the next allocations that are performed in the execution of the JPEG parser. Indeed, several other structures are going to be allocated as part of the parser implementation. It is possible to use this overflow to overwrite the `prev` and `next` pointers of the free chunk coming just after our JPEG data. Whenever this free chunk is being allocated, the allocation function removes the free chunk from the free chunk list by placing the address of the `prev` chunk in the

[2] `https://github.com/littlekernel/lk/blob/master/lib/heap/miniheap/miniheap.c`

`next` one, and sets the address of the `next` one in the `next` field of the `prev` one (see Listing 4).

Listing 4: Removing a free chunk named node from the list

```
node->next->prev = node->prev;
node->prev->next = node->next;
node->prev = node->next = 0;
```

This way, we can turn a heap overflow into an arbitrary write with the constraint that both addresses must be writable. To turn this into a code execution, we put in `next` the address of the stack where the return address of the allocation function is, and we put in `prev` the address of our buffer where we placed our shellcode (See Fig 3, step 2). As the heap is executable, when the allocation function returns, our shellcode will be executed (See Fig 3, step 3).

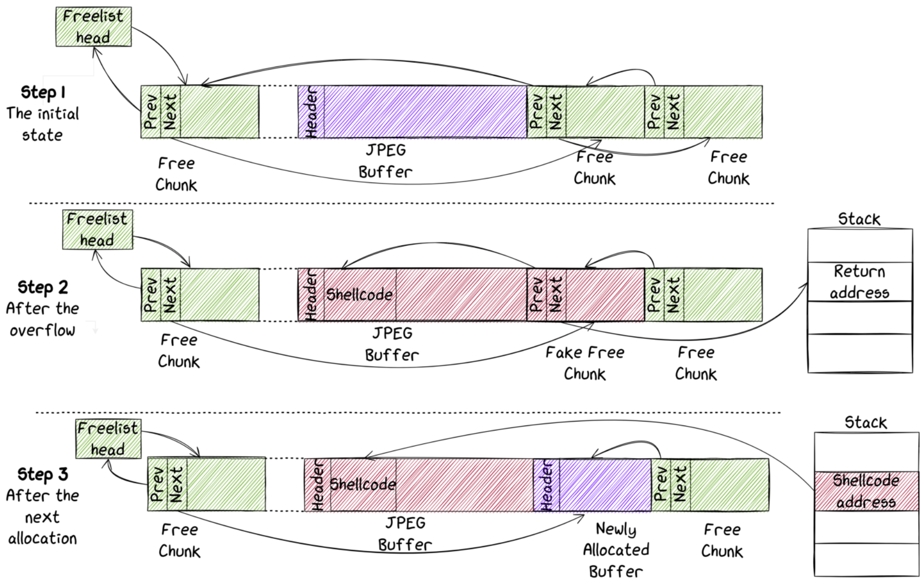


Fig. 3. The three states of the heap during the exploit

Of course there are many other little details that are important to make the exploitation succeed, here is a non-exhaustive list:

— the size of the free chunk must be exactly the same as the chunk being allocated, otherwise the algorithm will try to create another free chunk for the remaining size;

- the state of the heap after exploitation is broken and must be restored post exploitation for allocations to work again;
- the stack address is always the same (no ASLR), so we simply need to leak it once;
- the behavior is different whenever the next free chunk is followed by an allocated one in memory.

Debugging the Exploit: Emulation of the Parser It can be very helpful to have access to some sort of debugging capabilities to implement our exploit. For that, we implemented an emulator with Unicorn [2] which is the notorious CPU emulator framework based on Qemu and is quite straightforward and easy to use.

To build our emulator script we need to reverse engineer the bootloader and find:

- the base address where the code should be mapped;
- the entry point of the vulnerable function;
- the input format.

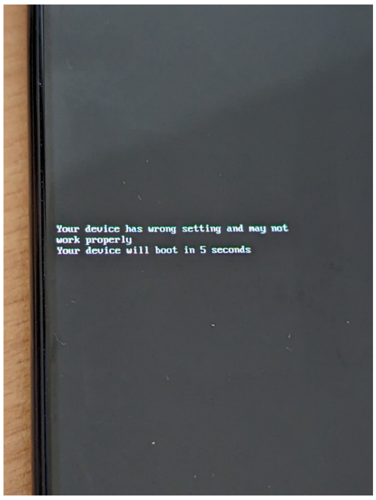
While writing an emulator with Unicorn can be very fast, we can note some limitations compared to using Qemu directly. For instance, there is no support for interrupts, signifying that we cannot emulate the full Operating System, but have to isolate a set of functions we want to target. On the same note, Unicorn is not designed to emulate hardware components. When building our emulator, we will have to hook every function or instructions that eventually perform one of these operations and deal with them through the hook handler, which can have a high impact on the execution speed.

In this context, we have to hook the functions used to log information and the ones used to read or write data on the flash memory, like the JPEG file.

As explained before, the state of the heap is really important for the exploitation. To make sure the heap of our emulator has the same state as the one in the device, we decided to dump its content from the device's memory, just before the execution of the code we want to emulate. We achieved this by patching the bootloader: we implemented our own function to write the content of the heap in a specific partition (we choose the partition named `SPU` which seems to be unused) and then we patched the part of the code loading the JPEG to call our dump function. To bypass the secure boot and run our modified bootloader, we used MTKClient[3] which exploits a vulnerability present in the boot ROM. Thanks to this,

[3] `https://github.com/bkerler/mtkclient`

we now have an emulator in which the heap will behave exactly the same as in the device and we are able to produce an exploit that will also work on both environments.



```
Your device has wrong setting and may not
work properly
Your device will boot in 5 seconds
```

Fig. 4. Message shown on the screen when the PoC works

To summarize, the vulnerability we discovered can lead to code execution in LK and the exploit will be persistent even after factory reset. This vulnerability impacts a wide range of Samsung devices: among them we identified A22, A32, A14, A34, A15 but there are probably others. We can also note that this vulnerability is also present in newer Samsung devices, such as the A15, which rely on the new generation of MediaTek SoCs where the infamous boot ROM vulnerability exploited by MTKClient has been fixed. Once our vulnerability is exploited, we have full control over the Normal World Execution Levels 1 and 0. This means that we can control the Android system and apply any modifications we want. Yet, to trigger our vulnerability, we need a way to write the `up_param` partition on the flash memory.

2.2 Odin Authentication Bypass

Odin is a protocol, on top of USB, that allows to write images on the flash storage of the device. Past research [5] have shown that it is an interesting target for vulnerability research in order to aim at the secure boot. It is implemented in BL3-3 (Little Kernel in our case) and can be started by booting the device in *Download Mode*. To do so, the user can maintain *Volume-up* and *Volume-down* buttons pressed while plugging the USB cable on an offline device.

Most images that can be flashed through Odin have to be authenticated. A footer, starting with the magic `SignerVer02` contains the signature and is present at the end of the image.

In Little Kernel, Odin is started as two threads: the first one, `Odin` implements the protocols, and the second one, `Odin_write` is used to write the file content in the flash memory. When a file is received, the first thread will wake up the other one by triggering an event. In order to write a partition in the flash storage, `Odin_write` will call the function `nand_write`.[4] This function will first look into the *Partition Information Table* (`pit`) for the partition name. This table is used by Odin to describe the partitions. Among other things, it indicates the partition names, the sector offset where it starts, its size, and so on. The following extract is a human-readable representation (see listing 5) that can be shown by the tool heimdall[5], which is an open source client for Odin protocol.

Note that only the entries present in the `pit` table, and for which the field `Flash Filename` is not empty can be flashed through Odin.

For these partitions, the function `nand_write` will call `check_secure_download` to authenticate the image. This function will first call `LookupAuthInfo` in order to retrieve from a global array, a structure describing how the image should be authenticated. This array is statically allocated and whenever a partition is not found by `LookupAuthInfo`, the system considers that no authentication is required, and `check_secure_download` will simply return without any error. As a result, `nand_write` will proceed and write the image in the flash memory.

We listed all the partitions present in the `pit` table but not in the global array: `md5hdr`, `md_udc`, `pgpt`, `sgpt`, `steady`, and `vbmeta_vendor`. Two partitions are particularly interesting: `pgpt` and `sgpt`. Indeed they point to the *GUID Partition Table* (GPT) headers of the flash memory. The GPT table is similar to the `pit` table: it describes all the partitions, indicating names, sizes, starting offsets, and so on. There is a primary header at the beginning of the flash, and a secondary one at the end.

Thus, the main issue here is that we can change the GPT table with a physical access to the device and without authentication.

[4] All the thread and function names where retrieved from strings and symbols present in the firmware through reverse engineering

[5] `https://github.com/Benjamin-Dobell/Heimdall`

Listing 5: Extract of the command `heimdall print-pit`

```
--- Entry #0 ---
Binary Type: 0 (AP)
Device Type: 2 (MMC)
Identifier: 80
Attributes: 2 (STL Read-Only)
Update Attributes: 1 (FOTA)
Partition Block Size/Offset: 0
Partition Block Count: 8192
File Offset (Obsolete): 0
File Size (Obsolete): 0
Partition Name: bootloader
Flash Filename: preloader.img
FOTA Filename:

--- Entry #1 ---
Binary Type: 0 (AP)
Device Type: 2 (MMC)
Identifier: 70
Attributes: 5 (Read/Write)
Update Attributes: 1 (FOTA)
Partition Block Size/Offset: 0
Partition Block Count: 34
File Offset (Obsolete): 0
File Size (Obsolete): 0
Partition Name: pgpt
Flash Filename: pgpt.img
FOTA Filename:

--- Entry #2 ---
Binary Type: 0 (AP)
Device Type: 2 (MMC)
Identifier: 71
Attributes: 5 (Read/Write)
Update Attributes: 1 (FOTA)
Partition Block Size/Offset: 34
Partition Block Count: 32
File Offset (Obsolete): 0
File Size (Obsolete): 0
Partition Name: pit
Flash Filename:
FOTA Filename:
...
```

The GPT table is used by most of the components of the firmware when reading or writing a partition. The `pit` table seems to be used mainly for features such as Odin or to show logos on the screen from the `up_param` partition. For the later one, it means that even if we change the partition `up_param` in the GPT table, Little Kernel will still read it from the same place, using information from the `pit` table.

Exploiting the vulnerability Yet, it is possible to leverage this vulnerability to flash all the partitions without authentication, including `up_param`. First of all, there is an official way to flash the `pit` table through Odin. The image containing the `pit` is signed and it requires authentication when being flashed. Nevertheless, there is no signature verification when the `pit` is read. By default, it is present at a fixed offset in the flash memory (`0x4400` in the case of the A225F). Even so, the function responsible for reading it will first look for a partition called `pit` in the GPT table. If there is one (and there is no such partition by default), it will be read from there (see listing 6).

Listing 6: Code snippet of the function `read_pit` extracted from Ghidra

```
uVar3 = 0x4400;
iVar1 = get_part_table("pit");
if (iVar1 == 0) {
    uVar3 = get_partition_offset("pit");
}
uVar2 = storage(3);
iVar1 =
↪  storage_read(uVar2,0x4000,(int)uVar3,(int)((ulonglong)uVar3 >>
↪  0x20),&ODIN_TEMP_BUF,0x4000);
```

Therefore we should be able to change the data being read by creating a `pit` partition in the GPT table. To achieve that, we can use the following steps:

1. First, we flash a new `pit` table in the partition `vbmeta_vendor`, which does not require authentication. In this table, we rename the partition `md5hdr` to `up_param` and do the other way around for `up_param`;
2. Then, we flash a modified version of the partition `up_param` in `md5hdr` where we place different JPEG files ;
3. Finally, we flash a new `pgpt` partition containing the GPT table where we simply renamed the partition `vbmeta_vendor` to `pit`.

Our proof-of-concept consists of three shell commands using heimdall as a client (listing 7).

Listing 7: The Odin exploit in three commands

```
$ heimdall flash --vbmeta_vendor file-patched.pit
...
$ heimdall flash --md5hdr up_param-patched.bin
...
$ heimdall flash --pgpt gpt/gpt-patched-pit.bin
...
```

As a result, we can see that the new `up_param` partition is used since our JPEG is shown on the screen (see figure 5).

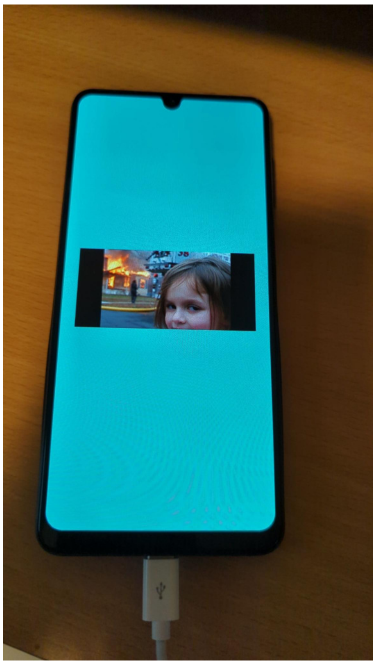


Fig. 5. Modified JPEG shown on screen which proves that the exploit worked

To sum up, with this vulnerability we can bypass the authentication in Odin, meaning that we can flash anything anywhere on the flash storage, with a physical access to the device as the only requirement. It seems to impact all the Samsung devices based on MediaTek SoCs. Combined with the previous vulnerability in the JPEG loader, we can bypass the secure boot and fully control the Normal World privilege level EL1, including the Android system. We have shown in our previous research how it is possible to modify Little Kernel to bypass secure boot and modify the Android system to root it.

However, we are still not able to control the Secure World. Indeed, in the Android field the Secure World is used to deal with sensitive information such as secret keys or DRMs.

3 Targeting ARM Trusted Firmware

3.1 Introduction to the Secure Monitor

Arm Trusted Firmware (ATF) is the reference implementation of a Secure Monitor for the ARM A-Profile platform.
To understand the role of the Secure Monitor, one must understand how modern mobile devices use security features such as *TrustZone*.
Android runs conjointly with a *TEE* (Trusted Execution Environment) such as TEEGRIS in the case of Samsung phones. A TEE is guaranteed to be isolated from Android by leveraging the TrustZone feature from ARM processors.
ARM defines different *Exception Levels* (EL) ranging from 0 to 3 that can be either Secure or Non-Secure:

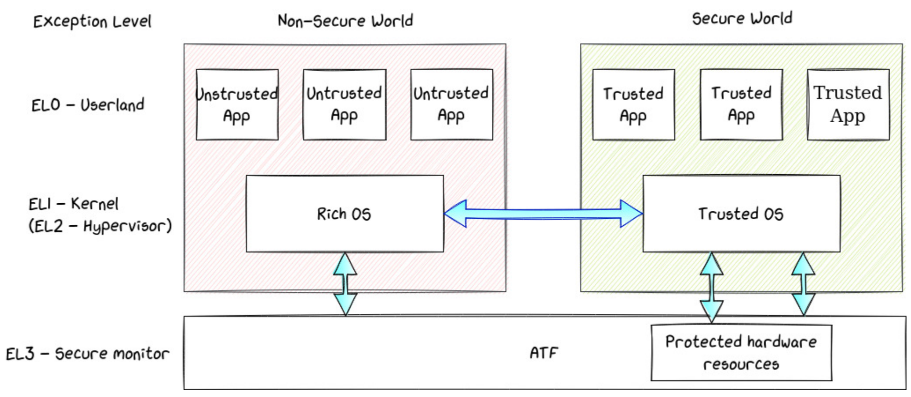


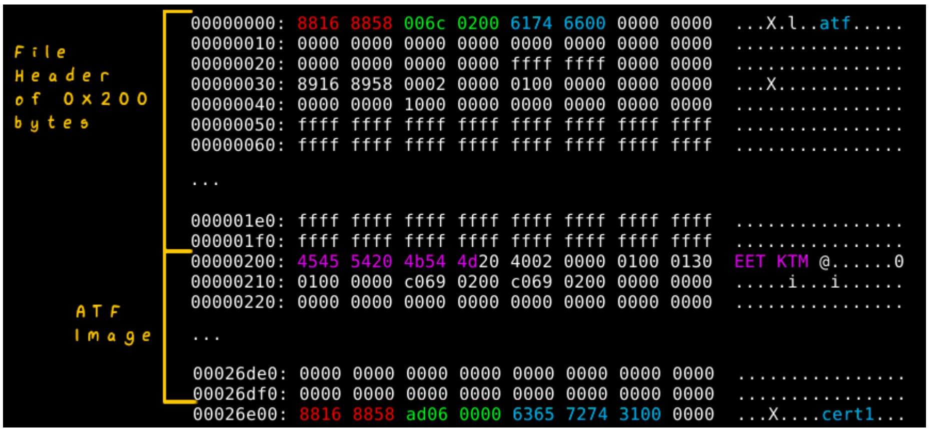
Fig. 6. Exception Levels with Secure and Normal Worlds

As seen on the schema above, the Secure Monitor is the most privileged piece of software running on Android mobile devices. It allows the Non-Secure World to communicate with the Secure World and vice-versa.

To communicate with the Secure World from the Normal World, the kernel running in NS.EL1 has to send Secure Monitor Calls (SMCs) to the Secure Monitor. The secure monitor uses the handler corresponding to the given SMC request, and may pass the data to the TEE running in S.EL1 if it is needed to process the request. The convention used regarding SMC parameters is defined in the SMC Calling Convention [3] documentation.

Exception Levels 0, 1 and 2 in both secure and Normal Worlds have their own virtual address space. The ARM Architecture Manual [4] describes several flags that are used in the page table entries to protect the memory pages, such as the regular *RWX* flags, but also the *NS* flag to distinguish between secure and Normal World. This prevents one context to manipulate memory of the other.
`EL3` is the most privileged Exception Level and can interact with both worlds.
And because of its privileges, it is able to map any physical address to its own virtual address space. This proves to be useful, as we show later on.

We need to retrieve the Secure Monitor binary in order to start reverse engineering it. It can be extracted from the Samsung ROM in the partition named `tee-verified.img`. Every file on this partition is preceded by a particular header:



```
00000000: 8816 8858 006c 0200 6174 6600 0000 0000  ...X.l..atf.....
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 ffff ffff 0000 0000  ................
00000030: 8916 8958 0002 0000 0100 0000 0000 0000  ...X............
00000040: 0000 0000 1000 0000 0000 0000 0000 0000  ................
00000050: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000060: ffff ffff ffff ffff ffff ffff ffff ffff  ................

...

000001e0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
000001f0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000200: 4545 5420 4b54 4d20 4002 0000 0100 0130  EET KTM @......0
00000210: 0100 0000 c069 0200 c069 0200 0000 0000  .....i...i......
00000220: 0000 0000 0000 0000 0000 0000 0000 0000  ................

...

00026de0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00026df0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00026e00: 8816 8858 ad06 0000 6365 7274 3100 0000  ...X....cert1...
```

Fig. 7. tee-verified partition architecture

A magic number of 4 bytes `0x88168858` (in red) indicates a new entry. The size of the entry of 4 bytes `0x26c00` is right next to it (in green) in little endian. Then we have the name of the entry (in blue).
As you can see, we are lucky as **atf** is the name of the first entry, which is

exactly what we are looking for. We can see that its content can be easily identified with the string **EET KTM** (in pink).
Once extracted, we can start doing static analysis.

3.2 Static Analysis versus Dynamic Analysis

As said earlier in this paper, ATF is the reference implementation provided by ARM. Even though Samsung and MediaTek modified it to their needs, most of the code remains the same and can be studied to provide a better insight at how things work.[6]

While analyzing the code, we noticed a few debug messages that allowed us to put a name on the unknown functions and have a rough idea of what the code is doing here and there.
To communicate with ATF, we must be able to send SMCs to it, but the SMC instruction is only available in EL1 (kernel mode).
Assuming we have root access on the device, we can simply create a kernel module that exposes a device file accessible from userland to forward the parameters to the ATF for the sake of convenience.
Once the kernel module is ready and successfully loaded, we can interact with ATF and try to debug it dynamically.

At this stage, it would be tempting to try fuzzing the SMC handlers to automate the vulnerability research. But because the ATF is the most privileged software running on the device, messing with it often means crashing the whole phone, which makes the process very tedious. So we would have to emulate the ATF entirely using Unicorn for example. However, while analyzing the code, we noticed that many of the handlers are closely interacting with the hardware and concluded that the few remaining candidates would not be very interesting to fuzz. So we resorted to pure static analysis instead.

3.3 The Vulnerabilities

The handlers for the different SMCs are defined in the `mediatek_plat_sip_handler_kernel` function. Two of them, SMC `0xc2000526` and `0x82000526` share the same handler that looks interesting (see Listing 8).

The first argument `arg1` is not checked and used to read a value at the address `arg1 * 4 + 0x4ce2f578`. This means that if we pass `(arbitrary_address - 0x4ce2f578) / 4` to our kernel module as the first argument, we can read an arbitrary address.

[6] `https://github.com/ARM-software/arm-trusted-firmware`

Listing 8: Code snippet of the handler for SMC `0x82000526` extracted from Ghidra

```
[...]
  smcid = 0x82000526;
  if (smc_id == smcid) {
      out_value = (ulong)*(uint *)(arg1 * 4 + 0x4ce2f578);
      goto exit;
  }
[...]
exit:
      param_7[2] = out_value;
      param_7[1] = arg1;
      goto LAB_4ce0c2c8;
[...]
LAB_4ce0c2c8:
      *param_7 = 0;
      return param_7;
```

However, we saw previously that the Secure Monitor also uses a virtual address space. This means that we can't just use a physical address to read. Or can we? Looking at the other SMC handlers, we saw that some eventually call what looks like an `mmap` function. This is the case for SMC number **`0x8200022a`** that calls a function that we named `spm_load_pcm_firmware`. This function also calls a wrapper to our `mmap` function (see Listings 9 and 10).

Luckily for us, this `mmap_wrap` function maps a physical address to the very same virtual address. This means we can mmap any memory region up to a size of `0x100000` and then read what's inside it using our previous leak.

Listing 9: Code snippet of SMC `0x8200022a` that leads to an arbitrary `mmap`

```
[...]
  if (smc_id == 0x8200022a) {
LAB_4ce0c208:
    spm_load_pcm_firmware(arg1,arg2,arg3);
    goto LAB_4ce0c2f0;
  }
[...]
```

Listing 10: Code snippet of function `spm_load_pcm_firmware`

```
undefined * spm_load_pcm_firmware(ulong param_1,undefined
↪  *addr,ulong size) {
  switch(param_1 & 0xffffffff) {
  case 0:
  [...]
    break;
  case 1:
    if (size < 0x100001) {
      mmap_wrap((ulong)addr,size);
  [...]
  }
```

Using these two vulnerabilities, we can `mmap` a memory region using its physical address and then leak it.
After several tests, it appears that we are limited to 8Mo of mapped memory, which corresponds to calling 8 times the `mmap` function. Calling it more than 8 times makes the device crash. It is likely that we exceed the maximum number of memory pages allowed for ATF.

4 Leaking secret keys

4.1 Android Keystore and Hardware-backed Keys

The Android Keystore [8] provides to applications a safe way to store and use cryptographic keys. These keys can rely on several security modules providing different levels of security which are:

— The *Trusted Environment*, which is used whenever the keys rely on the TEE (e.g., TrustZone with ARM architectures);
— *Strongbox* that is used when the keys rely on a security chip, such as the Titan M by Google [6];
— *Software*, the least secure one, which is used when no hardware protection is implemented.

TEEGRIS [1] has its own Keystore backend application (called *Keymaster*), which is used to store most of the keys.

As there is usually only a little persistent memory available in the secure hardware environment, the keys are stored in the Android data partition as encrypted keyblobs. Indeed, for every Keystore keys (called *Key Material* in Figure 8), the secure hardware (either *Strongbox* or *Trusted Environment*), derives a cryptographic key called a key encryption key, from its own internal secrets. The key encryption key is then used to encrypt the Key Material and generate the encrypted keyblob.

The careful reader may have noticed that an attacker, with enough privileges on the Android system, is able to use any of these keyblobs and ask the keystore to perform any operations with it. However, it should be possible to access these keys decrypted, with the capability to leak Secure World's memory.

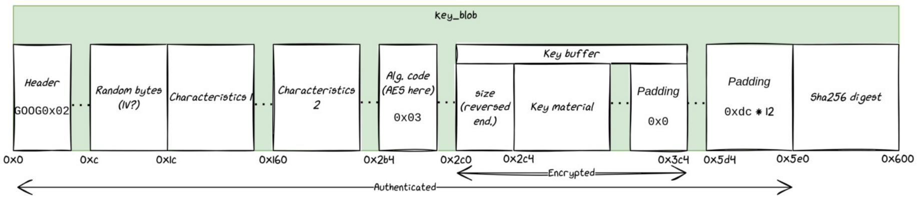


Fig. 8. Structure of the Keyblob

As with many other cryptographic mechanisms, the keystore internally uses *begin*, *update*, and *final* operations to carry its cryptographic operations. This is illustrated below:

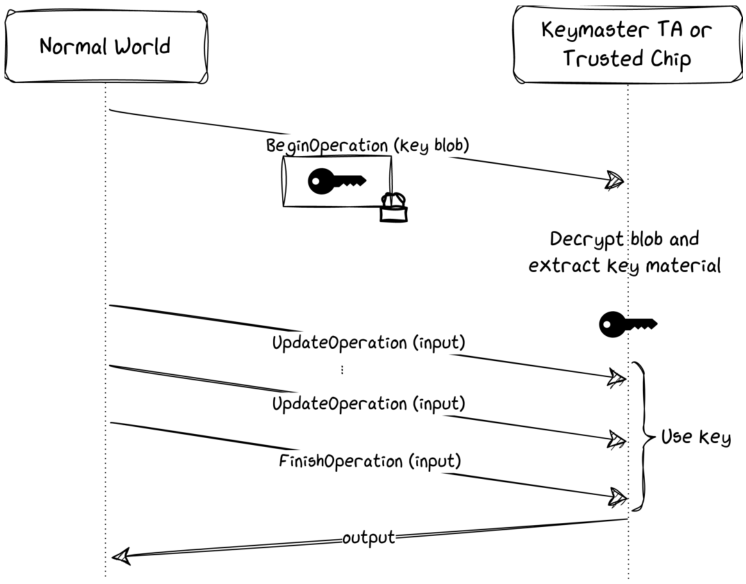


Fig. 9. Cryptographic operations used by the Keystore

While we won't delve too much into the details of these operations, it is important to understand that the *key* is loaded and decrypted in the Secure World RAM during the *begin* operation.

4.2 Leaking the keys

Now that we have a better understanding of the Keymaster *Trusted Application* (TA), we can try to leak a key that is being used in memory using our ATF out-of-bound memory read.
The plan is as follows:

- Locate the Keymaster TA in memory
- Start a cryptographic operation by triggering the `beginOperation()` function on the Keymaster side to load the key in memory
- Leak the TA memory using our ATF bug to recover the key
- Resume the cryptographic operation

To locate the Keymaster TA, we can use the logs of Little Kernel stored in the `/proc/last_kmsg` file:

Listing 11: Extract of Little Kernel logs in /proc/last_kmsg

```
a22:/ # cat /proc/last_kmsg
[...]
[4425] mblock_reserve-R[5].start: 0x4ce00000, size: 0x60000 map:0
↪  name:atf-reserved
[4425] mblock_reserve-R[6].start: 0xbff70000, size: 0x80000 map:0
↪  name:atf-ramdump-reserved
[4425] mblock_reserve-R[7].start: 0xbff00000, size: 0x40000 map:0
↪  name:atf-log-reserved
[4426] mblock_reserve-R[8].start: 0x7ac00000, size: 0x400000 map:0
↪  name:tee-secmem
[4426] mblock_reserve-R[9].start: 0x7f300000, size: 0xc0000 map:0
↪  name:SSPM-reserved
[4426] mblock_reserve-R[10].start: 0x7b200000, size: 0x4000000
↪  map:0 name:tee-reserved
[...]
[4436] lk finished --> jump to linux kernel 64Bit
```

These are logs from the previous bootloaders (including the preloader, the second bootloader, and Little Kernel), before the start of the Linux kernel. Interestingly, we can see in the logs the various memory regions allocated for different components such as the Secure Monitor (`atf-reserved` in the logs) and the TEE OS (`tee-reserved`). These addresses are physical addresses, which is good because we can `mmap`

physical addresses using our bug as shown previously. The memory block of interest here is *tee-reserved* starting at **0x7b200000**.
To determine where the Keymaster TA exactly is in this memory block, we can simply dump the whole *tee-reserved* memory block and search for the TA content.
The Keymaster TA can be found in `/vendor/tee/00000000-0000-0000-0000-4b45594d5354`. There are a few strings for example that we can find both in the binary and in the memory.
After a few dumps of RAM, we find the content of the TA around the address `0x7c200000`. The content doesn't seem to change much between restarts of the device.

Now that we know where the key is susceptible to be, we have to make sure that it is loaded in RAM while we dump it. To do so, we will start a cryptographic operation triggering a `beginOperation()` by the Keymaster TA. We have options here, we could for example:

1. Create an app that uses the Keystore API to encrypt a message
2. Use the `keystore_cli_v2` command-line tool on the Android system to encrypt a file

To speed up the implementation of our PoC, we implemented a dummy application (see Listing 12) that first imports an hardcoded key into the Keystore and then encrypts a message with it.

Listing 12: Hardcoded Key in our Dummy Application

```
byte[] key = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA".getBytes();
SecretKey yourKey = (SecretKey) new SecretKeySpec(key, 0,
↪  key.length, "AES"); ;
[...]
keyStore = KeyStore.getInstance("AndroidKeystore");
keyStore.load(null);
keyStore.setEntry(
  "key1",
  new KeyStore.SecretKeyEntry(yourKey),
  new KeyProtection.Builder(KeyProperties.PURPOSE_ENCRYPT |
  ↪  KeyProperties.PURPOSE_DECRYPT)
    .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
    .build());

keyStoreKey = (SecretKey) keyStore.getKey("key1", null);
```

While a bit unrealistic, spotting the key in memory will be easier for us this way.

Research [11] has been done on Keymaster and presents the different bricks involved by the Android Keystore API. Behind the scenes, calling the Keystore API will eventually reach the *Keymaster HAL* which is implemented by a service called **android.hardware.keymaster@4.0-service** that is in charge of communicating with the other daemons to, at last, reach the trustzone driver and forward messages to the Secure World. We can hook the `beginOperation()` function of the HAL service and make it wait while we take the time to dump the Keymaster TA memory. For this purpose, we used a simple Frida script that blocks the execution while we dump the memory.
This whole process can be better visualized with the Fig 10.

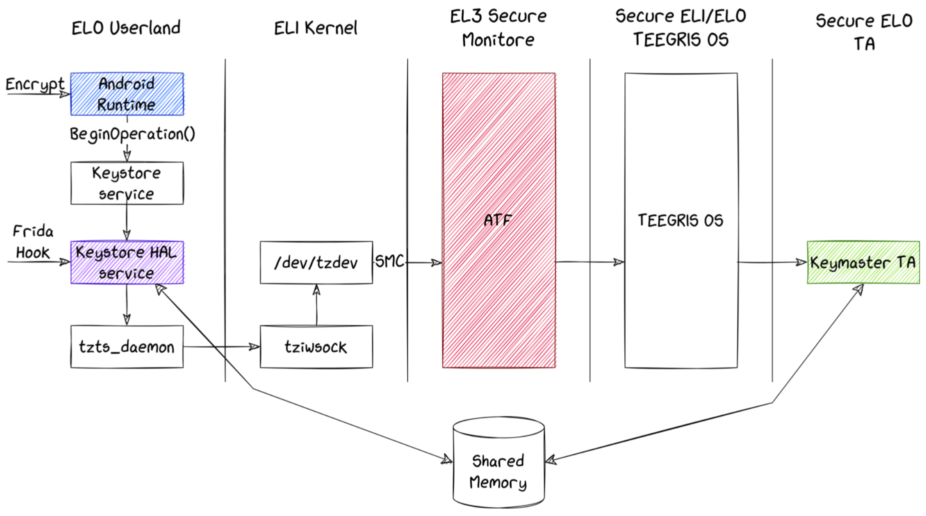


Fig. 10. Hooking the execution flow after a call to `beginOperation()` using Frida

Then we run our exploit to dump the raw memory from the Keymaster TA (see Listing 13).

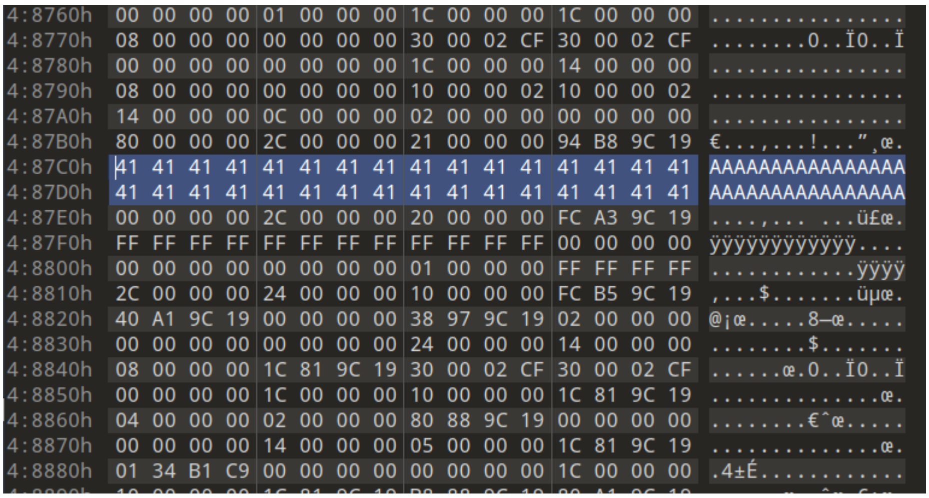
From there, is it quite easy to spot our key (See Fig 11).

This should be enough to show that our attack scenario can lead to leaking the Keystore keys. It is interesting to note that every time we repeat this test we will find the key placed in another address which is due to the allocation algorithm and its state when we run the attack.

Of course, in a real-world scenario, an attacker must find a way to identify the secret key without requiring prior knowledge about its content.

Listing 13: Dumping the Keymaster TA Memory

```
for i in `seq 0x7c200000 0x100000 0x7c500000`
do
addr=`printf "%x" $i`
./exploit leak_mmap ${addr} 0x100000 > dump-${addr}.txt
done
```



```
4:8760h  00 00 00 00 01 00 00 00 1C 00 00 00 1C 00 00 00  ................
4:8770h  08 00 00 00 00 00 00 00 30 00 02 CF 30 00 02 CF  ........0..Ï0..Ï
4:8780h  00 00 00 00 00 00 00 00 1C 00 00 00 14 00 00 00  ................
4:8790h  08 00 00 00 00 00 00 00 10 00 00 02 10 00 00 02  ................
4:87A0h  14 00 00 00 0C 00 00 00 02 00 00 00 00 00 00 00  ................
4:87B0h  80 00 00 00 2C 00 00 00 21 00 00 00 94 B8 9C 19  €...,...!...”¸œ.
4:87C0h  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
4:87D0h  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
4:87E0h  00 00 00 00 2C 00 00 00 20 00 00 00 FC A3 9C 19  ....,... ...ü£œ.
4:87F0h  FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00 00  ÿÿÿÿÿÿÿÿÿÿÿÿ....
4:8800h  00 00 00 00 00 00 00 00 01 00 00 00 FF FF FF FF  ............ÿÿÿÿ
4:8810h  2C 00 00 00 24 00 00 00 10 00 00 00 FC B5 9C 19  ,...$.......üµœ.
4:8820h  40 A1 9C 19 00 00 00 00 38 97 9C 19 02 00 00 00  @¡œ.....8—œ.....
4:8830h  00 00 00 00 00 00 00 00 24 00 00 00 14 00 00 00  ........$.......
4:8840h  08 00 00 00 1C 81 9C 19 30 00 02 CF 30 00 02 CF  ......œ.0..Ï0..Ï
4:8850h  00 00 00 00 1C 00 00 00 10 00 00 00 1C 81 9C 19  ..............œ.
4:8860h  04 00 00 00 02 00 00 00 80 88 9C 19 00 00 00 00  ........€ˆœ.....
4:8870h  00 00 00 00 14 00 00 00 05 00 00 00 1C 81 9C 19  ..............œ.
4:8880h  01 34 B1 C9 00 00 00 00 00 00 00 00 1C 00 00 00  .4±É............
```

Fig. 11. View of our memory dump that contains the key

5 Conclusion

This article dived into the boot chain and into the trusted execution environment of Samsung mobile devices based on MediaTek System-on-Chip. We presented a critical 0-day we discovered that impacts the JPEG loader on many different low-end devices. Combined with the vulnerability discovered in Odin, it allows an attacker with a physical access to the device to bypass the secure boot and take control over the Android system with persistency. We also showed that two vulnerabilities impacting the ARM Trusted Firmware, leading to an arbitrary read of the full memory of the device, are enough to reach the remaining secrets such as the keys stored in the Android Keystore. All the vulnerabilities we discovered have been reported to the vendor.

References

1. Breaking tee security: Tees, trustzone and teegris. `https://www.riscure.com/tee-security-samsung-teegris-part-1/`.
2. Unicorn: The ultimate cpu emulator. `https://www.unicorn-engine.org/`.
3. ARM. Smc calling convention. `http://infocenter.arm.com/help/topic/com.arm.doc.den0028b/ARM_DEN0028B_SMC_Calling_Convention.pdf`, 2016.
4. ARM. Arm architecture reference manual for a-profile architecture. `https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/LearnTheArchitecture-MemoryManagement-101811_0100_00_en.pdf`, 2019.
5. Jeff Chao. Breaking samsung's root of trust: Exploiting samsung s10 s-boot. `https://i.blackhat.com/USA-20/Wednesday/us-20-Chao-Breaking-Samsungs-Root-Of-Trust-Exploiting-Samsung-Secure-Boot.pdf`, 2020.
6. Maxime Rossi Bellom Damiano Melotti. Attack on Titan M, Reloaded: Vulnerability Research on a Modern Security Chip. `https://www.blackhat.com/us-22/briefings/schedule/index.html#attack-on-titan-m-reloaded-vulnerability-research-on-a-modern-security-chip-27330`, 2022.
7. Yegor Vasilenko Alex Ermolov Sam Thomas Anton Ivanov Fabio Pagani, Alex Matrosov. Logofail: Security implications of image parsing during system boot. `https://i.blackhat.com/EU-23/Presentations/EU-23-Pagani-LogoFAIL-Security-Implications-of-Image_REV2.pdf`, 2023.
8. Google. Android keystore system. `https://developer.android.com/privacy-and-security/keystore`.
9. Google. Android verified boot 2.0. `https://android.googlesource.com/platform/external/avb/+/master/README.md`, 2023.
10. Damiano Melotti Maxime Rossi Bellom. Android Data Encryption in depth. `https://blog.quarkslab.com/android-data-encryption-in-depth.html`, 2023.
11. Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust dies in darkness: Shedding light on samsung's trustzone keymaster design. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.