2021: A Titan M Odyssey

Maxime Rossi Bellom, Damiano Melotti, and Philippe Teuwen
mrossibellom@quarkslab.com
dmelotti@quarkslab.com
pteuwen@quarkslab.com

Quarkslab

Abstract. In the past years, most of the Android devices were relying on ARM TrustZone for critical security features.
In 2018, with the release of the Pixel 3, Google introduced the Titan M chip, a hardware security module used to enhance the device security by reducing its attack surface, mitigating classes of hardware-level exploits such as Rowhammer or Spectre, and providing several security sensitive functions, such as a Keystore backend called StrongBox, Android Verified Boot (or AVB) and others. It has been now almost three years since this announcement and yet very little information about it is available online. In this whitepaper, we deep dive into the Titan M's internals and usages. Our goal is to give an understanding of its attack surface as well as its role in some critical security features such as the StrongBox/Keymaster. We provide some details on how we performed our research from the reverse engineering of the firmware to the physical sniffing of the communication and fuzz testing. We discovered some known and previously unknown vulnerabilities which, among others, allowed us to execute code on the chip and helped us to solve some of the remaining mysteries behind this chip.

1 Introduction

Smartphones represent one of the most complex scenarios for information security. Over the years, their computational power has increased to a point that they can no longer be clearly distinguished from computers. At the same time, they store valuable data and are used to perform security-sensitive actions that represent interesting targets for attackers.

Given such a broad threat model [21], and considering that the extremely large computing base of a modern OS cannot be fully trusted, vendors started to leverage hardware to improve the security of their systems. Concretely, we can distinguish three ways to implement secure hardware solutions [24].

– Virtual Processor: this is the most widely adopted approach and it consists in separating hardware resources within the same chip,

implementing a secure and non-secure world as execution modes of the main CPU. ARM TrustZone is certainly the most notable instance of this solution [8].
- On-SoC Processor: this solution is used by Apple in its Secure Enclave [7]. Instead of featuring one CPU that can run in two states, in this case there are two CPUs, a main one dedicated to non-sensitive operations and one for the secure state. These two isolated processors lie together on the same System-on-Chip.
- External Coprocessor: the last option features a physically separated and completely independent chip, handling only security-sensitive operations. The chip can communicate with the main CPU using various types of buses, runs its own firmware and has full access to hardware resources. This is the solution adopted by Google in its Titan M, which is also the first example of a dedicated chip in an Android device. Before, other devices only supported Secure Elements, more limited modules only for payments or other restricted use cases [11].

One of the main features brought by Titan M is attack surface reduction, improving the isolation level given by TrustZone. Like with other trusted chips, since the firmware is limited in terms of functionality (with a size orders of magnitude smaller than the one of a standard OS), the probability of mounting a software attack is significantly reduced. In addition to that, the physical isolation between the chip and the main SoC also mitigates classic hardware-level exploits such as Rowhammer [15], Spectre, and Meltdown [16, 19, 23]. The presence of dedicated Tamper Resistant Hardware (TRH) guarantees improved resistance against side-channel attacks, which are actually one of the factors influencing this design choice [17, 22].

When the chip was announced, Google reported that its firmware source code would be made public, allowing anyone to reproduce binary builds [23]. To date, no source code has been published and not much information is available about it. Despite that, to motivate researchers into investigating this module, the company introduced a special reward of one million dollars, for whoever can find a full-chain remote code execution exploit with persistence [18]. Indeed, Titan M represents the so-called *Root of Trust* of a device, the baseline all security features rely upon: in case of compromise, the target falls completely under the attacker's control.

This is the first extensive study on Titan M. We start by analyzing the architecture and internals of the chip, and reverse engineering the firmware. With the acquired knowledge, we explain the boot and update procedure, how the chip communicates with Android and how a specific

Android feature (*StrongBox*) is backed by Titan M. We present the tools we developed during the research (available at `https://github.com/quarkslab/titanm`) and how we used them to facilitate both static and dynamic analysis of the chip. After showing our approach also on the hardware side, we dive into some vulnerabilities and exploitation, which allowed us to build a better picture of how the chip works. Finally, we design and implement a fuzzer that, based on the grammar of the exchanged messages, automatically tests the chip with a black-box approach. Thanks to fuzzing, we can rediscover some vulnerabilities found statically, as well as some new ones that we reported to the vendor.

2 Architecture and Internals

In its first communication about Titan M [23], Google provided some information about the chip hardware. We learned that the chip is based on an ARM Cortex-M3 architecture with an internal flash memory and 64 KB of RAM. Since the RAM is very small, the code is executed directly from the flash memory. We also learned that the chip contains several hardware accelerators for common cryptography algorithms (such as AES, SHA and public key algorithms), and a True Random Number Generator. On top of this, the chip includes a set of hardware defenses protecting against advanced hardware attacks.

2.1 Firmware Introduction

The firmware can be found in the filesystem of a Google Pixel 3 smartphone, at the path `/vendor/firmware/citadel`. It is a raw binary file, not encrypted nor obfuscated. We follow two parallel approaches to study it: on the one hand, we focus on pure static reverse engineering, on the other hand, we gather additional knowledge by reading the source code of the Android components responsible for the communication with the chip.

On the AOSP, the main source of information is the folder `platform/external/nos/host`, where we can find some header files containing relevant information about the module.[1] *Nos* is probably an abbreviation for *Nugget OS*, which might be a code name for chip's operating system.

The repository contains the source code of the `citadel_updater` tool, as well. The compiled utility, located at `/vendor/bin/hw` on the device, can be used in an `adb` root shell, to get the running version of the firmware

[1] `https://android.googlesource.com/platform/external/nos/`

and some statistics, update the chip, and perform other actions on its current state. `citadel_updater` also allows to retrieve a snapshot of the firmware dependencies: among the third party ones, for example, we can find `nanopb`, the library used to implement the communication protocol with Android (cf. Section 2.5).

The memory layout of the firmware is reported in a header file in the AOSP.[2] In total, there are four images, two RO and two RW. Despite these names suggesting the permissions of the regions (Read-Only and Read-Write), both can be overwritten during an update. Each image is duplicated to support A/B updates, ensuring that a valid image is always present on the device during an update [9].

The firmware is based on Chromium Embedded Controller (EC), an open-source microcontroller OS developed by Google.[3] This represents another useful source of information for reverse engineering: many functions are very similar and they can be easily matched thanks to the presence of debugging statements with the same strings.

EC is a lightweight OS written in C. It is built around the concept of *tasks*, which can be defined as independent execution units with a fixed pre-allocated stack. The firmware does not use dynamic allocation, thus all the memory required by a task must be explicitly defined at compile time.

During execution, the chip is interrupt-driven and can run in two execution modes: *handler* (privileged) and *thread* (privileged and non-privileged, depending on the configuration). Tasks run in thread mode; to change execution context, a *software interrupt* is raised (using the `svc` instruction) and processed by the scheduler, which instead runs in handler mode [1].

The latest version of the firmware contains nine tasks: some of them are proper Trusted Applications (TA), while some others work in support of the OS.

- `<< idle >>`: executed when no other tasks are;
- `HOOKS`: managing events and timers;
- `NUGGET`: responsible for OS control, implementing the required logic for password checks, firmware updates, and other system-related commands;
- `FACEAUTH`: the TA providing hardware-backed support for biometric authentication;

[2] `https://android.googlesource.com/platform/external/nos/host/generic/+/refs/heads/android11-release/nugget/include/flash_layout.h`

[3] `https://chromium.googlesource.com/chromiumos/platform/ec/`

- `AVB`: the Android Verified Boot TA;
- `KEYMASTER`: the Keymaster TA, corresponding to the *StrongBox* API in the Android Keystore [13];
- `IDENTITY`: the Identity TA, to securely store identity documents;
- `WEAVER`: the Weaver TA, which allows verification of user lock screen factor with hardware support (the equivalent of *Gatekeeper* [12]);
- `CONSOLE`: managing a simple console accessible from the chip's UART interface.

In EC, the list of tasks can be found in the `ec.tasklist` file. In the Titan M firmware, instead, we can find a data structure in memory storing for each task the value of the `r0` register, a pointer to the main routine of the task, and the associated stack size.

2.2 Boot Process

At boot, the chip first executes a small piece of software that is not present in the firmware raw file, called Boot ROM. The Boot ROM then verifies and launches one of the RO images, also called the loader. The loader verifies and launches one of the RW images, representing the main OS, that finally starts the different tasks in charge of the main features of the chip.

The image header contains all the information about the signature and the key that needs to be used with it, the version of the image, the parts of the image that contain executable code, and so on (see 1).

```
struct SignedHeader {
    uint32_t magic;        // -1 (thanks, boot_sys!)
    uint32_t signature[96];
    uint32_t img_chk_;     // top 32b of expected img_hash
    // everything below is part of img_hash
    uint32_t tag[7];       // words 0-6 of RWR/FWR
    uint32_t keyid;        // word 7 of RWR
    uint32_t key[96];      // public key to verify signature
    uint32_t image_size;
    uint32_t ro_base;      // readonly region
    uint32_t ro_max;
    uint32_t rx_base;      // executable region
    uint32_t rx_max;
    uint32_t fusemap[FUSE_MAX / (8 * sizeof(uint32_t))];
    uint32_t infomap[INFO_MAX / (8 * sizeof(uint32_t))];
    uint32_t epoch_;       // word 7 of FWR
    uint32_t major_;       // keyladder count
    uint32_t minor_;
    uint64_t timestamp_;   // time of signing
```

```
    uint32_t p4cl_;
    // bits to and with FUSE_FW_DEFINED_BROM_APPLYSEC
    uint32_t applysec_;
    // bits to mesh with FUSE_FW_DEFINED_BROM_CONFIG1
    uint32_t config1_;
...
```

Listing 1. Extract of the image header C structure from EC source code

The Boot ROM and the loader use similar initialization and verification steps when launching the next image. We can summarize them as follows.

First, they select the most recent candidate by comparing their versions computed from the fields `epoch`, `major`, `minor` and `timestamp` present in the images' headers.

Then, they verify that the magic number in the header is equal to `0xfffffffe` (-2 if used as a signed integer). Indeed, this value can be changed during the update process to disable an image (see Section 2.3).

Finally, the signature of the candidate is verified. Several SHA-256 hashes are computed: one from the full image and two others from values that are present in particular memory regions, a flash region called `INFO` and a second one that we called `fuses` (which is a memory region where bits can be turned to 1 only once). Only some values from these regions are retrieved based on the `fusemap` and `infomap` arrays present in the image header. This allows to protect against rollbacks. A final hash is computed from the previous hashes and given to the hardware component responsible for verifying the signature. The key used for the verification is selected using the `keyid` field of the image header from a set of keys already present in data of either the Boot ROM or the loader.

2.3 Firmware Update

There are two ways to change the firmware of the Titan M chip: the first one is a firmware update mechanism implemented in the nugget task, and the second one is a rescue feature implemented in the loader of the chip.

The firmware update mechanism implemented in the nugget task allows the Android system to send a new firmware image to the Titan M. Two commands are used by the Android system to achieve this:

– The first one takes as an input an address, a byte buffer and the first 4 bytes of the SHA-1 hash of the byte buffer. The command handler writes the buffer at the corresponding address. Of course, not

all addresses are allowed, only the unused RO and RW images can be overwritten. The buffer size is `0x800` bytes, so this command needs to be called several times to update a complete image. In addition, the nugget task replaces the magic number of the new image header by another value so that the image is ignored by the loader at boot.
– To reactivate the new image and have it considered as a candidate at boot, there is another command that takes as an input a hash value that is derived from the user password. If the hash is valid, then the nugget task replaces the magic number in the image header by its regular value (`0xfffffffe`).

Note that there is no image verification during the update mechanism. But there is a signature verification performed on the image at boot, and this prevents the system from booting on a non-legitimate image. Also, the user password is needed to activate the new image. Without it, the chip does not boot on the new image.

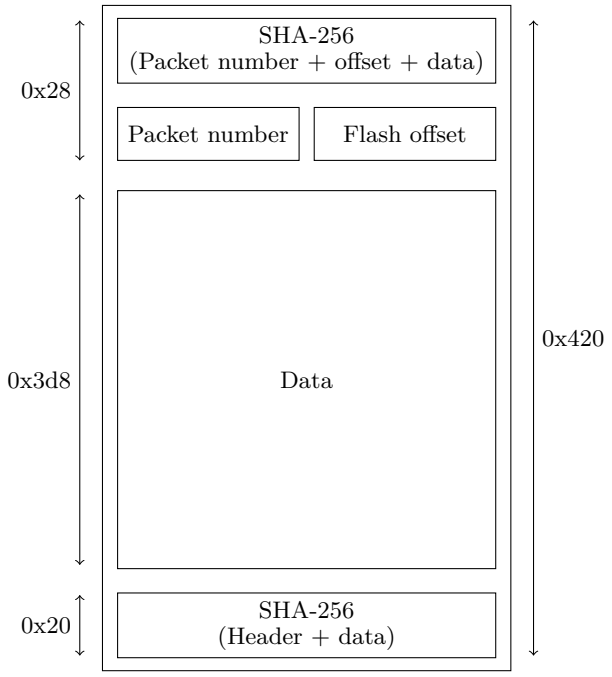

Fig. 1. Format of a packet from the *rec* file

There is another mechanism to change the firmware, which is the rescue feature. This one is implemented in the loader of the chip, and

allows to flash the `RW_A` image without the user password. Here again, the firmware is transmitted from the main CPU to the Titan M. The packets contain a SHA-256 hash of the data, a packet number, an offset that is used to compute the address in flash memory, and finally a byte buffer that is written at the provided address. With this mechanism, the magic number of the image is not changed, which means that the new image can be considered as a candidate at next boot. However, when this feature is triggered, and before updating the image, the chip erases all the user data, the secrets such as the keys and the `RW_B` image. This feature can be triggered from the bootloader of the main CPU when it is in `fastboot` mode, through the specific command *"oem citadel rescue"*. It takes as an input the image in a specific format, with the extension "*.rec*". In this format, the image is divided into chunks, each one containing extra metadata: a number, a flash offset and some SHA-256 hashes (see Figure 1).

2.4 Firmware Security

From a security point of view, the security of the firmware benefits from its simplicity. Since the chip does not use dynamic allocation, entire classes of bugs are avoided. In addition to that, in this section we analyze the actual security features implemented and the attack surface exposed.

First, on the Android side, we need to remark that root access is required to interact with the chip's driver and send custom messages. This is already a security protection against tampering with Titan M: the Android Platform Security Model features a sandboxing mechanism that prevents processes to start with superuser privileges [22]. Such mechanism can be bypassed through the so-called *rooting*, which implies modifying the system to ignore access control protections [20]. Root access can be obtained intentionally by the user, or maliciously through exploitation of a vulnerability (or a combination of them). Either way, this is the first step required for an exploit chain targeting the secure chip. Still, this does not lower the impact of a vulnerability on Titan M: in fact, the hardware module should be resistant to attacks even if the kernel is fully compromised.

The Titan M does not feature the standard protections that can be found on more complex operating systems. Being based on ARM Cortex-M3, the chip has no *Memory Management Unit (MMU)*, thus measures like *Address Space Layout Randomization (ASLR)* and such cannot be adopted. Despite that, there are two practical exploit mitigation techniques implemented against memory corruption vulnerabilities.

The first one relies on the hardware support given by the chip itself. The ARM Cortex-M3 CPU features an optional *Memory Protection Unit (MPU)*, which allows to divide the memory map into up to 8 regions, each of them with a specific location, size, attributes and permissions [2]. The MPU allows to set a region as non-executable and, in practice, this is used to disable instruction fetching on the stack. While reversing the firmware, we can find the appropriate functions that manipulate the MPU configuration, by accessing some hardware registers mapped at addresses from `0xe000ed90` to `0xe000edb8`.

The chip also comes with another mechanism to configure memory regions, that can be used to provide read and write permission, on top of the MPU configuration. In fact, this mechanism is the main one used by the system to configure the permission of the flash memory and some particular RAM regions. Each region is defined by three registers:

- A base address register indicating the start of the region;
- A size register which indicates the size of the region;
- And finally, a control register that represents the state of the region (if enabled or disabled) and the permission associated to it (read and/or write).

The concept of memory region exists in EC, but the regions in the Titan M are mapped at different addresses. For example, the flash region 0 base address register is mapped at the address `0x40100270` in the chip. We found 6 flash regions used in the Titan M. Flash regions 0 and 1 protect the code of the active *RO* and *RW*, usually with read-only permissions while the regions 2 and 3 protect the code of the inactive *RO* and *RW* with read and write permissions. And the other flash regions are used to configure the permissions of the memory containing the persistent data. There are also two staging flash regions that are used by the Boot ROM and the loader to validate the signature of the candidate images at boot. As of now, it is unclear how the execution permission is given to the active flash region. We believe the chip uses a custom mechanism tied to the staging regions and the signature verification hardware but we did not validate this theory by testing.

The firmware also contains a simple software control to detect memory overflows: the stack area of each task is initialized with a hardcoded stack canary, of value `0xdeadd00d`. The scheduler (that runs in handler mode, hence using a separate stack) checks the content of the address pointed by the process stack pointer before switching tasks, raising an interrupt that leads to a reboot if the canary is not found.

Stack canaries are a common exploit mitigation technique, which theoretically aims at increasing the difficulty for an attacker trying to gain code execution using an out-of-bounds write primitive. The inherent effectiveness of such a technique, though, relies on having the canary set to a random value, so that the attacker cannot easily predict it and include it in their malicious payload. Alternatively, the canary can include null bytes, which cannot be placed in a malicious payload if the out-of-bounds write consists in a function manipulating a C-string. Clearly, none of these properties is met in this case. Combined with the fairly easy access to the firmware file, this protection is therefore practically useless, since finding the canary value for an attacker is quite straightforward. Given these considerations, this measure may have been implemented just as an error detection mechanism, without aiming at improving the security of the chip. Initializing the stack with a recognizable value, in fact, allows the chip to observe when too much memory is used.

2.5 Communication with Android

In its functioning, the Titan M is seen as a security peripheral of the Android device and the two interact in a client-server architecture. At hardware level, the communication is done on the *Serial Peripheral Interface (SPI)* bus, connecting the security chip with the application processor.

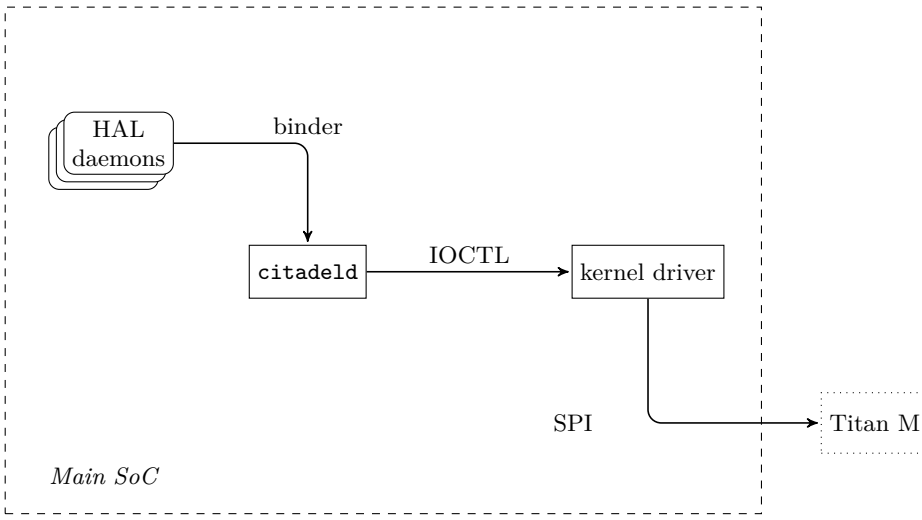

Fig. 2. Sending SPI command from Android

On Android, a kernel driver is in charge of handling the SPI communications. It is possible to communicate with the driver from the userland through *input-output control* (IOCTL) calls on the device driver `/dev/citadel0`. The Android daemon `citadeld` is actually the only one that communicates with the driver. Its role is to dispatch the commands coming from other Android components: the *Hardware Abstraction Layer* (*HAL*) services. *HAL* services act as an interface for the Android runtime to communicates with specific hardware. They expose a generic API to the rest of the system that should be the same across Pixel devices. In general, there is one HAL daemon per functionality. For example, in the case of Titan M we can mention the daemons `keymaster`, `identity` and `weaver`. The command's data are generated with `Protobuf`,[4] a well-known serialization project made by Google. The protobuf definitions for Titan M tasks are open source and part of AOSP. This is really helpful for the reverse engineering task, and we use them in one of our tools (see Section 3.3). But it is also interesting from the security point of view since it reduces the risk of introducing bugs while writing custom parsers. Finally, commands are sent to `citadeld` along with an application and a command id, through the Android interprocess communication driver: `binder` (more specifically, the `vndbinder` driver, dedicated to vendor services).

On the Titan M side, an SPI driver is responsible for reassembling the commands from the SPI packets. The driver is using a global array that contains, for each task that communicates over SPI, two callbacks for incoming and outgoing messages, a unique application id, and a structure that defines among others the buffers where the received data and the data to be sent should be placed. This is why each SPI packet contains an application id, so the driver can select the callback of the right task. Once a command is fully reassembled, the driver triggers an event that is received by the task of the corresponding application id. Most of the tasks use the library `nanopb`,[5] a protobuf library designed for embedded software, to handle the command requests and reply data. The nugget task is the only exception and the parsing of the request data is implemented directly in the task.

2.6 StrongBox and Keymaster

One of the most interesting tasks in Titan M is Keymaster. This is the trusted application associated to StrongBox, the highest level of security

[4] `https://developers.google.com/protocol-buffers`
[5] `https://github.com/nanopb/nanopb`

for keys generated using the Android Keystore system [13]. Keymaster is the largest task on the chip. It implements the logic to generate, use and attest cryptographic keys, as well as perform several other operations with them.

By design, StrongBox keys must never appear in plaintext in the main OS: cryptographic operations in fact only happen inside the hardware trusted module. Similarly to ARM TrustZone, Titan M does not store any key material: keys are encrypted with a *Key Encryption Key (KEK)*, and stored in the device as *key blobs*. A key blob is a structure meant to hold the actual key in the Android file system. These blobs are stored in `/data/misc/keystore/`. To keep track of the keys' owners, Android relies on the actual name of the files (that contains the application user id) and their Linux permissions.

A direct consequence of this architecture is that `root` can use any key, by requesting encryption or decryption operations while impersonating other apps. Note that this also holds for StrongBox keys: the Android Security Model only ensures that those keys cannot be *extracted* from the secure chip, but cannot prevent them from being used [22]. In practice, it is impossible for Titan M to check whether a request is legitimate: once it receives a message with a key blob, it uses it to perform the operation requested. The countermeasure offered by the Android system is authentication-bound keys. That is, if a key is created with `setUserAuthenticationRequired(true)`, the user will have to authenticate themselves using biometrics, whenever the key has to be used.[6] This clearly introduces some friction, but represents an acceptable trade-off for the most security-sensitive cases.

To encrypt StrongBox keys, Titan M uses a KEK derived from several components. Among them, one of the most interesting ones is the *Root of Trust*. This is a SHA-256 digest sent by the Pixel's bootloader, using the `SetRootOfTrust` command of the Keymaster task. The operation is performed whenever the chip is not initialized and can only be done in bootloader mode.

Whenever the Root of Trust is set, the Titan M also updates a 32-byte *salt* with random bytes generated locally. This salt is also used to craft the KEK. These values are stored in a memory area of the chip called SFS, together with other fields containing additional tokens and status information. The Keymaster section of the SFS region is 228-byte long.

[6] `https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder#setUserAuthenticationRequired(boolean)`

3 Tools

In this section we present the tools we developed during this research. This way, we also follow the evolution of our study and how we tackled the challenges and difficulties we had to face. All the tools are available at `https://github.com/quarkslab/titanm`.

3.1 Ghidra loader

The first tool we developed is a Ghidra [4] loader. This allows to open a Titan M firmware on the well-known disassembler, mapping the various regions at the right addresses thanks to the header files in the AOSP.

In addition, we created some further memory regions that we found while analyzing the firmware. Examples are the RAM partition, and some regions where hardware registers are mapped. Memory-mapped hardware registers are common in embedded devices: to communicate with peripherals, the CPU reads and writes data to specific memory addresses in the same address space as the main memory. In the case of Titan M, instances of these peripherals are the cryptographic accelerator or the MPU.

As mentioned previously, the firmware is a plain raw file, relatively simple to be reverse engineered. It also contains several debugging strings, among which some can be found in the EC source code. The combination of this background knowledge available and this tool makes reversing the chip even more accessible.

3.2 Frida tracer

Despite reversing being particularly effective with the Titan M firmware, there are some limitations to this approach. Some features of the chip are in fact particularly optimized and at low level, thus building a full picture of the internals is not trivial. In addition to that, interactions with hardware modules cannot be analyzed, since those functions often simply wait for the peripheral to complete an operation, to then read the result returned.

Based on these considerations, a dynamic view over the chip's functioning would help in understanding its state machines and internals. Debugging or instrumenting the Titan M is not possible, but we can explore where to hook on the Android side.

Starting from a high level, we can use the Android Studio debugger to follow some library calls that we know are supposed to interact with

the secure chip (e.g. generation of a StrongBox key). This is a useful and straightforward way to see which components forward a message within the Android OS from a user application to the kernel, but the view stands at an abstraction level too far from our target.

Instead, heading back to the AOSP, we can review the code at `platform/external/nos/host`, searching for interesting functions to trace. In the citadeld daemon, we identified in `nos_call_application` a good target, as it is called whenever the daemon wants to send a message to the Titan M.[7] This function takes as arguments the application and command identifier, the request, and the response with their respective lengths (the latter is filled by the function itself). We use the Frida dynamic instrumentation framework to trace calls to this function [3]. Thanks to the `Interceptor` API, we can attach to a function exported by a shared library, in this case `libnos_transport`, and explore its arguments before it starts to execute and before it returns.

We developed a tracing script that can be run while hooking the `citadeld` daemon. In such a script, we dump the memory region containing the request and the response. By interacting with a parsing library (developed together with `nosclient`, as shown in Section 3.3), these messages are also deserialized and printed on the Android log.

In addition to passively observing the messages, we can also modify the parameters passed to the function, by overwriting them in memory. In other words, with this approach we are able to simulate any interaction with Titan M and dynamically test it.

Although effective to start exploring the details of the communication protocol, this solution clearly has some limitations for a larger scale analysis. In fact, whenever we want to call the traced function with custom arguments, we have to make Android generate a legitimate command, to then alter the parameters. A convenient solution for this is `/bin/keystore_cli_v2`, another utility present in the device to generate, use, and delete keys from the command line. By writing a more complex Frida script, we actually only need to forge a valid request once, to record the memory addresses used. We can then alter their content and call `nos_call_application` with our new parameters.

3.3 Custom client

Despite these improvements, it is preferable to adopt a different strategy, allowing us to communicate with the chip in a more linear way, with more

[7] `https://android.googlesource.com/platform/external/nos/host/generic/+/refs/heads/android11-release/libnos_transport/transport.c#452`

efficient and automatic interactions. As mentioned, the Android system exposes the libraries responsible for communicating with Titan M, which are normally used by `citadeld`. What we can do, however, is writing a custom client that directly connects to the driver, bypassing the citadel daemon. Such a tool, which we call `nosclient`, is a binary compiled with the Android *Native Development Kit (NDK)*, and allows to send fully customized messages to the secure module. It does not require any special configuration on the device, apart from root privileges to open the driver and stop `citadeld`, as only one process can be communicating with the driver at the same time.

This client is the main tool we used during this research. Leveraging on the open source Protobuf definitions, it allows us to generate any type of command and analyze the response returned by the library functions. `nosclient` can be launched in an Android shell as a command-line script. Directly from its arguments, it can invoke a command defined with protobuf. We also implemented custom commands to create proofs of concepts or to exploit vulnerabilities (as explained in Section 5.2).

4 Hardware reversing

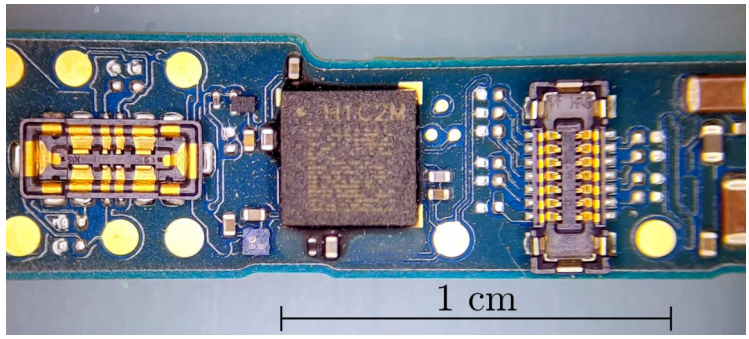

Fig. 3. Titan M on the PCB of a Google Pixel 3

The tools we presented enabled us to widely investigate the chip's interaction with the Android system dynamically. There is, however, a last limitation that is still in place. Both the Frida tracer and the custom client require the device to be fully booted to work. While this is generally not a serious problem, there are some interesting commands sent to the Titan M while the device is in bootloader mode. These are impossible to

trace from the Android perspective, therefore we need to focus on the hardware level and physically tamper with the SPI bus.

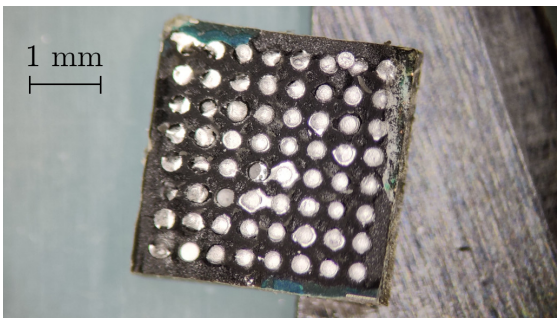

Fig. 4. Titan M contacts and underfill epoxy

The quickest way would be to find tracks on the PCB where the SPI bus is apparent, then cut tracks and take control of it. Unfortunately, the PCB is very dense and composed of many buried layers so it is not that simple.

Physically, the Titan M is in a 3.6×3.6 mm *Ball Grid Array (BGA)* package featuring 64 contacts on 13 mm^2, as shown in Figures 3 and 4.

Once the chip got desoldered, we probed the PCB contacts and other points of the PCB with a continuity tester to reconstruct the pinout, but besides ground, power rails, phone buttons and the debug UART (already accessible via debug pads), we could not find much. Then we experimented with a cheap *Vector Network Analyzer (VNA)* and a needle, to see roughly if there is a PCB track behind each BGA connector or not. If there is a track, it acts a bit like an antenna and we can observe some changes in the *reflection coefficient* (S_{11}) at high frequencies. The results are integrated in Figure 5 which was completed in the next step. White cells mean we believe there is no corresponding track, yellow ones correspond to the detection of a high impedance (or capacitance) on the corresponding pad, revealing the presence of a track (but of unknown functionality). This step helped prioritizing our reverse engineering efforts on the promising pads.

As we did not get quite the information we were seeking, we attempted to design a 10-layer flexible PCB to slip between the Titan M and the PCB and get access to the signals, but the production price was prohibitive and the results uncertain. Therefore we moved to the radical (some will say crazy) approach of rewiring the chip manually with tiny wires, using a

Fig. 5. Partially reversed pinout of the Titan M

Thin Quad Flat Package (TQFP) breakout board as intermediate support, as shown in Figures 6 and 7. Just in case, we wired all the 64 connections.

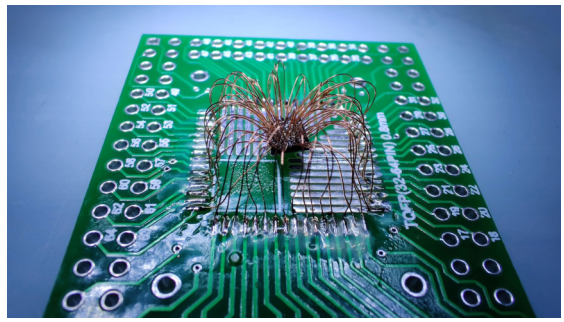

Fig. 6. Titan M wired on a TQFP-64 breakout board

We powered the phone and... were relieved when it booted successfully. This setup allowed to probe easily all the lines with an oscilloscope while the phone was booting and running. We could identify among other things the SPI bus, shown in green on Figure 5. The SPI bus is clocked by the CPU at 1.2 MHz except during the bootloader phase when it's clocked at 2.4 MHz.

The next step was to sniff the SPI bus during different operations. The example in Listing 2 shows some commands exchanges between the

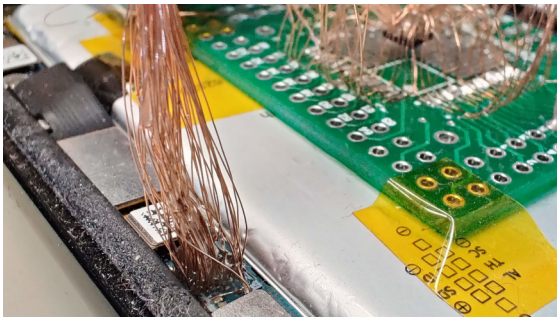

Fig. 7. Breakout board wired back to the PCB

bootloader of the main CPU and the Titan M, after the SPI rescue feature is used. Indeed, this feature also erases all the user data and secret from the chip. So the booloader of the main CPU has to initialize it by sending, among others, the Root of Trust through the command `setRootOfTrust`.

```
$ LD_PRELOAD=./libparser.so python \
    parse_sigrok-csv.py reboot_after_spi_rescue.csv
...
AVB: GetLock
{
    IN
    {
        lock: BOOT
    }
    OUT {}
}
Keymaster: SetRootOfTrust
{
    IN
    {
        digest:
            "4bf5122f344554c53bde2ebb8cd2b7e3
             d1600ad631c385a5d7cce23c7785459a"
    }
    OUT {}
}
Keymaster: SetBootState
{
    IN
    {
        is_unlocked: true
        public_key:
            "0000000000000000000000000000000000
             0000000000000000000000000000000000"
```

```
        color: BOOT_UNVERIFIED_ORANGE
        system_version: 163840
        system_security_level: 10568
        boot_hash:
            "00dfccb48f331975a1390d5133ce5321
             e65123bc1f1f76b6ffb9deb61f5d6be8"
    }
    OUT {}
}
...
```

Listing 2. Trace of data sent by the bootloader during initialization phase

Using a logic analyzer we can trace the data going on the SPI bus, and extract them into a *CSV* file containing the SPI packets. We made a simple python script to parse this file in order to rebuild the different commands. Then, using the protobuf definitions available, we made a parser library whose role is to print in a human readable way the content of the different commands.

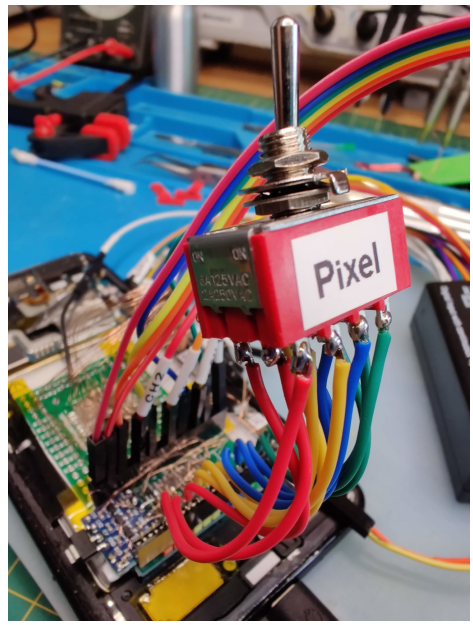

Fig. 8. 4PDT switch to swap SPI Controllers

To be able to take control of the Titan M over the SPI bus, one must connect the Titan M to another SPI controller. We chose to use a Raspberry Pi which offers SPI functionality on its pins header. The Titan M I/O works at 1.8 V while the Raspberry Pi SPI operates at 3.3 V, therefore we introduced a level shifter in charge of the voltage conversion across the SPI bus lines and a 1.8 V voltage regulator as no such voltage source is not directly accessible from the pinout. We cannot have two SPI controllers wired to the same peripheral, therefore we introduce a 12-pin 4-pole double throw toggle (4PDT) on-on switch shown in Figure 8 to choose which controller to connect to the Titan M: the Pixel 3 CPU or our Raspberry Pi. We also connect the debug UART to the Raspberry Pi to get a setup able to exploit the first vulnerability described in the next section. To reduce the risk of bad signals over our hectic wiring, we operate the SPI at a slower speed (300 kHz).

5 Vulnerabilities and Exploits

In this section, we present the vulnerabilities we found on the Titan M. We disclosed all of them to Google, following the community guidelines. The full disclosure timeline will be published in the coming month in Quarkslab's blog.[8]

5.1 Out-of-Bounds Read in Nugget task (CVE-2021-0939)

The first vulnerability we found is an out-of-bounds read that allows an attacker to leak parts of the memory of the chip. The vulnerability is present in the Nugget task, in the portion of the code that handles the command *UART passthrough*.

```
void nugget_ap_uart_passthru(uint index)
{
[...]
  if (PASSTHRU != index)
    cprint(4,"passthru %s",(&string_array)[index]);

[...]
```

Listing 3. Out-of-bounds read vulnerability

[8] https://blog.quarkslab.com/

This command takes a byte value as input, which is then used as an index in a string array to print a message on the UART console. The string array contains five strings, but the index value can be up to 255.

```
char** string_array = {
    0x65c00, // -> "off"
    0x65c04, // -> "usb"
    0x65c08, // -> "ap"
    0x65c0b, // -> "ssc"
    0x65c0f, // -> "citadel"
    // end of the array
    0x4004002c,
    0x0
    [...]
```

Listing 4. Reachable memory

As a consequence, an attacker can use an index value bigger than 5, so that the system prints a string placed on one of the addresses following the string array. By choosing a value carefully, it can be used to leak critical data from the memory of the chip. We wrote a script that prints all the reachable addresses through this vulnerability. We have been particularly interested by addresses from 0 and 0x1000, since they correspond to the memory region where the Boot ROM is. It means that we can use this vulnerability to leak parts of the Boot ROM.

A first limitation we encountered in our attempts to exploit this vulnerability was that the vulnerable code requires the main CPU to be in bootloader mode. And to overcome this limitation, we had to use our hardware setup described in the Section 4. First we boot the main CPU in fastboot mode so that it stays in the bootloader mode while waiting for a user input. Then we switch the SPI controller and use our Raspberry Pi to send our crafted command.

A second limitation is due to the fact that the system prints a string on the UART console, and not a byte array. While printing a string buffer, the system prints all the bytes of the buffer up to the first `0x00` byte encountered. And because it is ARM code, it is most likely that there will be some `0x00` bytes in the Boot ROM code. This is why we can only leak part of the Boot ROM using this vulnerability.

This vulnerability is referred by the number *CVE-2021-0939*. Google rated this vulnerability as high and fixed it with the Android security patch level 2021-01-10 [6].

5.2 Firmware Downgrade with SPI Rescue (CVE N/A)

The second vulnerability we found is more serious and allows to downgrade the Titan M firmware. Indeed, while an efused value is taken into account during the image verification phase at boot (see Section 2.2), it seems that these values are never changed when there is a new firmware. Therefore, it is possible to use the SPI rescue feature to downgrade the firmware to any one of the existing old versions.

We introduced the SPI rescue in Section 2.3. The command, which can be sent from fastboot mode, overwrites the RW A image, deleting the other one and the user data stored on the chip. During our research, we discovered that, providing a `rec` file containing an image older than the one in execution, we can successfully downgrade the firmware. We reversed the format and it does not require any authentication. This is why a *rec* file can be easily generated from a firmware file.

This vulnerability has a significant impact, as it allows an attacker to rollback the Titan M, to a version that can be attacked thanks to a known vulnerability. The fact that the SPI rescue deletes the secrets stored on the chip mitigates the consequences of such an attack, which, however, remain important. Google rated this vulnerability as high, and should fix it in the next bulletin.

Exploiting a Known Vulnerability We apply the aforementioned attack strategy to exploit a known vulnerability. This way, we explore how to mount an exploit against Titan M, a task that lets us interact with the protections in place and test their effectiveness.

Vulnerabilities in Android are reported on a monthly basis in the Android Security bulletin [10]. Very few of them involve Titan M and the related CVEs lack details. Automatic binary diffing, with tools like `bindiff` [27], is certainly an option, which, nonetheless, can be ineffective when we do not have a defined area to focus on. In any case, once a potential vulnerability is found, it remains non-trivial to understand if it is reachable by a maliciously crafted command or whether it can indeed lead to code execution.

After considering different ones, we investigate a vulnerability present in the firmware version *0.0.3/brick_v0.0.8232-b1e3ea340*, released on December 2020. Fixed in the March 2021 bulletin [14], this vulnerability could correspond to either CVE-2021-0454, CVE-2021-0455 or CVE-2021-0456: all of them have the same description, thus we cannot specify which one we are referring to.

The vulnerability is in the handler of the `ICpushReaderCert` command from the Identity task. The associated request includes a byte array and a series of 4-byte values corresponding to offsets and sizes of components of the byte array, as shown in Listing 5.

After decoding the request, the firmware parses the `x509Cert` buffer, retrieving its sections according to the specified offsets and sizes. Such fragments are then copied in a structure (which we call `ic_struct`) located in a global memory area, outside of the Identity task. This operation (performed using `memcpy`) is done without any check on the size of the source buffer. As a result, since we can control both the content of the buffer and its size, an overflow is possible on the global structure.

```
message ICpushReaderCertRequest {
  bytes  x509Cert = 1;
  uint32 tbsCertificateOffset = 2;
  uint32 tbsCertificateSize = 3;
  uint32 signatureOffset = 4;
  uint32 signatureSize = 5;
  uint32 publicKeyOffset = 6;
  uint32 publicKeySize = 7;
  uint32 signAlg = 8;
}
```

Listing 5. The Protobuf definition of the targeted request

Since `ic_struct` is not allocated on the stack of the vulnerable function, we cannot hijack execution by simply overwriting the saved return address pushed to the stack. Nonetheless, right after the structure, we can find some runtime information related to the management of the commands: among them, the address of the functions used to handle the communication on the SPI bus, and the list of callbacks associated with each command.

The exploitation strategy is therefore the following. First, we send an `ICpushReaderCert` command to overflow `ic_struct` and overwrite the callback related to the first SPI command (`GetState` from the AVB task). We can do this by simply crafting a message with a large `x509Cert`. Then, we send an empty AVB `GetState` request, which triggers the callback we have just overwritten. Note that we do not need to include the stack canary in our payload, as the structure is allocated in a shared memory area and not on the stack of a task.

As for the "new" callback value, we can write the address of an existing function in the firmware. This is a practical solution to show a proof of concept for a successful exploitation, but such an attack is not particularly

powerful. Instead, we can place there the first *gadget* of a *Return Oriented Programming (ROP)* chain, which allows to execute different fragments of code on the firmware [25].

To mount this type of attack, however, we have to control the stack of the AVB task (that is the context in which our attack is executed), where to place our sequence of gadgets. This is achieved by first calculating the expected stack pointer: since we know its initial value, we only need to traverse how the functions of the task manipulate it before jumping to the overwritten pointer. Once we calculated this value, we can include it in our `ICpushReaderCert` command, at an offset where it overwrites another pointer close to `ic_struct`: the address of the buffer where the Nugget requests received from the SPI are stored. This way, we can add a call to a Nugget command to our exploitation strategy, sending our ROP-chain, which will be copied to the overwritten address. Thanks to this, when we finally send the AVB `GetState` command, the stack already contains the sequence of gadgets composing our attack. Figure 9 graphically reports how we exploit the buffer overflow.

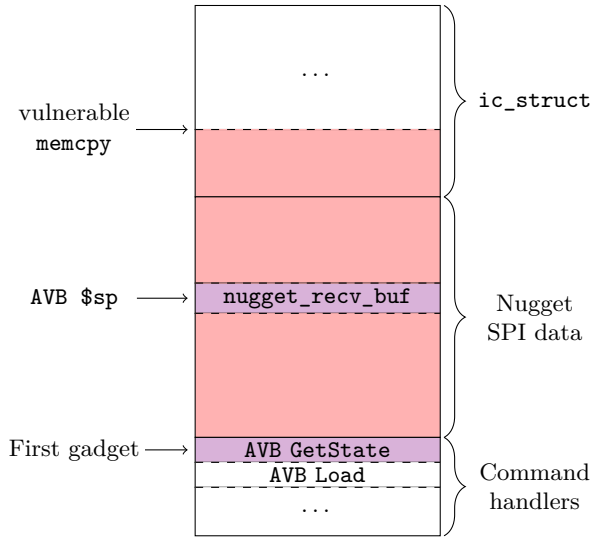

Fig. 9. The memory area interested by the `ICpushReaderCert` vulnerability.

At this point, we successfully achieve code execution on Titan M. With this approach, we obtain control of the instruction pointer in the context of a task, therefore in thread mode. By mounting a slightly different attack, we can also overwrite a different pointer and gain code execution

in handler mode. This is the first known code execution exploit on the chip.

An exploit using the ROP technique is classified as a *code reuse* attack, since it relies on instructions that are already present on the target. By crafting some calls to the logging functions in the firmware, we create an exploit that leaks any value in memory accessible with read permission. This is a very useful primitive, as it enables memory inspection at runtime, which can provide insights also while studying other vulnerabilities (as we show in Section 6). In particular, we can successfully extract the Boot ROM, thus discovering the only component not available statically. Also, the vulnerability could have been exploited to read values from the SFS region: in the case of Keymaster, this means breaking StrongBox. In our case, this consequence is not present, since when we downgrade the device using the SPI rescue feature, we also delete all the secrets.

The main limitation of this attack, in our case, derives from the simplicity of the firmware. In fact, there is no "special" function that would allow us to get full control over the target. In other words, we cannot craft a call to `execve` or `system` to obtain a shell on the Titan M, because the OS does not support it. To achieve a similar result, instead, we have to write our own *shell code*, i.e. inject a sequence of instructions written by us and execute them.

The key challenge of this second approach is finding a memory area which is both writable and executable. In practice, no such region exists by default: as explained in Section 2.4, the MPU makes the RAM non-executable, writing on the current flash region is not allowed and tampering with the other one would imply failing the signature check that occurs before starting the image. Bypassing the MPU, by disabling or reconfiguring it, does not seem to produce successful results, and we suspect that an additional security protection may be configured with the other mechanism to define memory regions that we described in the Section 2.4.

We modified our `nosclient` (see Section 3.3), to implement the PoC that exploit the vulnerability. We used it to create a *leak* command which allows to leak arbritrary data from the chip memory. We found that command quite useful to inspect the chip memory, such as the *SFS* where the *root of trust* lies, or the Boot ROM of the chip that is quite interesting to understand the first stage of the boot process.

```
# ./nosclient leak 0x0 0x10
00 00 02 00 99 14 00 00 b9 3e 00 00 b9 3e 00 00 b9 3e
```

Listing 6. Example of `nosclient` usage to leak 16 bytes from address 0

6 Fuzzing

To automate vulnerability research, we explore how to use fuzzing on the Titan M. Fuzz testing is based on generating random inputs and feeding them to the target program, monitoring its behavior and checking whether the processing yielded a crash or an unexpected result [26].

Fuzzing is particularly powerful when we can instrument our target, either at compile time or at binary level. This improves the ability to find bugs, detecting them even if they do not lead to a crash. Even more importantly, it generates *coverage*, which can be used by the fuzzer to produce highly diversified inputs that exercise different portions of the program's state space.

Instrumentation, at any level, is not possible on the Titan M firmware, which is a good example of a *black-box* target. Also known as an *oracle*, a black-box target offers no detailed feedback during its execution, but returns a signal that we can use to determine whether an input was processed successfully.

Since we have an accurate view over the format of the messages, we can design a fuzzer based on a grammar, represented by the Protobuf definition of the commands. With such an approach, the fuzzer does not evolve the corpus based on the coverage generated on the target (which we do not have), but rather mutates it using some operators, randomly selected and applied while respecting the Protobuf model. These mutations are applied with the objective of triggering typical input management vulnerabilities, e.g. integer overflows.

To implement this solution, we start again with the `nosclient`. We remark that this native binary allows to send arbitrary messages to the chip, by directly communicating with the system driver. After sending a command, it retrieves a return code sent by Titan M, together with the actual body of the response. Thanks to the experiments conducted while exploiting the known vulnerability, we know that when a memory corruption vulnerability is triggered, the command returns an error code equal to 2 and reboots. We can find the definition of these error codes in a

header file in the AOSP, where 2 corresponds to `APP_ERROR_INTERNAL`.[9] This represents our signal to retrieve the result of a command: generalizing, when the return code is greater than 1, the input is worth further investigation.

As a mutator, we use `libprotobuf_mutator`, an open-source library that allows to randomly mutate Protobuf variables [5]. The most important function of the library is `Mutate`, which takes as arguments a Protobuf type and a size, and adds, deletes or mutates its fields while respecting the specified size. Integrating it into our tool is fairly simple, as we only need to continuously pick a message, apply the mutation, send it to the chip and evaluate the result. Figure 10 illustrates our fuzzing architecture.

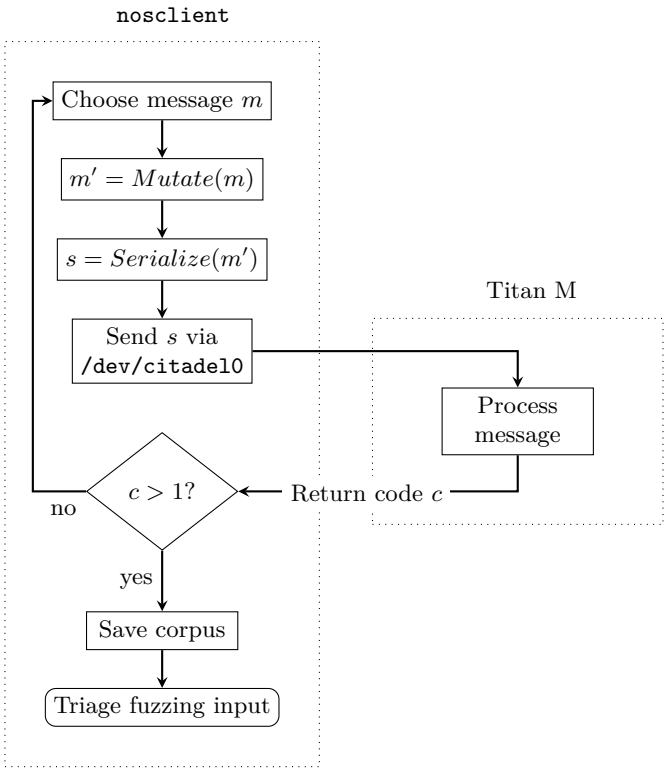

Fig. 10. The fuzzer workflow.

[9] `https://android.googlesource.com/platform/external/nos/host/generic/+/refs/heads/android11-release/nugget/include/application.h`

To implement our approach, we start by mutating requests of the tasks that communicate on the SPI using Protobuf. As mentioned, they are four in total: AVB, Keymaster, Identity, and Weaver. We decide to focus on the last three and exclude the first, since many AVB commands can only be sent in bootloader mode (to perform secure boot) and return application-specific error codes.

We run a first fuzzing campaign targeting the firmware version *0.0.3/brick_v0.0.8232-b1e3ea340*, the same as the one with the vulnerability we exploited in Section 5.2. Table 1 reports the results.

Table 1. Fuzzing results of the firmware version *0.0.3/brick_v0.0.8232-b1e3ea340*

Task	Command	Bug	Detection	Return code
Identity	`ICPushReaderCert`	Buffer overflow	Chip reboots	2
Identity	`ICsetAuthToken`	Buffer overflow	Stack canary	2
Identity	`WICaddAccessControlProfile`	Null-ptr deref.	Chip halts	4
Identity	`WICbeginAddEntry`	Null-ptr deref.	Chip halts	4
Identity	`WICfinishAddingEntries`	Null-ptr deref.	Chip halts	4
Identity	`ICstartRetrieveEntryValue`	Null-ptr deref.	Chip halts	4
Keymaster	`FinishAttestKey`	N/A	Chip reboots	2
Keymaster	`IdentityFinishAttestKey`	N/A	Chip reboots	2
Keymaster	`ContinueAttestKey`	N/A	Chip reboots	2

The `ICPushReaderCert` vulnerability has been successfully found by our fuzzer, proving its effectiveness. In addition, we find several other cases of inputs causing unexpected behavior from the chip. `ICsetAuthToken` contains a stack-based buffer overflow, that is detected thanks to the canary check. Four different command handlers instead generate a null-pointer dereference: in particular, the firmware retrieves a function pointer from a structure, initialized with null bytes and not yet filled, and executes it. This is a good use case for the read primitive built upon the previous vulnerability, to inspect this memory area and verify this assumption. On Titan M, though, a null address is actually valid and corresponds to the Boot ROM. This behavior is clearly unintended and causes a call to a function in the Boot ROM, which makes the chip halt. After triggering this vulnerability, the chip becomes unresponsive to the UART console and keeps returning error code 4, even to valid commands. The only way it can be restored is via a reset function exported by `libnos_datagram`, or a reboot of the phone. Finally, three Keymaster commands cause a

simple reboot: while this is probably not normal, we did not scope down any further, since these functions have been patched in the latest version.

These results allow us to validate the tool we developed and demonstrate the potential of the approach. Consequently, we run a second fuzzing campaign, this time targeting the latest version of the firmware at the time of writing (*0.0.3/brick_v0.0.8292-b3875afe2*), released in June 2021. Table 2 summarizes the results.

Table 2. Fuzzing results of the firmware version *0.0.3/brick_v0.0.8292-b3875afe2*

Task	Command	Bug	Detection	Return code
Identity	`WICfinishAddingEntries`	Null-ptr deref.	Chip halts	4
Identity	`ICstartRetrieveEntryValue`	Null-ptr deref.	Chip halts	4

As we can see from the table, the latest version of the firmware contains two vulnerable commands, with the same underlying function performing a null-pointer dereference that results in a call to a Boot ROM function. The vulnerability has been disclosed to Google, but it was not considered serious enough to be included in the monthly bulletin.

All the bugs have been consistently found by the fuzzer after few seconds of processing, with a throughput of approximately 75 messages per second. This is certainly a positive result and it suggests that the approach is promising. On the other hand, after quickly finding these crashes, the fuzzer does not encounter any further issue, even with hours of execution. The reason is probably a well-known limitation of black-box fuzzers, that is exploring only shallow states. Without any visibility over which code branches are taken by the inputs, the fuzzer may be only exercising the surface of the firmware.

These are some improvements that can be implemented to improve the fuzzer's coverage, while remaining in a black-box setting. First, we can start by checking the actual response returned by the commands, not only the return code. If a new response is received from a command, a new code branch has been exercised. In parallel, another option is exploring the UART output, following the same principle. To do this, though, we would have to change our fuzzing architecture, including another machine connected to the UART interface of Titan M (we cannot achieve this only with the Google Pixel device). Inevitably, this would impact the fuzzer throughput.

Despite promising, all these solutions eventually encounter a major limitation of fuzzers in general, that is testing stateful targets with complex relations between messages. For example, many Keymaster commands include a `KeyBlob` or an `OperationHandle` field, which has to be already initialized not to be discarded. Once we identify these relations (by reversing the firmware), we can create a corpus of valid messages and instruct the mutator not to alter certain fields. This is another possible improvement, although at some point we have a trade-off between the resources spent on reverse engineering and a more accurate fuzzer.

A completely different way to approach the problem is using *emulation*. After having succeeded in extracting the missing components of the firmware, we can emulate its execution, to both explore the internals and fuzz it. In this case, we would be in a *grey-box* configuration, in which we have full visibility over code coverage. Emulation is a very hot topic in embedded devices research; however, Titan M frequently interacts with hardware components (such as the random number generator, the cryptographic accelerator, etc.), and this represents an important challenge to be addressed by an emulation-based solution. To tackle it, a hybrid solution is to emulate only specific parts of the firmware, to simply investigate their execution or fuzz them.

7 Conclusion

This study offered a glimpse of the Titan M internals and usages in Android. Having access to the firmware file, to the sources of EC and some other tools in AOSP was really helping the reverse engineering task and allowed us to work out some of the mysteries behind this chip.

On the hardware side, we have been able to guess a large part of the pinout of the chip and to identify the most important buses, in particular the SPI bus. This allowed us to sniff and send commands at any time in the lifecycle of the chip, even when the main CPU is in bootloader mode. All this was done with low-cost hardware tools and handmade microsoldering. This was possible thanks to the architecture choice behind the Titan M chip: it is external to the main *SoC* and the buses used to communicate with the main *SoC* are exposed on the phone's PCB. Such analysis would have been different if the security chip were part of the main *SoC*, like SEP, the security chip by Apple, and as it seems to be with the Titan M2, the next security chip by Google, which should be released in November 2021 with the Pixel 6.

In addition, we discovered two important 0-day vulnerabilities, showing that mistakes still happen on such security chip. One of these vulnerabilities allowed us to downgrade the firmware and exploit a 1-day vulnerability which made possible to produce the first code execution proof of concept on the Titan M and lead us to leak the Boot ROM of the chip. We tried to push the vulnerability research forward by producing a black-box fuzzer which permitted to discover new issues, despite the limitations of the black-box approach.

All vulnerabilities we found had been reported to Google and are now fixed. We released the different tools and proofs of concepts we made for our research here: `https://github.com/quarkslab/titanm`.

References

1. Cortex-M3 Devices Generic User Guide. `https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/programmers-model/processor-mode-and-privilege-levels-for-software-execution`.
2. Cortex-M3 Devices Generic User Guide. `https://developer.arm.com/documentation/dui0552/a/cortex-m3-peripherals/optional-memory-protection-unit/`.
3. Frida – A world-class dynamic instrumentation framework. `https://www.frida.re/`.
4. Ghidra – software reverse engineering framework. `https://github.com/NationalSecurityAgency/ghidra`.
5. libprotobuf-mutator. `https://github.com/google/libprotobuf-mutator`.
6. Pixel Update Bulletin—October 2021. `https://source.android.com/security/bulletin/pixel/2021-10-01`.
7. Secure Enclave overview. `https://support.apple.com/en-gb/guide/security/sec59b0b31ff/web`.
8. ARM Security Technology Building a Secure System using TrustZone Technology. *Arm white paper*, page 108, 2009.
9. AOSP. A/B (Seamless) System Updates | Android Open Source Project. `https://source.android.com/devices/tech/ota/ab`.
10. AOSP. Android Security Bulletins. `https://source.android.com/security/bulletin`.
11. AOSP. CTS Test for Secure Element. `https://source.android.com/compatibility/cts/secure-element`.
12. AOSP. Gatekeeper. `https://source.android.com/security/authentication/gatekeeper`.
13. AOSP. Hardware-backed Keystore. `https://source.android.com/security/keystore`.
14. AOSP. Pixel Update Bulletin—March 2021. `https://source.android.com/security/bulletin/pixel/2021-03-01`, 2021.
15. Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, Minneapolis, MN, USA, June 2014. IEEE.

16. Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *arXiv:1801.01203 [cs]*, January 2018. arXiv: 1801.01203.
17. Ben Lapid and Avishai Wool. Cache-Attacks on the ARM TrustZone Implementations of AES-256 and AES-256-GCM via GPU-Based Analysis. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, Lecture Notes in Computer Science, pages 235–256, Cham, 2019. Springer International Publishing.
18. Jessica Lin. Expanding the Android Security Rewards Program. `https://security.googleblog.com/2019/11/expanding-android-security-rewards.html`, November 2019.
19. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv:1801.01207 [cs]*, January 2018. arXiv: 1801.01207.
20. René Mayrhofer. Android security trade-offs 1: Root access. `https://www.mayrhofer.eu.org/post/android-tradeoffs-1-rooting/`, May 2019.
21. René Mayrhofer, Vishwath Mohan, and Stephan Sigg. Adversary Models for Mobile Device Authentication. *arXiv:2009.10150 [cs]*, September 2020. arXiv: 2009.10150.
22. René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The Android Platform Security Model. *arXiv:1904.05572 [cs]*, December 2020. arXiv: 1904.05572.
23. Nagendra Modadugu and Bill Richardson. Building a Titan: Better security through a tiny chip. `https://security.googleblog.com/2018/10/building-titan-better-security-through.html`, October 2018.
24. Maxime Peterlin, Joffrey Guilbon, and Alexandre Adamski. Breaking Samsung's ARM TrustZone. `https://www.blackhat.com/us-19/briefings/schedule/#breaking-samsungs-arm-trustzone-14932`, August 2019.
25. Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1):2:1–2:34, March 2012.
26. Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021. Retrieved 2021-03-12 11:41:11+01:00.
27. Zynamics. BinDiff. `https://www.zynamics.com/bindiff.html`.