Reversing and Fuzzing the Google Titan M Chip

Damiano Melotti
University of Twente & Quarkslab
dmelotti@quarkslab.com

Maxime Rossi-Bellom
Quarkslab
mrossibellom@quarkslab.com

Andrea Continella
University of Twente
a.continella@utwente.nl

ABSTRACT

Google recently introduced a secure chip called Titan M in its Pixel smartphones, enabling the implementation of a Trusted Execution Environment (TEE) in Tamper Resistant Hardware. TEEs have been proven effective in reducing the attack surface exposed by smartphones, by protecting specific security-sensitive operations. However, studies have shown that TEE code and execution can also be targeted and exploited by attackers, therefore, studying their security lays the basis of the trust we have in their features.

In this paper, we provide the first security analysis of Titan M. First, we reverse engineer the firmware and we review the open source code in the Android OS that is responsible for the communication with the chip. By exploiting a known vulnerability, we then dynamically examine the memory layout and the internals of the chip. Finally, leveraging the acquired knowledge, we design and implement a structure-aware black-box fuzzer.

Using our fuzzer, we rediscover several known vulnerabilities after a few seconds of testing, proving the effectiveness of our solution. In addition, we identify and report a new vulnerability in the latest version of the firmware.

CCS CONCEPTS

• **Security and privacy** → **Trusted computing**; **Mobile platform security**; *Embedded systems security*.

KEYWORDS

Android Security, Trusted Execution Environments, Reverse Engineering, Vulnerability Research, Fuzzing

1 INTRODUCTION

In 2018, Google launched the Pixel 3 smartphone. Among its many improved features and specifications, this device was the first one embedding Titan M, a secure hardware module specifically built for Pixel devices to improve their level of security [40].

Deploying security measures at the hardware level is not new, as described in Section 2. However, it is not so common for mobile devices to have a dedicated chip, physically separated from the main CPU, implementing a Trusted Execution Environment (TEE) and ensuring tamper-resistant properties.

When the chip was announced, Google reported that its firmware would be open source [33]. To date, no source code has been published and not much information is available about the internals of this chip. Despite that, to motivate researchers into investigating this module, Google introduced a special reward of one million dollars for whoever can find a full-chain remote code execution exploit with persistence [27]. 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.

Given the lack of available research, in this paper we present the first investigation of Titan M. We start by focusing on reverse engineering the firmware of the chip. In parallel, we review the code in the Android Open Source Project (AOSP) that enables the communication with the chip. Using dynamic instrumentation, we trace the messages exchanged between the Android OS and Titan M, validating the hypothesis we derive from our static analysis. We then exploit a known vulnerability to understand the challenges related to gaining control of the chip and to obtain a picture on the implemented protections. Finally, we design and develop a fuzzer that, based on the grammar of the exchanged messages, automatically tests the chip using a black-box approach.

In summary, we make the following contributions:

- We study the main components and internals of the Titan M firmware, how it communicates with Android and which software protections it presents; in addition, we demonstrate the exploitability of a known vulnerability and provide the first known code execution exploit for Titan M.
- Based on the knowledge obtained through our reversing, we design a structure-aware black-box fuzzer, which mutates commands—together with their parameters—accepted by the chip and then sends them to the chip via the Android kernel.
- We implement our fuzzer and test it against both an old version of the firmware and the latest one, discovering several known vulnerabilities and a 0-day bug, demonstrating the effectiveness of our approach.

In the spirit of open science, we make the code developed for this work publicly available at https://github.com/quarkslab/titanm. We responsibly reported all our findings to the affected vendors following the ethical guidelines established in our community.

2 BACKGROUND

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 [31], and considering that the extremely large computing base of a modern OS cannot be fully trusted, vendors started to leverage hardware components to improve the security of their systems. Among the different solutions proposed in the past [17, 38], the concept of Trusted Execution Environment (TEE) was introduced.

A TEE is an isolated computing area where the executed code and processed data are protected in terms of confidentiality, integrity and authenticity [37]. With a TEE, a new division takes place in the OS, between the *non-secure world* and the *secure world*. The secure world normally runs *Trusted Applications (TA)*, authorized software performing security-sensitive operations from within the trusted environment. As a result of this separation, even if the non-secure environment is fully compromised, an attacker is physically separated from the secure one and cannot tamper with it.

Concretely, there are three ways to implement a TEE [35].

- Virtual Processor: this is the most widely adopted approach and it consists in separating hardware resources within the same chip, implementing the secure and non-secure world as execution modes of the main CPU. ARM TrustZone is certainly the most notable instance of this solution [9].
- On-SoC Processor: this solution is used by Apple in its Secure Enclave [8]. 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 (thus running in the non-secure world) 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 [13].

One of the main features brought by Titan M is attack surface reduction. 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 chip also mitigates classic hardware-level exploits such as Rowhammer [24, 39], Spectre, and Meltdown [25, 28, 33]. The presence of dedicated Tamper Resistant Hardware (TRH) guarantees improved resistance against side-channel attacks, which are one of the main factors influencing this design choice [26, 32].

3 REVERSE ENGINEERING

We start our research by focusing on the firmware of the chip, which can be found in the filesystem of a Google Pixel 3 smartphone, at the path `/vendor/firmware/citadel`. The firmware 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, using the Ghidra disassembler and decompiler [4]; on the other hand, we gather additional knowledge by reviewing 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 an abbreviation for *Nugget OS*, which might be a code name for chip's operating system. In addition, Titan M is mostly referred to as *citadel*.

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 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 3.2). By using this utility and exploring its sources, we can better understand how the Titan integrates with Android and how the system communicates with it, to gather some specific information.

3.1 Firmware Internals

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. The RO images contain the bootloader, which verifies and starts the main OS in the RW partition. The verification step is based on a cryptographic signature, as to prevent firmware modification attacks. Each image is duplicated to support A/B updates, ensuring that a valid image is always present on the device during an update [11]. Based on such a layout, we create a custom loader in Ghidra, allowing us to correctly map the memory of the binary file and start reversing.

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. By design, the OS does not use dynamic allocation, thus all the memory required by a task must be explicitly defined at compile time.

Titan M is based on an ARM Cortex-M3 CPU [33], which features no *Memory Management Unit (MMU)*: the memory layout is therefore static. An optional Memory Protection Unit (MPU) can be activated to divide the memory space into different regions with some attributes, as explained in Section 3.3. The chip features an

---

internal flash memory and 64 KB of RAM. Since the RAM is very small, the code is executed directly from the flash memory.

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 at the time of writing (*0.0.3-/brick_v0.0.8292-b3875afe2*, released in June 2021) 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;
- `AVB`: the Android Verified Boot TA;
- `KEYMASTER`: the Keymaster TA, corresponding to the *Strongbox* API in the Android Keystore [15];
- `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* [14]);
- `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.

In its functioning, the chip is inevitably bound to 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 secure module with the application processor. When a message arrives on this bus, an interrupt is raised and the dedicated routine is executed. This function iterates on the list of tasks that use the SPI: Nugget, AVB, Keymaster, Weaver and Identity.

On top of SPI, an actual protocol is implemented (cf. Section 3.2), which defines a set of commands composed of a request and a response. The SPI driver on the Titan M side reconstructs the command received from several SPI packets, and then copies it to the memory section of the right task. Some integrity checks are performed with a simple CRC. Finally, the system triggers an event, which makes the proper task start parsing the received command.

In particular, the parser function extracts the command identifier of the message and uses it to retrieve and call the right subroutine from a list present in a global memory area. Finally, after decoding the command, the operation requested by the Android system is performed. Figure 1 shows an example for a Keymaster command received.
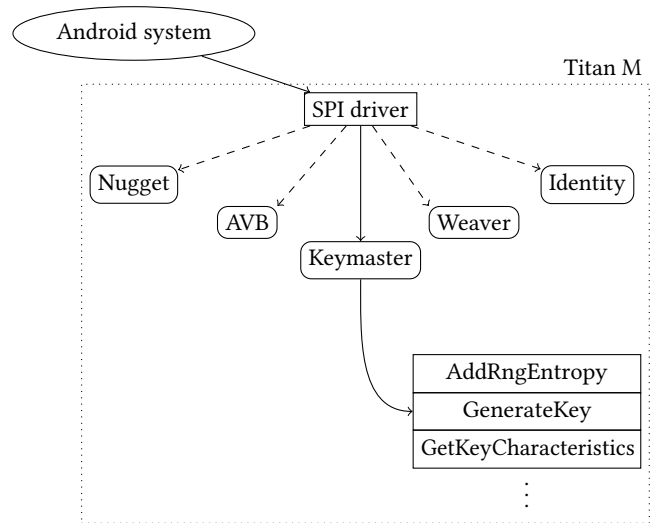

Figure 1: Reception and processing of an SPI command on Titan M.

The mechanism used to send a response, when available, is symmetric. Once the command handler has processed the request, another function writes it in a specific memory buffer, then notifies the driver that – in handler mode – applies the CRC code and sends the raw bytes on the SPI bus.

3.2 Communication with Android

The SPI messages exchanged with Titan M are encoded using *Protocol Buffers (Protobuf)*, a language-agnostic framework to serialize data [7]. As aforementioned, the firmware uses nanopb, a custom Protobuf implementation for microcontrollers, written in C [6].

The underlying `.proto` files, defining the format of these messages, are also present in the AOSP, hence we have a clear picture of what can be sent to (and received from) the chip via SPI. All the applications using this bus encode data with Protobuf, except for Nugget.

After having explored the life-cycle of a command once it arrives on Titan M, we can now briefly go over the components involved on the Android side, to find how we could investigate the exchanges dynamically. When application developers want to use one of the hardware-backed APIs, they usually simply have to write some high-level API calls. Behind these functions, a number of processes interact before finally sending the command to the chip.

Let us take as an example an application requesting a Strongbox AES key.[4] The API call forwards the request down to the Android Runtime. The runtime communicates with the dedicated Keystore daemon, which simply stands between the platform and the *Hardware Abstraction Layer (HAL)* and is only accessible by internal components. Once the execution reaches the HAL, the request is encoded using Protobuf and sent, via some further intermediate steps, to the citadel daemon (`citadeld`) using the `vndbinder` IPC.

---

[4]If the application uses the Java classes `KeyGenerator` and `KeyGenParameterSpec`, it needs to specify `.setIsStrongBoxBacked(true)` while building the `KeyGenParameterSpec`. This does not happen by default: such a request would throw a `StrongBoxUnavailableException` on devices without a TPM.
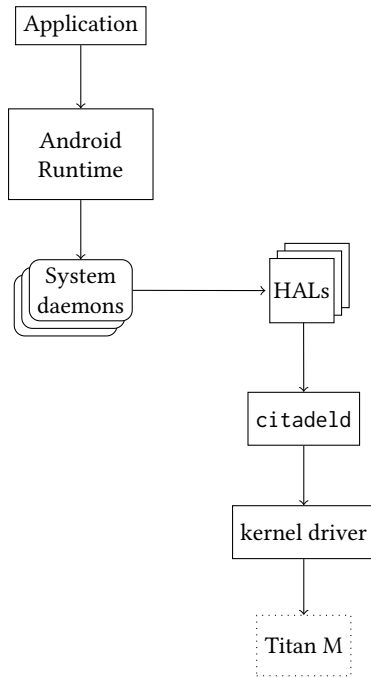
Figure 2: Android components interacting to send an SPI command to Titan M.

This daemon uses the `/dev/citadel0` driver to communicate with Titan M. `libnos_datagram` and `libnos_transport` are the two libraries that handle the actual transmission of the raw bytes on the SPI bus. Figure 2 schematizes the whole process on the Android side.

A key function used at this stage is `nos_call_application`, which 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). After having identified this interesting target, 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. 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 one, 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.

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 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. To achieve this, we develop `nosclient`, a binary compiled with the Android *Native Development Kit (NDK)*, which 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.

`nosclient` is the first step towards vulnerability analysis and automatic testing. With a relatively simple binary, we can craft our messages with any combination of values, encode them using Protobuf and send them leveraging the libraries from the AOSP. In the process, we also receive a return code after each command, from which we can derive the associated status, and the actual encoded response from the chip. In Section 4, we show how to put these elements together to fuzz the Titan M.

3.3 Security Features

Having built some background on the chip internals and its communication protocol, in this section we dive into the actual security features that can be found in the firmware and analyze the attack surface exposed. Other protections relate to hardware details, since Titan M is hardened against physical attacks with some defenses. These are, however, outside of the scope of this paper.

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 [32]. Such mechanism can be bypassed through the so-called *rooting*, which implies modifying the system to ignore access control protections [30]. 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.

As briefly mentioned above, the chip implements a boot security measure by checking the images signatures before launching them. In practice, the RO image selects the most recent RW image based on the version number, checks a magic number (both these values are specified in the header) and verifies the signature, which is computed on the rest of the header and the actual code. During updates, one command overwrites the (unused) RO and RW images, invalidating the associated magic numbers. Subsequently, another command restores them, to mark the new images as valid.

Another interesting feature accessible from Android, to interact with the firmware, is the *SPI rescue*. Using the `fastboot` command, included in the Android Software Development Kit, we can interact with the device in bootloader mode. `fastboot` exposes an `oem`

option to execute commands related to Original Equipment Manufacturer (OEM) components. Among them, we can interact with Titan M, i.e. *citadel*. With `fastboot oem citadel` commands, we can print information related to the chip, reset it and, most importantly *rescue* it. The rescue feature allows to flash a `rec` image, which overwrites the `RW_A` section, and wipes the user data stored on the chip. Since version *0.0.3/brick_v0.0.8232-b1e3ea340*, a `rec` file is present on the Android filesystem, at the same path as the full binary file of the firmware. Interestingly, with the `citadel rescue` command and an older `rec` file, we can downgrade the firmware of the Titan M. We reported this issue to Google, since an attacker could use this feature to flash a vulnerable version on the chip. The bug has now been fixed and has been assigned CVE-2021-1043.

Since the Titan M firmware is very simple, we do not observe the standard protections that can be found on more complex operating systems. Measures like *Address Space Layout Randomization* and such cannot be adopted, given the absence of virtual memory. The simplicity of the chip is, however, a strength point, as it is generally agreed that the larger a code base, the more vulnerabilities it contains. To this end, not using dynamic allocation is another design choice that excludes entire classes of bugs related to runtime memory management. Finally, relying on a third-party solution for decoding potentially untrusted messages (Protobuf and its `nanopb` implementation) also represents a good security practice, reducing the risk of input validation errors.

Indeed, memory corruption bugs are one of the main concerns in this scenario, as other case studies have shown [19, 22]. In addition to the security-centric design choices we just outlined, Titan M features at least two practical exploit mitigation measures against this type of attacks.

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 the corresponding registers.

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 aim 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.

3.4 Known Vulnerability Exploitation

All these findings are the result of static reverse engineering, tracking the first functions executed at boot, which are responsible for setting up the chip's memory. Despite the firmware not being obfuscated nor particularly hardened against reversing, this code uses hardware registers mapped in memory and optimized constructs, therefore building an accurate picture of the underlying operations is not trivial. In addition, parts of the logic related to the module set-up is contained in the *Boot Read-Only Memory (BROM)*, the only part of the firmware we do not have access to.

Given the aforementioned ability to downgrade the firmware, we can then explore the possibility of exploiting a known vulnerability in an older version of the firmware, with the goal of obtaining code execution and using it to gain a better visibility over the internals of Titan M.

Vulnerabilities in Android are reported on a monthly basis in the Android Security bulletin [12]. Very few of them involve Titan M and the related CVEs lack details. Automatic binary diffing, with tools like `bindiff` [43], 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 in December 2020. Fixed in the March 2021 bulletin [16], 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 entry 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 bytes values corresponding to offsets and sizes of components of the byte array, as shown in Listing 1.

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 1: 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 [36].

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 3 graphically reports how we exploit the buffer overflow.

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 (i.e., overwriting another function pointer in the same memory area, and triggering a call to it with an additional Nugget command), we can also 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 5). In particular, we can successfully extract the Boot ROM, achieving the initial goal of this task. The BROM is a simple loader that cannot be updated (thus truly *Read-Only*). At high level, it verifies the RO firmware section (which is instead more complex and can be updated) and launches it.
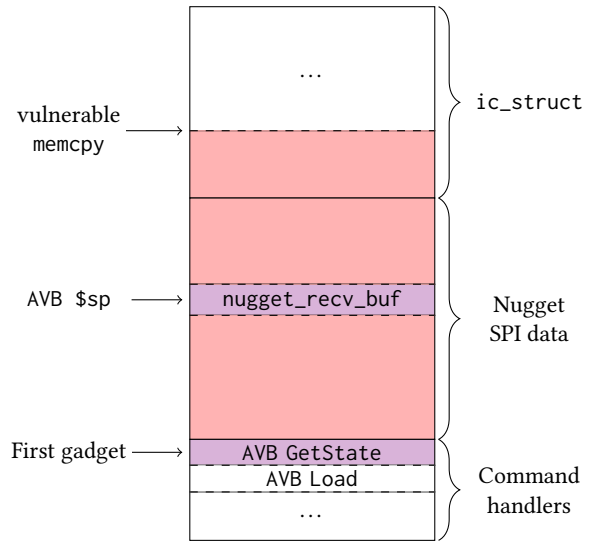

Figure 3: The memory area interested by the `ICpushReaderCert` vulnerability.

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. Since we have already obtained what we had targeted, we decide to leave this final step as future work.

In general, the key challenge with 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 3.3, 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. There is one potential solution to be considered: create a ROP-chain that either disables the MPU, or changes its configuration to make the RAM executable. We provide more details concerning this strategy in Section 6.

4 FUZZ TESTING

Combining the in-depth reverse engineering with the vulnerability exploitation study, we focus on vulnerability research. Using the background we accumulated, we design and implement a new strategy to security test Titan M, aiming at finding bugs like the one we analyzed in Section 3.4.

A common technique that has been proven effective to test software against this type of bugs is fuzz testing, or *fuzzing* [41]. This approach 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.

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 is not possible on the Titan M firmware, unless an exploit for an existing vulnerability is used. Being the code executed directly from the flash and considering the signature checks in place, binary rewriting is infeasible, both statically and dynamically. The chip is therefore a good example of a *black-box* target: also known as an *oracle*, it 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, selected randomly 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 `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 response body. 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`.[5] 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 4 illustrates our fuzzing architecture.

5 EXPERIMENTAL EVALUATION

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 conducted our experiments on a Pixel 3 smartphone, running Android 11 – the latest version available at the time. To obtain root access, we used the Magisk tool. 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 3.4. Table 1 reports the results.

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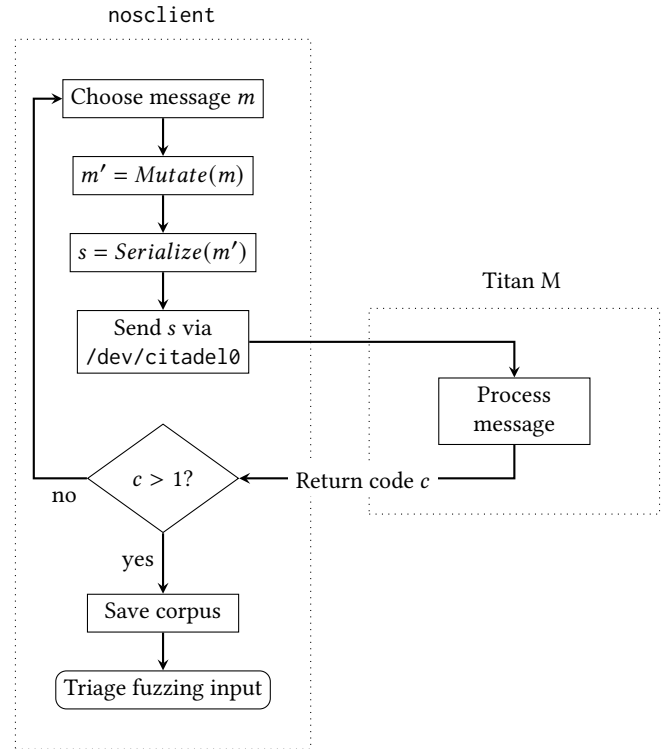

Figure 4: The fuzzer workflow.

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. 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, a null address is actually valid and corresponds to the Boot ROM's vector table. After applying a small offset, the firmware jumps to this address. This behavior is clearly unintended and causes a call to a function in the BROM, 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, two Keymaster commands cause a simple reboot. While this is probably not normal, we do not investigate 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 1: Results of fuzzing the Titan M firmware, version *0.0.3/brick_v0.0.8232-b1e3ea340*

Task	Command	Bug	Detection	Return code	Avg. # of messages
Identity	ICPushReaderCert	Buffer overflow	Chip reboots	2	74
Identity	ICsetAuthToken	Buffer overflow	Stack canary	2	475
Identity	WICaddAccessControlProfile	Null-pointer dereference	Chip halts	4	57
Identity	WICbeginAddEntry	Null-pointer dereference	Chip halts	4	99
Identity	WICfinishAddingEntries	Null-pointer dereference	Chip halts	4	82
Identity	ICstartRetrieveEntryValue	Null-pointer dereference	Chip halts	4	105
Keymaster	FinishAttestKey	N/A	Chip reboots	2	257
Keymaster	IdentityFinishAttestKey	N/A	Chip reboots	2	192

Table 2: Results of fuzzing the Titan M firmware, version *0.0.3/brick_v0.0.8292-b3875afe2*

Task	Command	Bug	Detection	Return code	Avg. # of messages
Identity	WICfinishAddingEntries	Null-pointer dereference	Chip halts	4	72
Identity	ICstartRetrieveEntryValue	Null-pointer dereference	Chip halts	4	126

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

All the bugs have been consistently found by the fuzzer after few hundreds of messages at most, with a throughput stabilizing to approximately 75 commands per second after the first crashes have been detected. 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. We discuss this and other limitations in the next section.

6 DISCUSSION AND FUTURE WORK

In this work we explored several different aspects of Titan M and its security, both statically and dynamically. By exploiting a known vulnerability, we actively simulated the challenges of an attacker trying to tamper with the secure chip and gain code execution. To this end, we left as future work the implementation of an exploit executing code injected on the chip. A possible approach to achieve this is manipulating the MPU and creating a region with both write and execute permissions. To gain persistence on the target, then, we would have to write the shell code on the flash and patch the signature checking logic, executed before launching the OS. Such integrity control would inevitably detect changes to the flash region.

From a vulnerability research perspective, gaining code execution certainly enables more capabilities. For example, we could mount a debugging server on the chip, allowing to break execution, inspect memory, or instrument the firmware for fuzzing. This would not be trivial anyway, due to the peculiar communication channel, but gives an idea of the potential features. In any case, the final goal would be to test the latest flashed image in a RW section,

while exploiting an older one in the other section to control the execution. While this task would technically be possible, assuming code execution on the chip, its feasibility needs to be investigated.

Concerning the fuzzer, all the potential improvements relate to improving its code coverage. As explained, we do not have a direct feedback source to guide the fuzzer, but there are some options to be explored. 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.

One last potential source of feedback could come from a side channel, such as the time required to execute a command. Execution times distant from the average may imply new coverage. Yet, most of the commands implement a fairly simple logic, thus inferring information would be challenging and subject to mere noise.

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.

It is worth remarking that having obtained access to the BROM, this is a particularly interesting section to test. In case a vulnerability is found there – either via fuzzing or static reversing – it could not be patched, as the memory is Read-Only.

Finally, Google recently presented the new Pixel 6 smartphone, which will feature a new version of the secure chip, Titan M2 [10]. No information is available at the moment, but this is certainly a new target for future studies based on our work.

Coordinated Disclosure

We disclosed all the vulnerabilities we found during this research to Google, following the ethical guidelines established in our community. In this section, we provide a summary of the disclosure process and timeline. We reported the SPI Rescue downgrade flaw on July 23rd, 2021. On July 29th the vendor requested additional details on our configuration and tooling, and on August 17th they assigned severity *high*. On October 22nd, Google communicated that a patch would be released in the November security bulletin and that the vulnerability was assigned CVE-2021-1043.

The null-pointer dereference found with fuzzing was reported on August 6th, 2021. On August 12th, the Android security team rated the bug as not serious enough to meet the severity bar for inclusion in a security bulletin. After we requested further details, Google clarified that the communication with Titan M does represent a security boundary, but a local temporary denial of service does not have bulletin class severity. Moreover, they determined that obtaining code execution based on this vulnerability is impossible, unless we could demonstrate that the dereferenced value could be controlled by an attacker. Anyway, they mentioned that the bug is likely to be addressed in a future update, as a code hygiene fix.

7   RELATED WORK

Since this is the first study on Titan M, there is no related work on this chip proposed by other researchers. However, there are relevant studies on similar scenarios or targets.

The most popular solution for building Trusted Execution Environments is ARM TrustZone, and the recent study from Cerdeira et al. well summarizes the most relevant research activities on its security [19]. The authors identified three main causes of vulnerabilities in TrustZone, namely critical implementation bugs, architectural deficiencies, and overlooked hardware properties. We investigated the first two on Titan M; the last point is outside of the scope of this paper, but reviewing the hardware protection of Titan M certainly represents an insightful research direction for future work.

Fleischer et al. published another study in which they also summarize the main findings on vulnerable TEEs and the challenges for exploitation [22]. In this paper, they note that all major commercially used TEEs were compromised, highlighting the intrinsic risks of implementing them relying on programming languages with explicit memory management (like C and C++). We share some of their conclusions to this end, given the findings on Titan M.

Focusing on Huawei's TEE implementation, Busch and Dirsch conducted a research that also starts from exploiting a known vulnerability to acquire more accurate information on their target [18]. They then propose a solution based on symbolic execution to investigate Trusted Applications. Symbolic execution is another approach that has not been considered in this work, as it shares similar challenges to emulation. Nonetheless, with enough knowledge of the target, this can represent a valid alternative.

Apple's Secure Enclave Processor (SEP) is another case study similar to ours, both from the chip architecture point of view and the closed nature of the system. In 2016, Mandt et al. presented a very complete research at Black Hat US, following a mostly black-box approach [29]. They tackled the chip from different perspectives, both on the hardware and software side. In short, they first analyzed how SEP physically integrates on iPhone devices, then how it communicates with the iOS kernel and how its firmware (SEPOS) works. Finally, they extensively analyzed its security properties, looking at robustness and attack surface. Their strategy and findings represented a point of reference for our project.

Shifting the focus to direct testing of embedded devices, other relevant works are [21, 42]. Both proposed novel techniques for firmware fuzzing, with solutions circumnavigating the typical problems of code coverage and throughput with emulation-based solutions. The goal of these researchers is to develop universal tools for vulnerability research, but their design choices can provide valuable insights. In another work suggesting the adoption of emulation combined with fuzzing, Muench et al. presented the common problems related to fuzzing embedded devices [34]. As mentioned in Section 4, what the researchers named *silent* memory corruptions cannot be detected by our testing architecture, while a set-up based on partial or full emulation would allow to investigate them too.

Finally, related work on firmware re-hosting includes HALUCI-NATOR [20] and, with the possibility of integrating an embedded coverage-guided fuzzer, PARTEMU [23].

8   CONCLUSION

In this paper, we provided the first study of the Titan M chip, recently introduced by Google in its Pixel smartphones. Despite being a key element in the security of these devices, no research is available on the subject and very little information is publicly available.

We approached the target from different perspectives: we statically reverse-engineered the firmware, we audited the available libraries on the Android repositories, and we dynamically examined its memory layout by exploiting a known vulnerability.

Then, we used the knowledge obtained through our study to design and implement a structure-aware black-box fuzzer, mutating valid Protobuf messages to automatically test the firmware. Leveraging our fuzzer, we identified several known vulnerabilities in a recent version of the firmware. Moreover, we discovered a 0-day vulnerability, which we responsibly disclosed to the vendor.

Our results demonstrate that our approach is effective, and there are several directions in which it can be improved, either via refining the black-box fuzzer or by changing perspective and exploring emulation. To foster new research on this subject, we released all the tools developed for this study at the following URL: https://github.com/quarkslab/titanm.

ACKNOWLEDGEMENTS

REFERENCES

[1] [n.d.]. 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] [n.d.]. Cortex-M3 Devices Generic User Guide. https://developer.arm.com/documentation/dui0552/a/cortex-m3-peripherals/optional-memory-protection-unit/.

[3] [n.d.]. Frida • A world-class dynamic instrumentation framework. https://www.frida.re/.

[4] [n.d.]. Ghidra. https://ghidra-sre.org/.

[5] [n.d.]. libprotobuf-mutator. https://github.com/google/libprotobuf-mutator.

[6] [n.d.]. Nanopb - Protocol Buffers for Embedded Systems. https://github.com/nanopb/nanopb.

[7] [n.d.]. Protocol Buffers. https://developers.google.com/protocol-buffers.

[8] [n.d.]. Secure Enclave overview. https://support.apple.com/en-gb/guide/security/sec59b0b31ff/web.

[9] 2009. ARM Security Technology Building a Secure System using TrustZone Technology. *Arm white paper* (2009), 108.

[10] 2021. Google Tensor debuts on the new Pixel 6 this fall. https://blog.google/products/pixel/google-tensor-debuts-new-pixel-6-fall/.

[11] AOSP. [n.d.]. A/B (Seamless) System Updates | Android Open Source Project. https://source.android.com/devices/tech/ota/ab.

[12] AOSP. [n.d.]. Android Security Bulletins. https://source.android.com/security/bulletin.

[13] AOSP. [n.d.]. CTS Test for Secure Element. https://source.android.com/compatibility/cts/secure-element.

[14] AOSP. [n.d.]. Gatekeeper. https://source.android.com/security/authentication/gatekeeper.

[15] AOSP. [n.d.]. Hardware-backed Keystore. https://source.android.com/security/keystore.

[16] AOSP. 2021. Pixel Update Bulletin—March 2021. https://source.android.com/security/bulletin/pixel/2021-03-01.

[17] Mohamed Amine Bouazzouni, Emmanuel Conchon, and Fabrice Peyrard. 2018. Trusted mobile computing: An overview of existing solutions. https://www.sciencedirect.com/science/article/pii/S0167739X16301510. *Future Generation Computer Systems* 80 (March 2018), 596–612. https://doi.org/10.1016/j.future.2016.05.033

[18] Marcel Busch and Kalle Dirsch. 2020. Finding 1-Day Vulnerabilities in Trusted Applications using Selective Symbolic Execution. https://www.ndss-symposium.org/wp-content/uploads/2020/04/bar2020-23014.pdf. In *Proceedings 2020 Workshop on Binary Analysis Research*. Internet Society, San Diego, CA. https://doi.org/10.14722/bar.2020.23014

[19] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1416–1432. https://doi.org/10.1109/SP40000.2020.00061 ISSN: 2375-1207.

[20] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. https://www.usenix.org/conference/usenixsecurity20/presentation/clements. 1201–1218.

[21] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. https://www.usenix.org/conference/usenixsecurity20/presentation/feng. 1237–1254.

[22] Fabian Fleischer, Marcel Busch, and Phillip Kuhrt. 2020. Memory corruption attacks within Android TEEs: a case study based on OP-TEE. https://doi.org/10.1145/3407023.3407072. In *Proceedings of the 15th International Conference on Availability, Reliability and Security (ARES '20)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3407023.3407072

[23] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. 2020. PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. https://www.usenix.org/conference/usenixsecurity20/presentation/harrison. 789–806.

[24] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. http://ieeexplore.ieee.org/document/6853210/. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, Minneapolis, MN, USA, 361–372. https://doi.org/10.1109/ISCA.2014.6853210

[25] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. http://arxiv.org/abs/1801.01203. *arXiv:1801.01203 [cs]* (Jan. 2018). arXiv: 1801.01203.

[26] Ben Lapid and Avishai Wool. 2019. Cache-Attacks on the ARM TrustZone Implementations of AES-256 and AES-256-GCM via GPU-Based Analysis. In *Selected Areas in Cryptography – SAC 2018 (Lecture Notes in Computer Science)*, Carlos Cid and Michael J. Jacobson Jr. (Eds.). Springer International Publishing, Cham, 235–256. https://doi.org/10.1007/978-3-030-10970-7_11

[27] Jessica Lin. 2019. Expanding the Android Security Rewards Program. https://security.googleblog.com/2019/11/expanding-android-security-rewards.html.

[28] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. http://arxiv.org/abs/1801.01207. *arXiv:1801.01207 [cs]* (Jan. 2018). arXiv: 1801.01207.

[29] T. Mandt, M. Solnik, and David Wang. 2016. Demystifying the Secure Enclave Processor. https://www.blackhat.com/us-16/briefings/schedule/#demystifying-the-secure-enclave-processor-3438.

[30] René Mayrhofer. 2019. Android security trade-offs 1: Root access. https://www.mayrhofer.eu.org/post/android-tradeoffs-1-rooting/.

[31] René Mayrhofer, Vishwath Mohan, and Stephan Sigg. 2020. Adversary Models for Mobile Device Authentication. http://arxiv.org/abs/2009.10150. *arXiv:2009.10150 [cs]* (Sept. 2020). arXiv: 2009.10150.

[32] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. 2020. The Android Platform Security Model. http://arxiv.org/abs/1904.05572. *arXiv:1904.05572 [cs]* (Dec. 2020). arXiv: 1904.05572.

[33] Nagendra Modadugu and Bill Richardson. 2018. Building a Titan: Better security through a tiny chip. https://security.googleblog.com/2018/10/building-titan-better-security-through.html.

[34] Marius Muench, Jan Stijohann, Frank Kargl, Aurelien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-4_Muench_paper.pdf. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2018.23166

[35] Maxime Peterlin, Joffrey Guilbon, and Alexandre Adamski. 2019. Breaking Samsung's ARM TrustZone. https://www.blackhat.com/us-19/briefings/schedule/#breaking-samsungs-arm-trustzone-14932.

[36] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. https://doi.org/10.1145/2133375.2133377. *ACM Transactions on Information and System Security* 15, 1 (March 2012), 2:1–2:34. https://doi.org/10.1145/2133375.2133377

[37] M. Sabt, M. Achemlal, and A. Bouabdallah. 2015. Trusted Execution Environment: What It is, and What It is Not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. 57–64. https://doi.org/10.1109/Trustcom.2015.357

[38] C. Shepherd, G. Arfaoui, I. Gurulian, R. P. Lee, K. Markantonakis, R. N. Akram, D. Sauveron, and E. Conchon. 2016. Secure and Trusted Execution: Past, Present, and Future - A Critical Review in the Context of the Internet of Things and Cyber-Physical Systems. In *2016 IEEE Trustcom/BigDataSE/ISPA*. 168–177. https://doi.org/10.1109/TrustCom.2016.0060 ISSN: 2324-9013.

[39] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[40] Xiaowen Xin. 2018. Titan M makes Pixel 3 our most secure phone yet. https://blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/.

[41] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/ Retrieved 2021-03-12 11:41:11+01:00.

[42] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng. 1099–1114.

[43] Zynamics. [n.d.]. BinDiff. https://www.zynamics.com/bindiff.html.