UNIVERSITÀ
DI TRENTO

Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

Diving into Starlink's User Terminal

Reverse engineering a modern satellite Internet antenna

Supervisor
Prof. Bruno Crispo

Student
Carlo Ramponi

Co-Supervisor
Maxime Rossi Bellom

Academic year 2022/2023

Acknowledgements

Many thanks to:

- Quarkslab, which made this amazing opportunity possible and welcomed me as part of the *Cryptobedded* team (i.e. the best team).
- Maxime Rossi Bellom, my internship tutor, for following and guiding me during my research.
- Damiano Melotti, my buddy, for making my life in Paris and the company much easier (i.e. How to Survive in Paris 101).
- Lennert Wouters, who was the author of the blog post regarding the dumping of the firmware and the fault injection attack of Starlink's User Terminal, for helping us in the early stages of this research.
- Tim Ferrell, from SpaceX's security team, for sending us a testing dish with root access.
- And many other amazing colleagues for helping me with topics in their fields of expertise.

Contents

Abstract

Starlink is a satellite-based Internet access service provided by Space X. This service already has more than 1.5 million subscribers all around the world [22], using the very same infrastructure. Starlink satellite internet has garnered a diverse and widespread user base since its inception. Initially, its primary target audience included rural and remote communities that lacked reliable broadband access, connecting them to high-speed internet previously out of reach. However, as the service expanded, it found applications in various sectors, such as maritime and aviation, providing connectivity to ships, planes, and even remote research stations. Additionally, Starlink has attracted users in regions with unreliable terrestrial internet infrastructure, including urban areas suffering from network congestion or frequent outages. In recent times, amid conflicts like the Ukrainian war, Starlink has played a crucial role by offering connectivity to war-torn regions where traditional infrastructure has been disrupted [21]. It has become a lifeline for civilians, aid organizations, and journalists, ensuring that critical information can be shared and emergency assistance coordinated despite challenging circumstances, which has also brought Russians to try and disrupt the service [12]. Starlink relies on 3 components:

- A user terminal, which communicates with the satellites, and on which most of the current research is focused.
- A satellite fleet acting as a mesh network.
- A gateway that connects the satellites to the internet.

Numerous studies have already been conducted on the subject [24, 23, 32, 28, 29], mainly on the user terminal, which is the only component of the infrastructure that anyone can buy and analyze. During my 6-month internship at Quarkslab as part of my Master's degree program at the University of Trento, I carried out the analysis of Starlink by reverse-engineering its firmware and the various protocols it uses. At the end of the internship, I gained a good knowledge of how the device works internally and developed a set of tools that could help other researchers working on the same topic. These tools will be described and published along with this work.

In the first part of this document, we introduce the subject of this work and the objectives (Chapter 1), then we describe how the overall Starlink network works with some more details (Chapter 2). After this, Chapter 3 contains a hardware analysis of Starlink's User Terminal and a description of the dumping process of the flash memory to extract its firmware, based on existing research. Following, Chapters 4 and 5 will show how the flash memory is organized, partitioned and verified, and how the secure boot procedure works. Chapter 6 describes how we've implemented an existing hardware attack to gain root access to the device, which, unfortunately, failed by breaking two devices. Back on the software side, Chapter 7 shows how the runtime is organized and how we've analyzed some binaries that run in it, and Chapter 8 shows how we dynamically tested runtime binaries in an emulated environment since, at the time, no physical device was available to us. The last part of the document (namely Chapters 9 and 10) shows where we focused and how we changed the objectives of this work, due to the encountered technical problems, by describing how the communication within the device (IPC) and with external devices is implemented, and by also fuzzing some parts of this software in different ways.

1 Introduction

This work was developed during my internship at Quarkslab in Paris during the spring and summer of 2023. The objective of the internship was to analyze Starlink's User Terminal, a device used to connect to satellites for internet service. This includes:

- Dumping the firmware of the device.
- Reverse-engineering the software running on the device.
- Trying to gain root access to the device to perform dynamic testing.
- Analyzing the network stack and protocols used to communicate with satellites.
- Performing vulnerability research and studying the attack vector.

Some of these objectives changed during the internship due to some complications in gaining access to the device and in the analysis of lower-level binaries.

Previous research on the User Terminal had already been performed by a few researchers, but still not many details were publicly available at the time. Other researchers focused on the hardware analysis of the device [24, 23, 32] and some high-level firmware analysis [28], without going into details, and finally, one researcher developed a hardware attack to gain root access [29], which we tried to replicate in Chapter 6. Some tools that interact with the APIs of the device that are used by the mobile application to act as alternative dashboards to inspect statistics gathered by the User Terminal have also been implemented [18, 19].

SpaceX's security team encourages responsible security research on their devices by providing testing hardware with root access on it [15], but we were only accepted in this program in late July, actually receiving the hardware in August, which was the last month of the internship, thus it wasn't much of help to us.

During this internship, I've also developed some tools to better analyze the firmware or the runtime of the device under study, which will be published as open-source code and will be discussed in the following chapters [27]. This research is meant to be a starting point and can be used as a reference for further security research on the device.

2 Starlink

Starlink is the new satellite internet solution provided by SpaceX, which offers greater speed and bandwidth compared to other solutions already present on the market, at a reasonable price. At the moment of writing, one can have a satellite internet connection in Europe for as little as 40-50€/month with a startup cost of around 300€ for the hardware [17]. Moreover, the bandwidth and latency of the connection are pretty decent, with Starlink you can reach up to 200Mbps and latency between 25 and 50 milliseconds, which is already way better than an ADSL connection and comparable to a typical FTTC setup.

All of this is only possible because of where Starlink's satellites are deployed. Most satellite internet providers, before Starlink, use geostationary satellites, which are 35786km far away from the Earth. A radio signal takes around 120ms to travel that distance, and since one wants to reach the internet, the signal has to travel from the final user to the satellite (1), from the satellite to a ground station connected to the internet (2), from the ground station to the server the user wants to reach and back, then again from the ground station to the satellite (3), and finally from the satellite to the final user (4). This means that the signal has to travel that distance 4 times, so the latency will be, without taking into account the latency from the ground station to the server and any processing time, at least

$$4 \times \left[\frac{35786km \times 1000\frac{m}{km}}{299792458\frac{m}{s}} \right] \times 1000\frac{ms}{s} \approx 477.5ms$$

And that is why existing satellite internet solutions that use this kind of satellite are not usable if the customer needs to use real-time applications such as video calls or gaming. As an example, in the third quarter of 2022, HughesNet had an average latency of 850ms and Viasat had an average latency of 720ms [10].

Starlink satellites are instead LEO (Low Earth Orbit) satellites, which are much closer to the Earth, at approximately 550 km. Making the same computations as before, a signal takes around 1.8ms, which sets a lower physical limit of

$$4 \times \left[\frac{550km \times 1000\frac{m}{km}}{299792458\frac{m}{s}} \right] \times 1000\frac{ms}{s} \approx 7.3ms$$

Because of this, from the same Ookla report, Starlink users experience a latency that can be lower than 100ms [10].

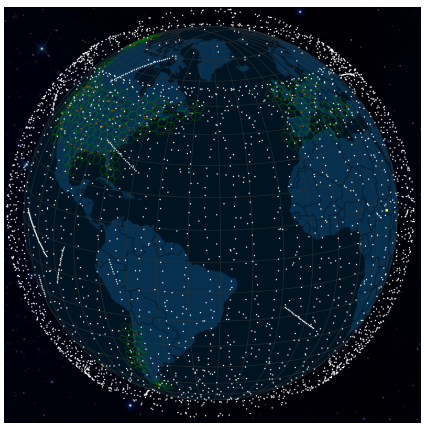


Figure 2.1: Starlink satellites [13]

Objects in this orbit need to move much faster than the Earth in order not to experience orbital decay, so to always have a visible satellite from a fixed point on the ground you need a lot of them to form a sort of mesh all around the globe. At the time of writing, Starlink has 3798 active satellites, 541 inactive satellites and 354 satellites that have reached the end of life or crashed for some reason (see Figure 2.1) [13], but Starlink plans to deploy nearly 12,000 satellites, with a possible later extension to 42,000.

Starlink satellites are also much smaller, lighter and cheaper than traditional geostationary satellites. Thanks to this, Starlink can deploy a lot of them together, up to 60 with a single launch, which drives down a lot of production and launch costs of a single satellite.

2.1 Components

The three main components of Starlink's network infrastructure are Satellites, User Terminals and Gateways. This research's focus is the User Terminal, here is a quick overview of all the components.

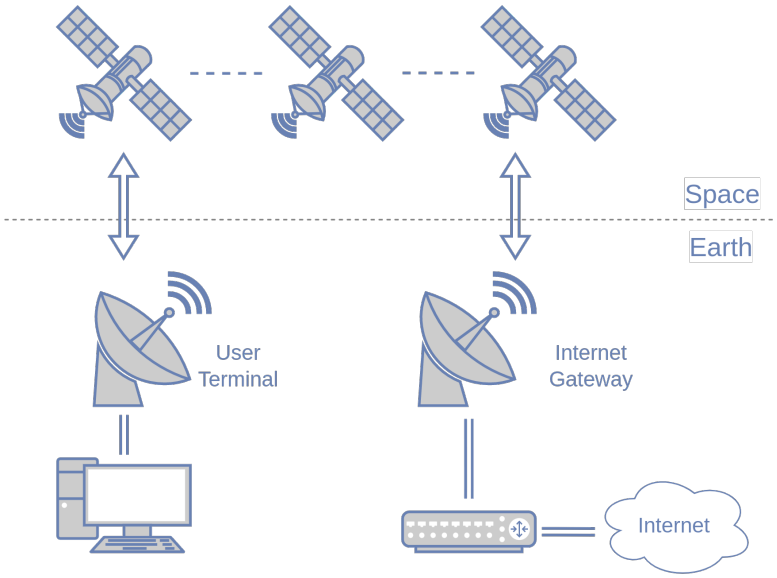


Figure 2.2: Starlink components

User Terminals (UTs)

The User Terminal is the only piece of hardware you can get your hands on, and it is also the focus of this research. This device handles communication with satellites and, on higher levels, routing packets from the internal (customer) network to Starlink's private network.

(a) Version 1 and 2

(b) Version 3

Figure 2.3: Starlink User Terminal [17]

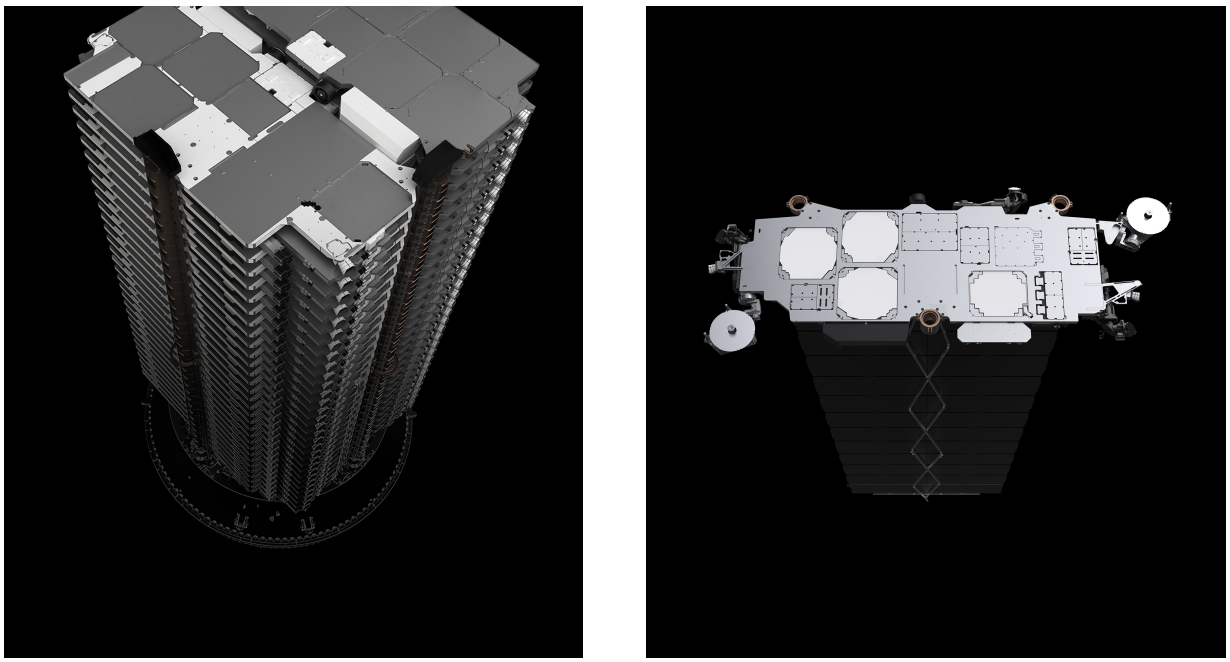
Figure 2.3 shows two different versions of the User Terminal. The antenna will be placed where it can directly see the sky (e.g. on a roof) and will then automatically tilt toward the best direction and continuously scan the sky for satellites to always connect to the optimal one. Customers will be able to connect to Starlink's network using a wifi access point directly connected to the User Terminal.

Gateways

Gateways are ground stations that are connected to the internet and can reach satellites. There currently are 147 operational gateways, placed all around the globe and many more are in the process of construction and/or approval [7]. They act, as the name suggests, as gateways for Starlink's private network, so when a customer tries to reach the Internet, this is what will be used as an exit point.
We know very little about these components, but the hardware is probably very similar to the one of User Terminals, perhaps more "powerful".

Satellites

Satellites are the missing piece to connect the User Terminals to the gateways. They act as a bridge between the two aforementioned devices, thus connecting the User Terminal to the Internet (through the gateways).



(a) 60 satellites, stacked for launch

(b) Satellite

Figure 2.4: 3D rendering of a satellite and the payload of a rocket launch [17]

Figure 2.4a shows how 60 of these satellites can be stacked and fit into Falcon 9 rocket launch, and figure 2.4b shows what a Starlink satellite looks like, it is equipped with [17]:

- a solar array panel;
- autonomous collision avoidance sensors;
- ion thrusters, for collision avoidance, orbit raise and deorbit at the end of life;
- star tracker, to determine the location, altitude and orientation of the satellite;
- optical space lasers (Optical Intersatellite Links), to communicate with each other (still testing);
- 4 phased array antennas and 2 parabolic antennas.

2.2 Radio communication

Starlink uses the K_u and K_a microwave bands to communicate with user terminals and gateways, more specifically the downlink for the K_u band is in the range $[10.7, 12.7]$ GHz, while the uplink is in $[14.0, 14.5]$ GHz, and for the K_a band downlink and uplink are in $[17.7, 19.7]$ GHz and $[27.5, 30.0]$ GHz, respectively. While for modulation they use Quadrature Phase-Shift Keying (QPSK) and Binary Phase-Shift Keying (BPSK) in the K_u band and more advanced modulation schemes such as 8-PSK and 16-APSK in the K_a band [20]. Every component (UT, Sat, gw) is equipped with (at least) a phased array antenna which creates a beam of radio waves that can be electronically steered to point toward the receiving device, without physically turning the antenna. This is particularly useful in this case because satellites are moving w.r.t. the Earth, and switches from one satellite to another have to be made frequently by ground stations.

Starlink is still in an early stage of deployment (the goal is circa 12,000 satellites in multiple orbits). As of now, results achieved by the new LEO constellation are very promising and this could be a game changer for internet service in rural areas.

3 User Terminal teardown

The device we analyzed is the User Terminal version 2 (`REV 2, PROTO 3`), which is the round dish. Several other researchers [23, 24, 32] have already deeply described the hardware of this device.

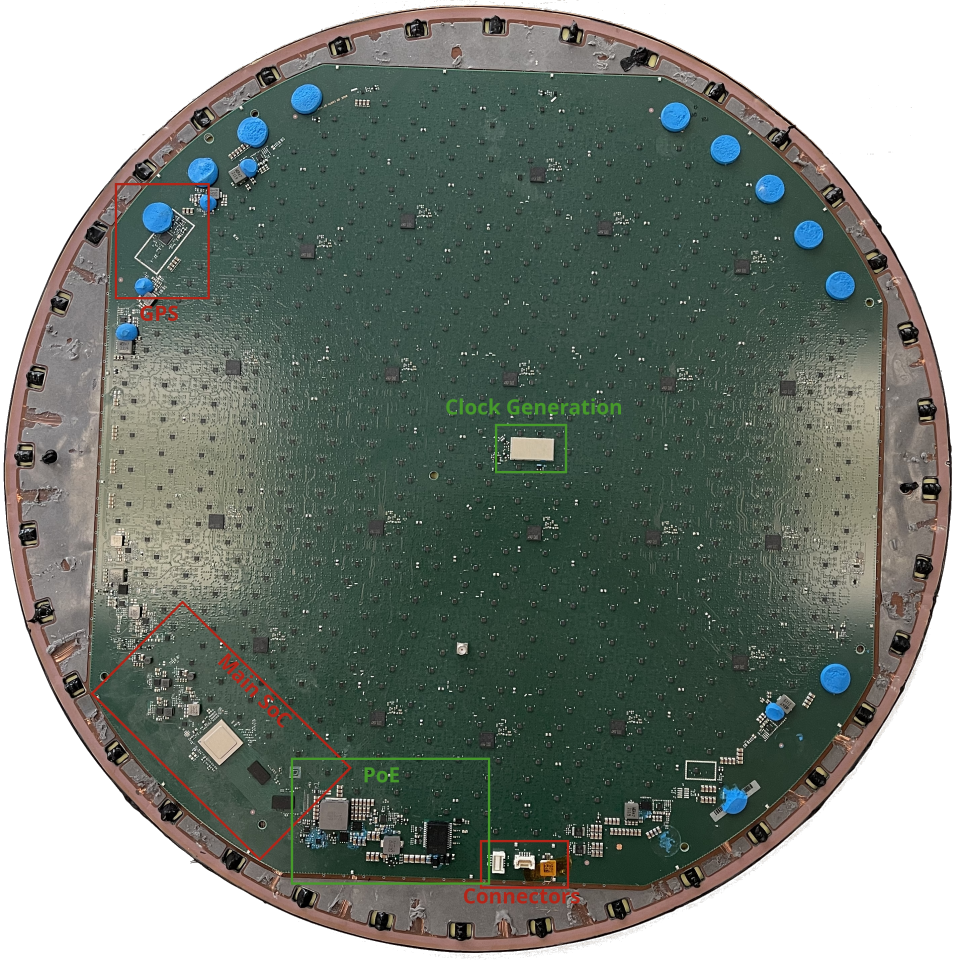


Figure 3.1: Disassembled Starlink User Terminal

Most of the area on the board is used for RF hardware, which is composed of a set of cells, containing a Digital BeamFormer (DBF), namely `STM GLLB-SUABBBA` and several Front-End Modules (FEMs), namely `GEAAA21132DH501`, shown in Figure 3.2. This set of Digital BeamFormers forms a phased array antenna, which can "tilt" the transmitted signal toward a specific direction (i.e. the satellite) without actually tilting the antenna.

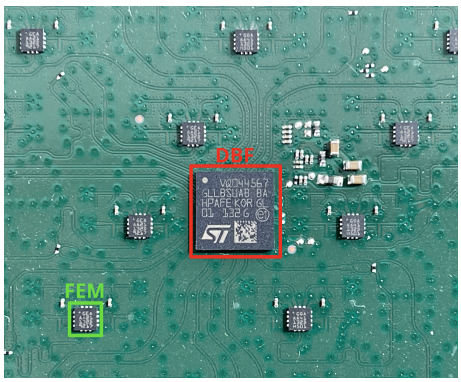


Figure 3.2: RF Hardware

Other than that, there is a GPS chip (`STM STA8089`) and a chip for clock generation (`GLLBLU`), probably used by the RF components, some connectors, a circuit that handles power over ethernet and, of course, the main System-on-Chip (SoC).

The connectors, shown in Figure 3.3, are also accessible without fully disassembling the device. The first one on the left is where the ethernet cable that comes from the provided power supply (versions 1-2) or the wifi router (version 3) is connected, and serves both data and power (Power over Ethernet, PoE), it doesn't use any IEEE 802.3* standard though, because it requires a lot of power (up to 180W at 56V DC) [3]. The second one is the controller for the motors that tilt the antenna. Finally, the third one, which is a set of test pins, is the UART interface, which had been disabled in the version of the dish we were analyzing.

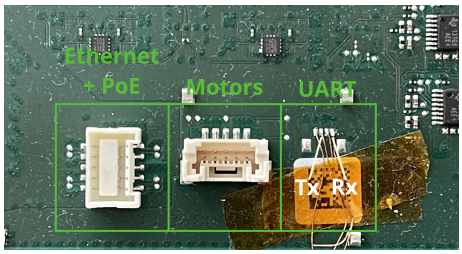


Figure 3.3: UT Connectors

3.1 System-on-Chip overview

The main System-on-Chip of the board is shown in the following picture.

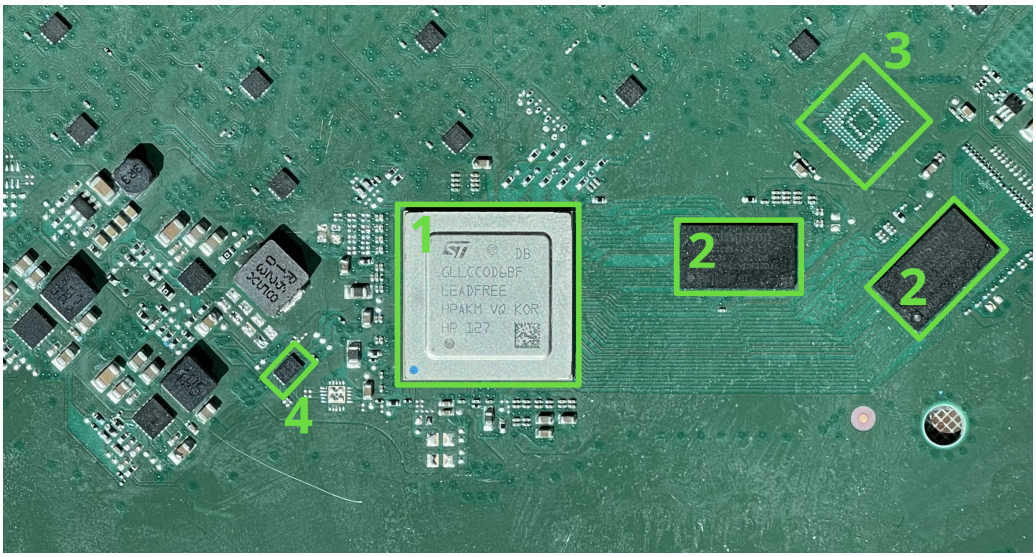


Figure 3.4: Main System-on-Chip

The main components are:

1. The microcontroller (`STM GLLCCODGBF`, codename: `CATSON`) is a custom quad-core ARM Cortex-A53 processor.
2. The RAM is composed of two DDR3 chips of 4Gbit each.
3. The persistent memory (removed in this picture) is a 4GB eMMC.
4. The secure element is an `STM STSAFE-A110`

From a visual inspection of the hardware, one can also understand the specific version and cut it is being analyzed: 5 "fixed" GPIO pins are either connected to the ground or 3.3V, see Figure 3.5.
And from the bootloader code [31]:

```
#define BOARD_REV_2_3P0 "#rev2_proto3"
// [...]
case 0b10001:
    board_rev_string = BOARD_REV_2_3P0;
// [...]
```

We can say that our board is the User Terminal version 2 cut 3.

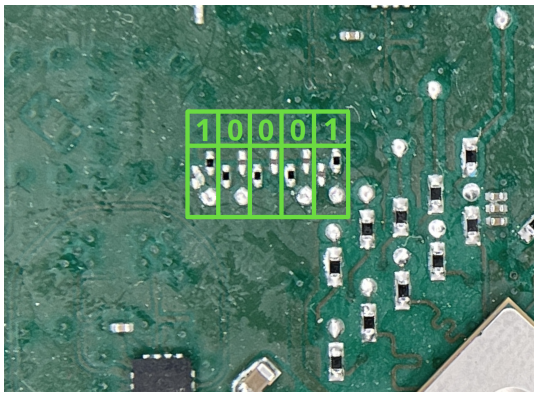


Figure 3.5: Hardware version GPIO pins

3.2 Persistent memory dump

To get the firmware of the User Terminal (which is not publicly available), we used a procedure explained by the COSIC research group at KU Leuven in a blog post [28].

We first identified the test pins of the eMMC chip, which are:

- The Clock pin (CLK), to synchronize the host and the eMMC chip
- The Command pin (CMD), to send commands to the eMMC
- The data pins (D[0-7]), for data transfer

With these, it is possible to dump the whole content of the chip, without the need of removing it from the board.

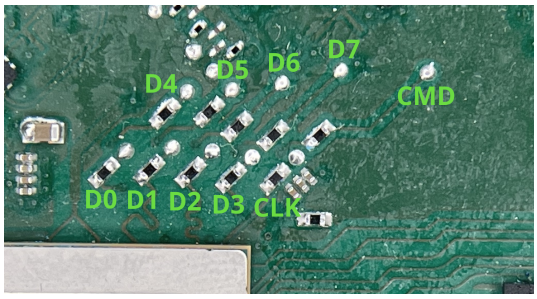


Figure 3.6: eMMC test pins

The host is using the Secure Digital Input/Output (SDIO) protocol to interact with the chip, which supports 8-bit, 4-bit and 1-bit modes [14]. This means that the only required pins needed for the firmware dump are CLK, CMD and D0[1]. The dump can be done by wiring those pins to an SD-Card adapter[2] and has to be done when the board is not powered, otherwise, the SoC will interfere with the transfer by trying to access the eMMC and sending commands to it. This eMMC supports both 3.3V and 1.8V, SD cards usually work at 3.3V, but the SoC uses the 1.8V mode, so it could be dangerous to use 3.3V, even if the board is not powered on, since the eMMC pins will still be connected to the processor. For this reason, a voltage shifter has to be used[3]. Figure 3.7 shows the final setup.

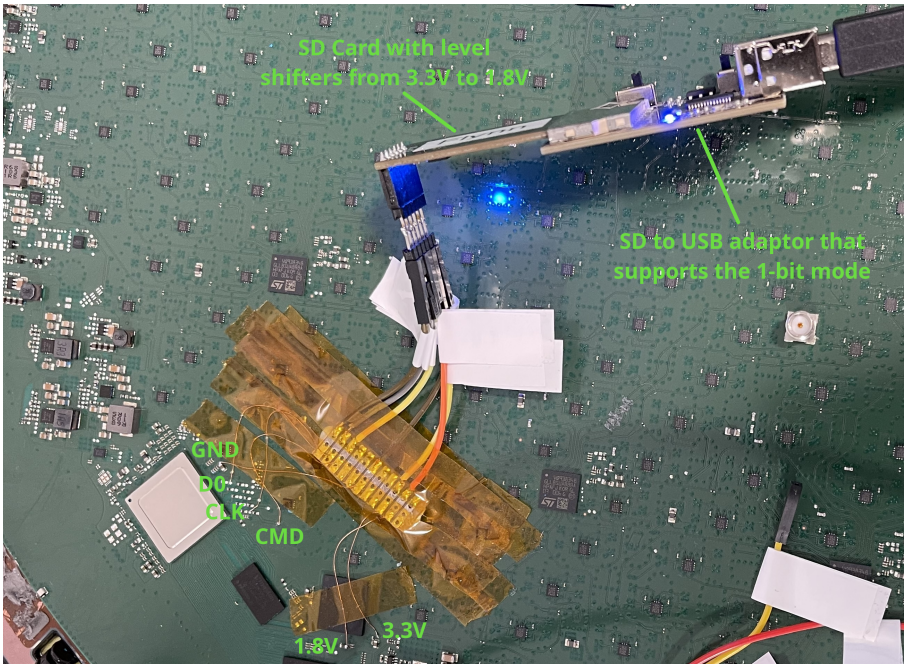


Figure 3.7: Setup for the firmware dump

After connecting the adapter to a computer, the memory can be dumped completely in a few minutes. With a first look at the content, the device does not appear to be partitioned nor contain any standard filesystem. Further manual analysis will be required, see Chapter 4.

[1] By using the 1-bit mode, data transfer will be much slower
[2] An adapter that supports the 1-bit mode
[3] The eMMC needs to be powered with 3.3V, but control and data pins need to use 1.8V

4 Firmware analysis

After dumping the whole content of the eMMC chip, the persistent memory of the dish, the image does not seem to contain any standard filesystem, and by using standard extraction tools such as `binwalk`, some of the content has been identified, with a lot of false positives. Here is part of the output:

Listing 4.1: `binwalk` output

```
HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0x644E08        SHA256 hash constants, little endian
0x67C2C4        Unix path: /usr/sbin/sxruntime_start [...]
[...]
0xF31000        romfs filesystem, version 1 size: 736 bytes, named "sxverity"
0xF51000        romfs filesystem, version 1 size: 736 bytes, named "sxverity"
0x1000007       Flattened device tree, size: 13668532 bytes, version: 17
0x147C198       Flattened device tree, size: 57661 bytes, version: 17
[...]
0x8000000       LUKS_MAGIC sha256
[...]
```

From this output, it looks like the image could contain a Unix-like filesystem since there is a reference to a Unix path, some filesystems named "sxverity" that could be something like Linux's "dm-verity", some Flattened device trees (FDTs) that are descriptions of the hardware that will be used by the operating system to know which hardware is available and where/how it can find peripherals. Finally, there are also some LUKS (Linux Unified Key Setup) encrypted partitions.

But this is not all, there are still a lot of missing pieces in the image that have to be identified and analyzed. Luckily, SpaceX is using some open-source projects that come with the GPL license, thus they need to publish their modifications to such projects for compliance. One of these is U-Boot, which is the BL33 stage of the boot chain, and from its code [31], we can get a lot of useful information on how the memory is structured and partitioned.

Thanks to this we can now split the whole image into small partitions and analyze each of them separately. In Appendix A.1 you can find a Python script that takes the whole dump and splits it into different partitions.

From this, we can also notice that almost every partition is present multiple times (e.g. `linux a/b`). This is probably because of the software update procedure, which will overwrite the partition that is not currently being used so that in the case of an error, there will still be the original partition that is known to be "correct".

Listing 4.2: include/configs/spacex_catson_boot.h

```
    +-----------------+ 0x0000_0000
    | bootfip0 (1 MB) |
    +-----------------+ 0x0010_0000
    [...]
    +-----------------+ 0x0060_0000
    | fip  a.0 (1 MB) |
    +-----------------+ 0x0070_0000
    [...]
    +-----------------+ 0x0100_0000
    | linux a (32 MB) |
    +-----------------+ 0x0300_0000
    | linux b (32 MB) |
    +-----------------+ 0x0500_0000
    [...]
```

4.1 Partitions

Here is an overview of each partition that we identified inside the dumped image, some of them were empty (full of `0xFF`) probably because they were either used in an older version or they are only used on development hardware.

Bootfip [0/1/2/3]

This section is replicated four times, which means that there can be four different versions of what's inside at the same time. Tools such as `binwalk` weren't able to find results on this partition, and from manual analysis, of the available open-source code and the image itself, it became clear that the partition contains a FIP (Firmware Image Package) image.

This type of image is used by ARM Trusted Firmware-A (ARM TF-A), which is a reference implementation of the secure world initialization and boot chain for Armv7-A, Armv8-A and Armv9-A architectures [1]. ARM TF-A also includes `fiptool`, a tool used to pack and **unpack** FIP images.

Listing 4.3: `fiptool info bootfip0`

```
Trusted Boot Firmware BL2:
    offset=0x118, size=0xF178,
    cmdline="--tb-fw",
    sha256=53a34633e3a9dbb89998bd43e01edf34006462df6005f000af439452eb1abce7
Trusted Boot Firmware BL2 certificate:
    offset=0x88, size=0x90,
    cmdline="--tb-fw-cert",
    sha256=e08bf3277c9761b64fd21907a59e12a0e7e56d567ee4ef90011b1592f00a345c
```

The image contains a boot image (BL2) and its certificate, which can be extracted by directly using the offset and size of each element or by using `fiptool unpack /path/to/image`. This will be analyzed further in Chapter 5.

fip [a.0/a.1/b.0/b.1]

This section, as the name suggests, also contains a FIP image.

Listing 4.4: `fiptool info fip_a.0`

```
Trusted Boot Firmware BL2: [...]
EL3 Runtime Firmware BL31: [...]
Non-Trusted Firmware BL33: [...]

Trusted key certificate: [...]
Trusted Boot Firmware BL2 certificate: [...]
Non-Trusted Firmware key certificate: [...]
Non-Trusted Firmware content certificate: [...]
SoC Firmware key certificate: [...]
SoC Firmware content certificate: [...]
```

It contains 3 other bootloader images, that will be loaded and executed by the BL2 image in the previous section. This will be analyzed further in Chapter 5 as well.

Linux [a/b]

This partition, as the name suggests, should contain the full-fledged Operating System that will run in the Normal World, but the image starts with what looks like a magic number, but not a known one:

Listing 4.5: FIT magic bytes

```
$ dd if=linux_a bs=1 count=7 status=none
SXECCv1
```

By looking into the available open-source code from SpaceX, we were able to identify the format of the image, which is a customized ECC-protected (Error Protecting Code) image. Here is the code from SpaceX's uBoot bootloader [31]:

Listing 4.6: **include/spacex/ecc.h**

```
#define ECC_BLOCK_SIZE    255
#define ECC_MD5_LEN   16
#define ECC_EXTENSION   "ecc"
#define ECC_FILE_MAGIC    "SXECCv"
#define ECC_FILE_VERSION  '1'
#define ECC_FILE_MAGIC_LEN  (sizeof(ECC_FILE_MAGIC) - 1)
#define ECC_FILE_FOOTER_LEN sizeof(file_footer_t)
#define ECC_DAT_SIZE    (ECC_BLOCK_SIZE - NPAR - 1)

#define ECC_BLOCK_TYPE_DATA '*'
#define ECC_BLOCK_TYPE_LAST '$'
#define ECC_BLOCK_TYPE_FOOTER '!'
```

After extracting the actual image (removing ECC information) with a script you can find in Appendix A.2 (see Section 4.2.1), we were able to correctly identify the content of the image, which is a Flattened uImage Tree (FIT). This type of image has been created for U-Boot, to pack everything the bootloader needs to load into a single image, which is the kernel, the ramdisk and the Flattened Device Tree (FDT), possibly with multiple configurations. With the `dumpimage` tool (from U-Boot) we were able to inspect the image and its content:

Listing 4.7: **dumpimage -l linux_a**

```
FIT description: Signed dev image for catson platforms
Created:         Fri Jan  6 05:28:00 2023
  Image 0 (kernel)
    Description:  compressed kernel
    Type:         Kernel Image
    Architecture: AArch64
    OS:           Linux
    Load Address: 0x80080000
    Entry Point:  0x80080000
  [...]
  Image 14 (rev2_proto3_fdt)
    Description:  rev2 proto 3 device tree
    Type:         Flat Device tree
    Load Address: 0x8f000000
  [...]
  Image 27 (ramdisk)
    Description:  compressed ramdisk
    Type:         RAMDisk Image
    Architecture: AArch64
    OS:           Linux
    Load Address: 0xb0000000
    Entry Point:  0xb0000000
  Default Configuration: 'rev2_proto2'
  [...]
  Configuration 13 (rev2_proto3)
    Kernel:       kernel
    Init Ramdisk: ramdisk
    FDT:          rev2_proto3_fdt
  [...]
```

The FIT image contains a kernel, a ramdisk image and a bunch of FDTs, one for each hardware revision of the User Terminal, Transceiver and other devices (utdev, mmut, mini, hp). All the different configurations share the same kernel and ramdisk image.

Kernel

The Kernel image is `Linux kernel ARM64 boot executable Image, little-endian, 4K pages`

Listing 4.8: `strings kernel | grep version`

```
Linux version 5.15.55-rt48-g29cea1731348 (buildroot@buildroot)
(aarch64-ct_aarch64-linux-musl-gcc
(crosstool-NG 1.24.0.11-6c073bf - 10ec388) 8.3.0,
GNU ld (crosstool-NG 1.24.0.11-6c073bf - 10ec388) 2.32)
#1 SMP PREEMPT_RT Fri Jan 6 04:28:00 UTC 2023
```

It's a customized version of Linux which, thanks to its license, is available on SpaceX's GitHub page [16]. The last available source code is `sx_2022_04_29` which is a little bit older than the one we find in the dish. Furthermore, the open-sourced code only contains modifications made to the kernel code, but not new device drivers or runtime binaries.

By using the `extract-ikconfig` script from Linux, it is possible to extract the build configuration of Linux from the compiled kernel image. This will be useful in Chapter 8 when we'll have to compile a kernel that will work with the emulator engine and with SpaceX's software.

Ramdisk

The `ramdisk` is usually the filesystem that is initially mounted in "RAM", during the Linux startup process, that can be used by the kernel to load external modules. In this case, the ramdisk is the root filesystem and will live throughout the whole execution of the kernel.

Listing 4.9: `file ramdisk`

```
ramdisk: ASCII cpio archive (SVR4 with no CRC)
```

It comes in an uncompressed CPIO archive, which can be easily extracted with the `cpio` utility. The content of this filesystem is the standard structure you expect to find in a Linux root filesystem, with BusyBox instead of the usual GNU toolset. From this, we can already understand some details on how the runtime works (network configuration, mount points, etc.), but this will be discussed in Chapter 7.

FDT

The Flattened Device Tree is a data structure holding information about the hardware, such as the number and type of CPUs, size of RAM, busses and bridges, peripheral device connections, interrupt controllers and IRQ line connections, GPIO pins, and more [2]. This is used, on different levels, by the bootloaders and the kernel to know how to interact with the hardware it is running on and how to initialize all the components of the machine. This is why every little modification in the hardware needs a separate Device Tree. Some useful information about hardware and peripherals can be found by looking at the FDT of our hardware revision. Here are some small parts of it:

The "header" contains the name of the device and the compatible architectures:

Listing 4.10: FDT header

```
compatible = "st,gllcff";
model = "spacex_satellite_user_terminal";
sx-dts-commit = "29cea17313485caecba83aec7b5a97e9ef7cefda";
sx-dts-branch = "ssh://git@stash:7999/fswplat/linux.git master";
sx-dts-build = "e98c62a7d269cc897a63f485c84344e8893bfe59";
```

A section containing information about the main CPU and its caches:

Listing 4.11: CPU and cache

```
cpus {

    cpu@0 {
        compatible = "arm,cortex-a53\0arm,armv8";
        device_type = "cpu";
        reg = <0x00 0x00>;
        next-level-cache = <0x4b>;
    };

    // other three CPUs...

    l2-cache0 {
        compatible = "cache";
        phandle = <0x4b>;
    };
};
```

A section called **security**, which holds some fixed public keys that are used for validation:

Listing 4.12: Security section

```
security {
    dm-verity-pubkey = <REDACTED>;
    dm-verity-version-info = <REDACTED>;
    dm-verity-tftp = <REDACTED>;
    [...]
};
```

A section containing mappings of GPIO pins:

Listing 4.13: GPIO mappings

```
gpios {
    ant_power = <0x14 0x00 0x00>;
    dbf_reset = <0x0e 0x06 0x00>;
    gps_boot = <0x0b 0x04 0x01>;
    gps_reset = <0x0f 0x04 0x01>;
    self_reset = <0x10 0x00 0x01>;
    [...]
};
```

sx [a/b]

This section contains the runtime of the User Terminal, but it comes in yet another custom format. The magic bytes of this image are "sxverity" which reminds "dm-verity", a Linux feature to mount devices whose content is digitally signed and verified upon access [4]. In fact, by further analyzing the content of this image and the `sxverity` binary that was found in the ramdisk, we confirmed this assumption. The format of this image and the mounting/verification process will be discussed in section 4.2.2, for now, we just need to know that the actual content of the image can be found at offset 0x1000, in the form of a `rom1fs` filesystem.

After extracting the content of the filesystem, we found all the binaries that will handle the runtime of the dish and some configuration files used by them. By looking into the initialization scripts found in the ramdisk, we also know that this partition will be mounted at `/sx/local/runtime`. This will be discussed further in Chapter 7.

EDR & DishConfig

These two partitions are encrypted using the Linux Unified Key Setup (LUKS) because the device needs to be able to write on them with persistence. These are the only two partitions mounted at runtime with write permissions. The first one, EDR, stands for Ethernet Data recorder and serves as storage for telemetry data regarding the traffic flowing through the device. The file `/sx/local/runtime/dat/common/file_edr` contains some rules on which data to record and how to save it. The second one has a self-explanatory name, it is used as a persistent storage for configuration files of the dish that can change often (e.g. the inclination of the antenna).

The keys used to encrypt the partitions are stored inside the CPU, in the form of eFuses (electronic fuses). An eFuse works exactly like a fuse, once it has been blown, its state cannot be changed, thus it can be used as a write-only-once bit cell. With multiple eFuses you can store more information, such as an encryption key.

eFuses are exposed to the userland by using device drivers so that the userland applications can interact with them (read and blow) by simply reading or writing to files under the folder "/sys/devices/platform/soc/22400000.catson_fuses/". The encryption key can be read through the file `ekey` (`ckey` for dish config), and it is written the first time the device boots when the encrypted partition is initialized. This is done by writing a "1" to the file `burn_ekey`, whose handler can be found in the open-sourced version of Linux by SpaceX.

Listing 4.14: linux-sx/drivers/misc/sx/sx_catson_fuse.c

```
arm_smccc_smc(CATSON_SIP_SVC_BLOW_EKEY, 0, 0, 0, 0, 0, 0, 0, &res);
```

The driver simply calls the secure monitor, which will handle the secure monitor call (SMC) and physically burn the eFuses using the hardware random number generator. The source code of the secure monitor is not available to us, but we know it is based on ARM TF-A, and it's using the ARM SiP (Silicon Provider) service.

version [a/b]

This section contains a sxverity image as well, by using the same technique we used for the `sx` partition, we were able to extract the content. The internal filesystem only contains one file, `version_info`:

Listing 4.15: version_info

```
[metadata]
gauntlet_build_id c36a30b4-93ac-4a14-b663-0622ef7ed944.uterm.release
generation_number 1674091226

[partitions]
Image.fit.ecc 6784c5aa76887cdc950b6a5e723c7095149713f0dd8897460721b93e3a00c76d 15700403
SPACEX_CATSON_TRANSCEIVERboot.bin
    b4614fd5d697ba878e0759bd831ff03c5837a060ae2c394e8b32389bd256f32b 579344
SPACEX_CATSON_UTERM_EMMCboot.bin 62
    f005932196f61210c1bc5cd4099d654354a05a54c48bc4d687019822ae85e5 590808
SPACEX_CATSON_UTERMboot.bin
    c6b600b3d4c11f90e48c4adcf35f9602f851c2a08a083b409ccafa251c0339e4 579344
runtime.sxv b90409e252c916a56da2f25cbbbab5c8758f66f8d3ba175a5f7ced610ec3567c 20419584
```

The file contains information about the installed version of some software components, such as the U-Boot bootloader (`SPACEX_CATSON_UTERM_EMMCboot.bin`), the FIT image containing the kernel, ramdisk and FDTs (`Image.fit.ecc`) and the runtime image (`runtime.sxv`)

The other partitions were either found to be empty in our dump (full of `0xFF`) or they are simply not worth mentioning.

4.2 Data Integrity

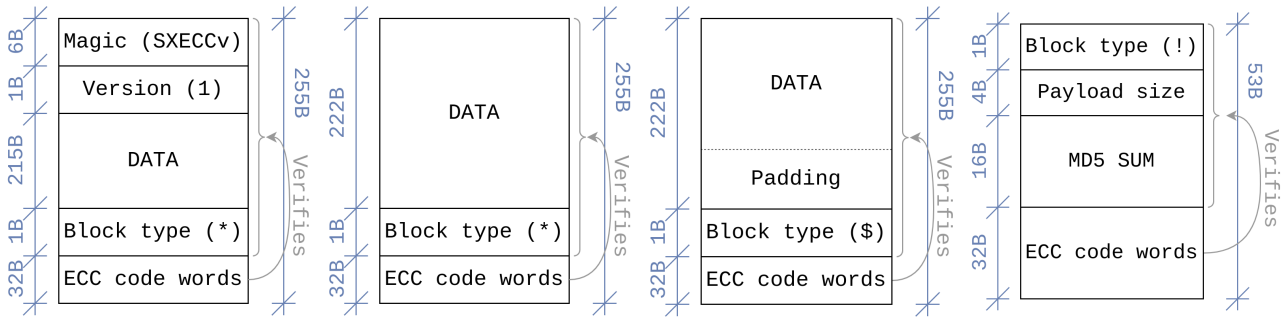
As we have seen from a brief analysis of the content of the eMMC dump, SpaceX is using some custom-made mechanisms for data integrity, along with the standard ones already included in ARM TF-A and U-Boot. A detailed analysis of the chain of trust can be found in Chapter 5, here is an overview of the custom-made components.

4.2.1 ECC

The Error Correcting code mechanism is only used in the FIT image and only provides data integrity, without considering authenticity. This means that, in theory, if one wants to tamper with some ECC-protected components of the dish they can do that by providing a correctly formatted version of it, so by implementing themselves the `ecc` procedure (or by using the binary that can be found in the ramdisk). But for the FIT image, this is not possible because authenticity is also checked by the last bootloader stage (see Chapter 5). So this is just used to prevent errors in the eMMC storage.

This works similarly to its original ancestor, the ECC RAM, which embeds some additional data in between the actual content of the memory - originally Hamming codes - that are computed as a function of the data they are "protecting". Then, when some data is accessed, Hamming codes are recomputed and if they do not correspond to the ones saved in memory, an error occurred, and depending on how many bits have been mistakenly flipped, the error can be corrected. This version of ECC uses Reed-Solomon error correction (instead of Hamming codes) and a final hash (i.e. MD5) to check the integrity of the whole file that is being decoded.

In Appendix A.2 you can find a simple Python script that simply strips out ECC information from a file, without checking the correctness of the data, that we used to be able to unpack the FIT image. But inside the ramdisk, there is a binary (`unecc`) that does the same thing also checking and trying to correct possible errors.



(a) Header block (b) Data block (c) Last data block (d) Footer block

Figure 4.1: ECC blocks

The content of ECC-protected files is organized into blocks of different types, Figure 4.1 shows how each block is structured. The file starts with a header block (a), which contains the magic number and the version of the protocol, along with some data and the corresponding control codes. Then, there can be zero or more data blocks (b) containing just data and control codes. A last data block (c), which is recognized by its block type field (`$`, instead of `*`), marks the end of the payload, some padding is added here if needed. Finally, the footer block (d) contains the size of the payload (needed to know the number of padding bytes), an MD5 checksum of the whole payload and, of course, ECC code words for the footer block itself.

4.2.2 sxverity

`sxverity` is a custom wrapper for the device-mapper-verity (dm-verity) kernel feature, which provides transparent integrity checking of block devices [8]. The source code of the `sxverity` binary is not publicly available, thus we had to reverse the compiled binary to understand the internals. This provides both data integrity and authenticity thanks to a signature check that verifies the whole content of the device. `sxverity` internally uses the dm-verity kernel feature, by directly interacting with

it through the `/dev/mapper/control` device.

SpaceX only tackled the verification of the root hash, everything underneath, which is handled by the kernel, has not been reimplemented, so here is a quick overview of the internals of dm-verity.

dm-verity

The dm-verity feature lets you look at a block device, the underlying storage layer of the file system, and determine if it matches its expected configuration. It does this using a cryptographic hash tree. For every block (typically 4k), there is a SHA256 hash. Because the hash values are stored in a tree of pages, only the top-level "root" hash must be trusted to verify the rest of the tree. The ability to modify any of the blocks would be equivalent to breaking the cryptographic hash. See the following diagram for a depiction of this structure [8].

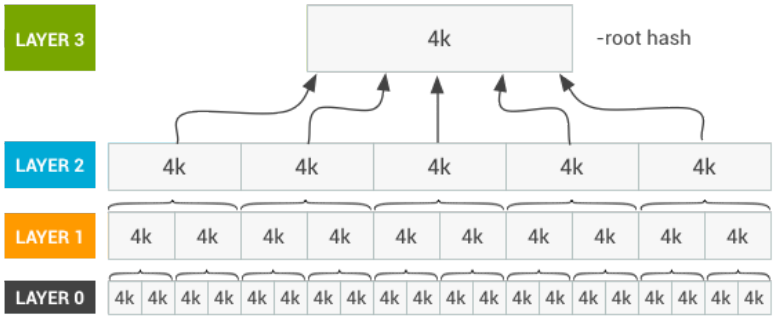


Figure 4.2: dm-verity hash table [8]

This is very convenient because the kernel doesn't need to compute the hash for the whole content of the device to assess its integrity. When a block is read, just the hash of that block will be computed and compared with what there is in the hash tree, all the way to the root hash, which is the root of trust in this scheme.

As we have seen in Section 4.1, sxverity is used to verify some of the partitions that reside in the persistent memory, this is to prevent persistent exploits. But as we'll see in Section 10.1, it is also used to verify software updates for the dish. Thus, it is a critical component for the overall security of the device.

In Figure 4.3 you can see the structure of an sxverity image. It is composed of a header[1], which contains:

- The magic bytes `"sxverity"`
- The version and some flags indicating which algorithms have been used for signing and hashing
- The root hash, which indirectly covers the whole payload (through the hash tree)
- The public key that has been used to sign the image
- The signature of all the fields above, using an elliptic curve (ED25519)

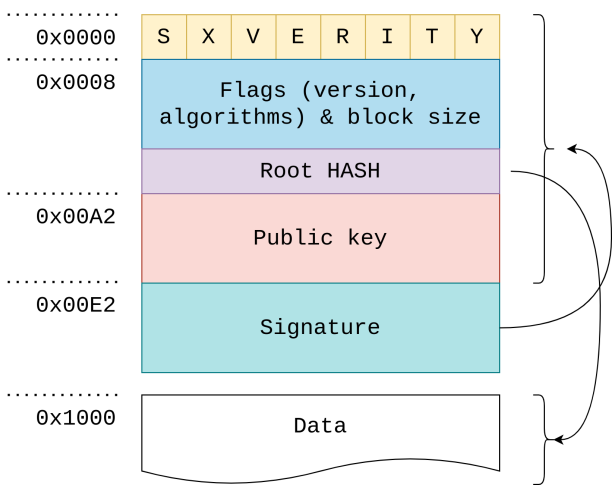


Figure 4.3: sxverity image structure

[1]The header is repeated 4 times, possibly signed with different public keys

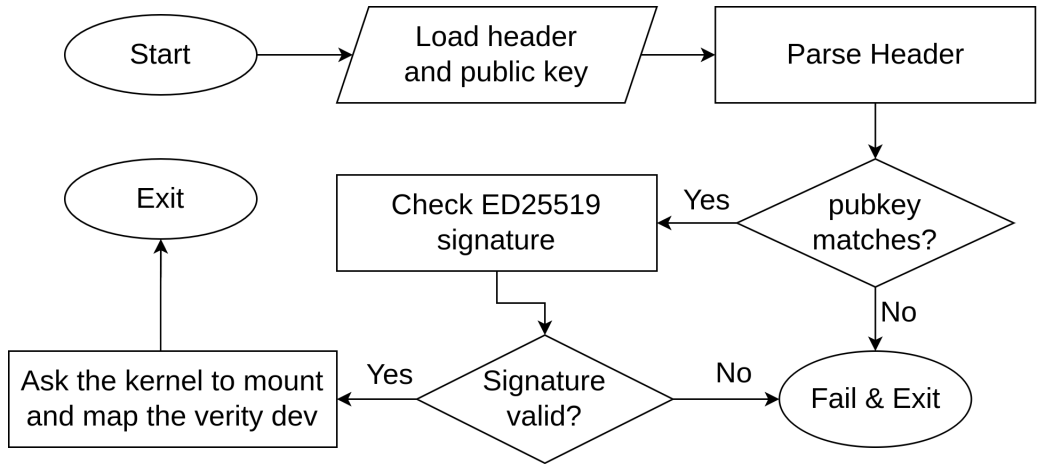


Figure 4.4: Flow diagram of the `sxverity` verification process

The verification and mount process made by the `sxverity` binary boils down to the procedure shown in Figure 4.4. This will be discussed further in Section 10.1 when we'll talk about fuzzing the software update process, which is internally using `sxverity`.

5 Boot procedure

With the same technique used in Chapter 4, we can also **write** in the persistent storage, but a modification made to the software stored in that memory, at any level, will be detected and will cause the device to fail. This is thanks to a chain of trust, that protects the integrity of what is being executed from the very first bootloader to userland processes. The integrity of the components is checked at multiple stages and in multiple ways, in this chapter, we will see an overview of this process, starting from the bootloader to the full-fledged operating system running at EL1 (i.e. Linux).

5.1 Bootloader chain

As for every cryptographic system, there has to be a Root of Trust (RoT), i.e. a component of the system that can be trusted and upon which is based the security of the whole system and from which the Chain of Trust (CoT) is built. If the RoT is compromised, the whole chain (i.e. the system) is compromised as well. In this case, the RoT is the internal read-only memory (ROM) of the micro-controller, which contains the first bootloader stage (BL1) and the public key with which the next bootloader stage is signed.

Early boot stages are using ARM TF-A [1] which unfortunately doesn't come with a GNU-like license, thus modifications made by SpaceX are not publicly available. By reversing the compiled code found in the eMMC and comparing it with the "base" version of ARM TF-A, we were able to analyze it easily. Then, the last bootloader stage is U-Boot [31] which will load and boot the Linux kernel, which are both using an open source license that forced SpaceX to publish their changes. Here is a detailed description of each stage

- **BL1**, part of ARM TF-A and stored in the internal ROM, is executed at exception level 3 (EL3), in an internal (safe) RAM. It will install its exception handler that will function as a temporary secure monitor and will then load BL20 from the eMMC, verify its integrity using the key stored in the ROM and jump to its entry point.
- **BL20**, part of ARM TF-A and stored in the `bootfip` section of the eMMC, is executed in

secure exception level 1 (EL1S). It is in charge of performing some architectural initialization and loading the next bootloader stage (BL21), verifying its integrity and executing it.

- **BL21**, part of ARM TF-A and stored in the fip section of the eMMC, is executed at EL1S. This stage is in charge of loading and verifying the integrity of BL31 and BL33. Once this is done, it will notify the secure monitor, which at the moment is BL1, that will execute BL31.

Listing 5.1: Secure Monitor call

```
bl_ep_info = load_BL31_and_BL33();
SMC_call(4, bl_ep_info, 0, 0, 0, 0, 0, 0);
// From ARM TF-A source code:
// smc(BL1_SMC_RUN_IMAGE, (unsigned long)next_bl_ep_info, 0, 0, 0, 0, 0, 0);
```

- **BL31**, usually called "trusted firmware" is part of ARM TF-A and stored in the fip section of the eMMC, is executed at EL3. It will install its exception handler replacing the one installed by BL1, becoming the new secure monitor which will live throughout the whole execution of the kernel. After this, it will execute BL31, which has been loaded and verified by BL21.

- **BL33**, usually called "untrusted firmware" is based on U-Boot and stored in the fip section of the eMMC, is the first stage that is executed in the normal world, more specifically at exception level 2 (EL2). This stage is in charge of loading the FDT from storage, performing additional architectural initializations, loading the Linux kernel and its ramdisk and finally starting the kernel in EL1 (normal world).

- **Linux**, which isn't a bootloader stage, runs at EL1 in the normal world and it's stored in the `linux` section of the eMMC. During the boot process, it will load and check the runtime package, using `sxverity`, which is stored in the `sx` section of the eMMC and contains all the runtime binaries that will be executed at Exception Level 0 (EL0) in the normal world. At this point, the situation is the following: BL31 is running at EL3 as a secure monitor, and the Linux kernel is running at EL1 in the normal world and will handle processes running at EL0. There isn't any Trusted operating system running at secure EL1, thus no trusted applications are running at secure EL0.

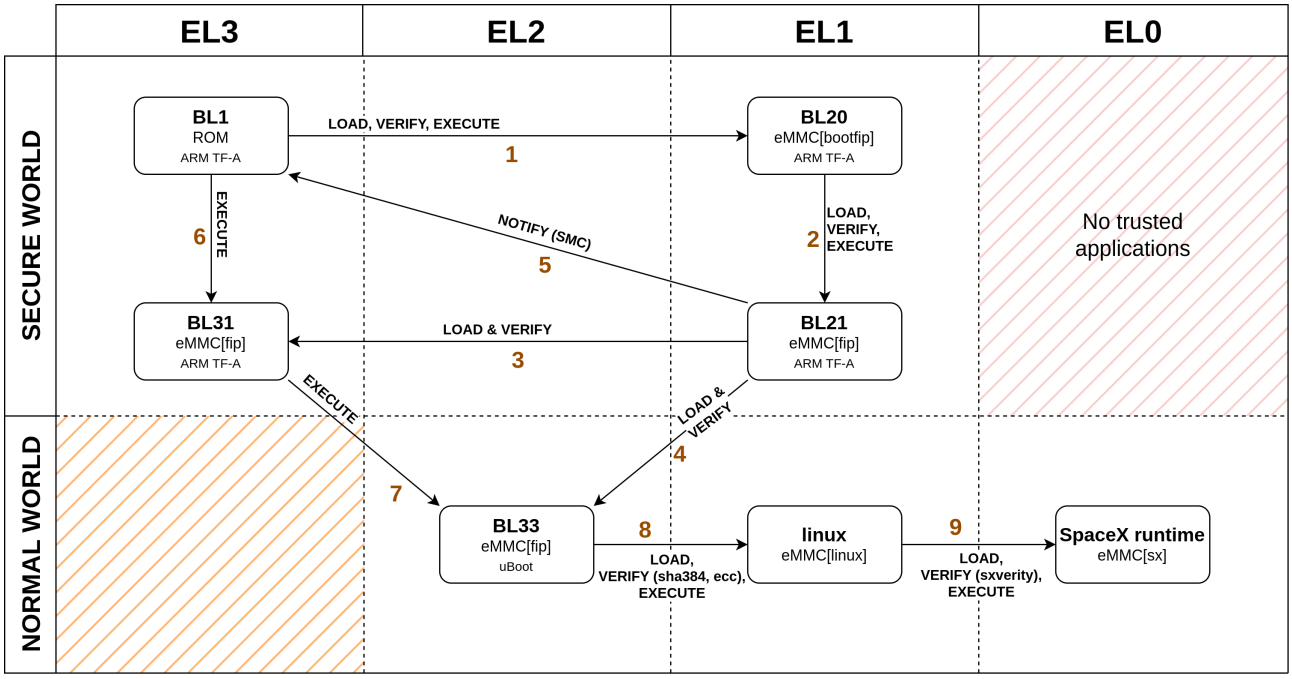


Figure 5.1: Boot chain

Figure 5.1 shows this process graphically. From this, it's clear that tampering with any component along the chain would be detected by the previous stage and the boot process would be interrupted. Thanks to this it is very difficult for an attacker to execute code on the device, even when having physical access, and it is even more difficult to install persistent code, i.e. code that survives a reboot.

5.2 I/O initialization

Interacting with a black box device isn't usually an easy task, and for this device, it has been particularly difficult. Gaining root access to the devices has been a big part of my work, in Chapter 6 we will describe how we tried to do it, by using a hardware attack. This is an overview of the possible ways we could interact with the device.

5.2.1 UART

As we have seen in Chapter 3, the device is equipped with a serial port that uses the Universal Asynchronous Receiver/Transmitter (UART) protocol. This type of port is usually used for debugging purposes, but by connecting to it we didn't receive any output nor we were able to send any input. It is known that previous versions of the software were printing debug messages through this interface, but a software update has disabled it [28]. This has been done by blowing an eFuse and adding a check in every boot stage. As explained in Section 4.1, eFuses are accessible in the Linux userland by interacting with a file driver. But at the bootloader level, they are accessed through Direct Memory Access (DMA) at addresses starting from `0x22400000`. Each bootloader stage will enable or disable the UART interface by checking a particular eFuse, at address `0x22400008`. In Listing 5.2 are shown some pieces of the code from BL20 that perform these operations.

Listing 5.2: BL20 io setup

```
void init_uart(unsigned long address, int clock, int baud_rate);
void arm_io_setup() {
  //...
  if (read_uart_fuse() != 0) {
    enable_uart();
  }
  //...
  init_uart(0x8850000, clock, 0x1c200);
  //...
}

uint read_uart_fuse() {
  return *((unsigned int *)0x22400008) >> 0x1f ^ 1;
}

void enable_uart() {
  *((unsigned int *)0x8200000) = *((unsigned int *)0x8200000) & 0xfffffff8 | 1;
  *((unsigned int *)0x82000a0) = *((unsigned int *)0x82000a0) | 1;
}
```

5.2.2 Development hardware

SpaceX developers have to be able to debug the hardware and the software running on it, and they do it by using development hardware. The software knows it is running on development or production hardware, again, by reading an eFuse which is only blown on production hardware. Listing 5.3 shows a script used by programs running in the OS to check if the device is production- or hardware-fused. The same check is made in boot stages by directly reading from memory (DMA).

Listing 5.3: /bin/is_production_hardware

```
#!/bin/sh

# is_production_build.sh
# Wrapper to determine if Catson board is production fused.
```

```
set -e

fuseFile="/sys/devices/platform/soc/22400000.catson_fuses/fuse_map"

if [ -e $fuseFile ]; then
    fuseMap=$(cat $fuseFile)
    if [ $(( fuseMap & 0x80 )) -eq 0 ]; then
        echo "1"
    else
        echo "0"
    fi
else
    # Default to the more secure option
    echo "1"
fi
```

Among all the differences between development and production software, the most interesting to us is shown in Listing 5.4, in which the script sets a root password. The hashed password is `tSXNnW65X1Er` (DES Hash), which corresponds to the password `falcon`. When the root password is set, it is possible to log in as root through the serial console (if enabled) or through SSH.

Listing 5.4: /bin/setup_console

```
#!/bin/sh

# start_console
# Wrapper to launch serial console.

set -e

# Report if development login is enabled over the serial console.
echo -n "Development login enabled: "
if [ "$(is_production_hardware)" -eq 0 ]; then
    echo "yes"
    sed -i -e 's/^\(root:[^:]*\)/root:tSXNnW65X1Er./' /etc/shadow
else
    echo "no"
fi
```

6 Fault injection attack

As we have seen in Section 5.2, it is not easy to interact with production-fused hardware, let alone gain root access to it. This chapter describes an attack that has been developed by a research group at KU Leuven, which aims to execute arbitrary code from the early bootloader stages, by trying to skip a signature check, to enable UART output and SSH access by tricking the device into thinking it is not production-fused.

6.1 Introduction to the attack technique

Every hardware component is using some physical properties to store, transfer or manipulate data. These physical properties are influenced by the environment they are working in and they are usually designed to resist normal circumstances (i.e. temperature, magnetic fields, etc.). But even under normal circumstances, devices can experience some errors, thus protections have been implemented both at the hardware level (e.g. CRC and shields in memories, voltage regulators, etc.) and software level (e.g. error-correction mechanisms, redundant checks, etc.). Fault injection is an attack technique in which the attacker stresses the environment in which the component under attack is working on purpose, trying to cause the component to fail or to behave in unexpected ways. There are multiple ways to perform hardware fault injection, such as:

- Power/Clock Glitch Fault injection (PGFI), in which you inject a fault by altering the clock's frequency or power supply in the corresponding pin of the chip.
- Light/Laser Fault injection (LFI), in which you inject a fault by inducing a light pulse on the back side or front side surface of the silicon dice.
- Electromagnetic Fault injection (EMFI), in which you inject a fault by applying an electromagnetic pulse near the surface of the chip.
- Body Biased Fault injection (BBFI), in which you inject a fault by applying a forward or reverse bias at hundred of mV to the substrate of the chip.

Each one of them requires different access to different elements of the chip, some of them are more precise and some of them are more difficult to execute [6].

6.2 Attack strategy

As shown in Section 5.1, every software component is verified before being executed, by constructing a Chain of Trust (CoT), as shown in Figure 6.1.

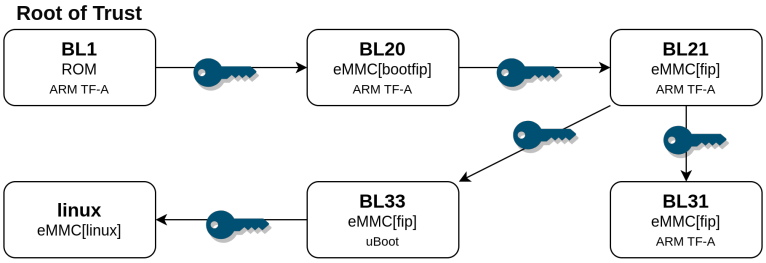


Figure 6.1: Chain of Trust

Since we cannot tamper with the Root of Trust (i.e. BL1), the Chain of Trust has to be broken to be able to execute some arbitrary code. This can be done by trying to glitch the very first signature check in BL1 and by patching any other successive check, as shown in Figure 6.2. The chosen attack by Lennert was Power glitch fault injection (or Voltage fault injection), which consists in injecting a different Voltage (higher or lower) to the chip exactly when the signature is being checked.

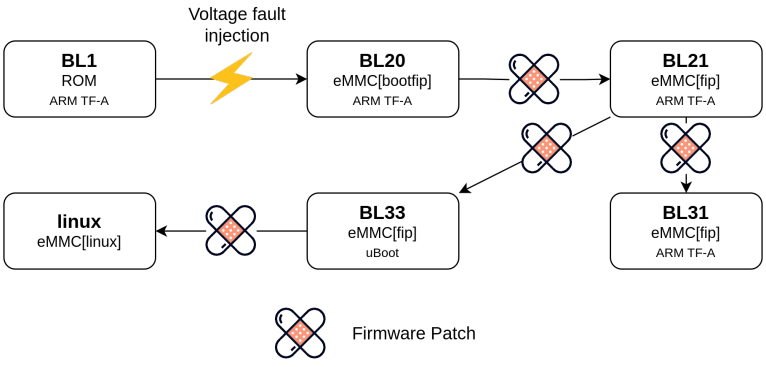


Figure 6.2: Broken Chain of Trust

The injection must be extremely precise to skip the right instruction(s), so we have to find a way to synchronize with the device, and doing this by hand would be impossible. In Sections 6.4 and 6.5 we will show how to solve this problem.

6.3 Firmware patching

In the previous section, we mentioned firmware patching, this section will show what exactly has to be patched to perform the attack. First of all, we need to understand what we want to change in the

firmware after which we will know what else needs to be patched in order not to break any signature check. In our case, we want the operating system to think it is a development device (1), so a patch has to be made in the `FIT` section of the eMMC, and since this section is almost at the end of the chain of trust, we need to patch every previous signature check (3), as shown in Figure 6.2. Then we also want to re-enable UART output (2), to have some feedback on the outcome of the attack, otherwise, we would need to wait for the main OS to boot and try to connect through SSH just to know if the attack worked. Following is a description of every patch mentioned above.

1. The first patch is the easiest one since every process in the Linux userland is using the same bash script to check whether the hardware is production fused, we just need to edit that script. It can be found in the ramdisk, at `/bin/is_production_hardware` (shown in Listing 5.3), and can be patched by adding the following lines at the beginning of it.

   Listing 6.1: Patch for is_production_hardware script

   ```
   echo "0"
   exit 0
   ```

   Then, the ramdisk needs to be repacked using `cpio`, the whole `FIT` image needs to be repacked using the tools provided by U-Boot and finally, `ECC` data has to be added, by using the `ecc` binary that can be found in the ramdisk itself.

2. The second patch, that enables UART output, involves every boot stage since the UART interface is re-initialized in each stage. As seen in Listing 5.2, the check is made by calling the function `read_uart_fuse`. We can either patch the function to always return a positive value or patch the function call. In Figure 6.3 you can see the patch made in BL20, this has to be done for every bootloader stage. Then, each stage has to be packed using `fiptool`.

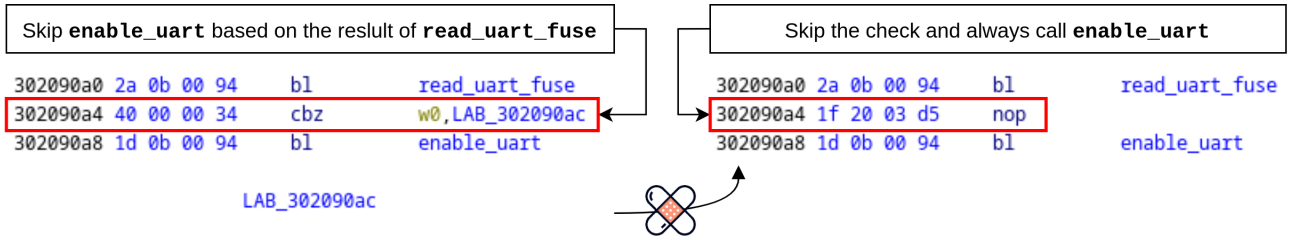


Figure 6.3: Patched UART initialization in BL20 image

3. Finally, the most important set of patches are the ones for skipping the signature verification of successive stages. By reversing the bootloader images and comparing them with ARM TF-A source code, we identified where the signature verifications happen. It is enough to skip them completely, as shown in Figure 6.4.

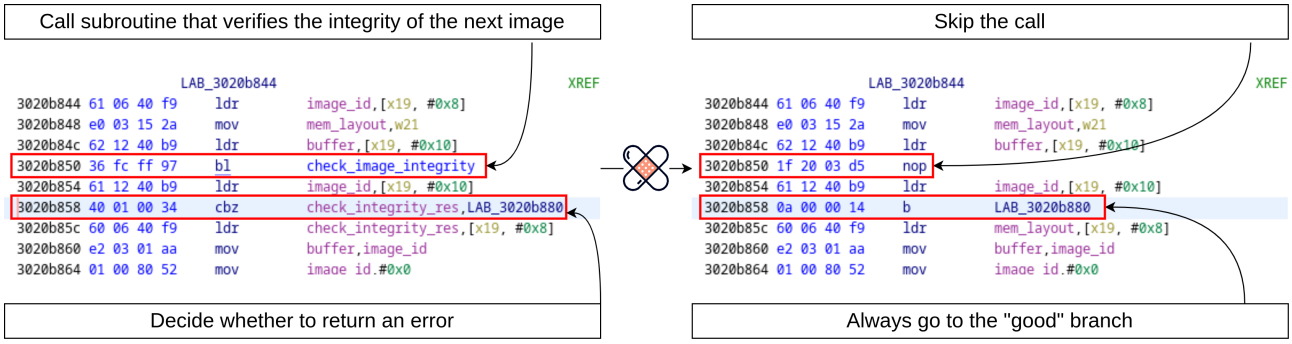


Figure 6.4: Patched signature verification in BL20 image

For the attack to be successful it is necessary to understand exactly how and when the integrity of BL20 is verified. Since we don't have the compiled BL1, we need to rely on the available source code from ARM TF-A and assume that the check is implemented in the same way as the other bootloader stages. Every image comes with a certificate, containing the hash of the image itself and a signature for the hash. The check happens in two steps:

1. First the signature is checked for the provided hash;
2. Then the hash is recomputed for the current image and is compared with the provided one.

The glitch can happen in one of the two steps, as long as the other step is valid. So, we either:

1. can patch the hash of BL20 with the newly computed one (including our patches) and glitch on the signature check;
2. or we can leave the original hash which will make the signature valid, and glitch on the hash comparison.

We decided to go with option number 1, in Listing 6.2 you can see the bash script we've implemented to patch the hash with a newly computed one.

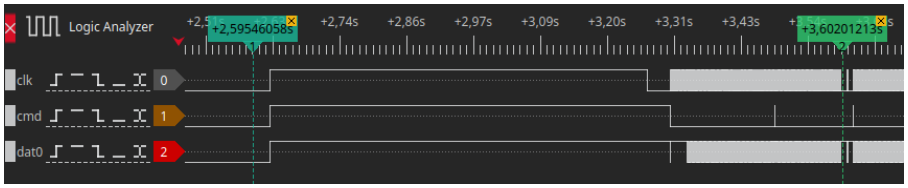
Listing 6.2: Script that updated the hash in the patched BL20 image

```
HASH_OFFSET_IN_CERTIFICATE=80
CERTIFICATE_OFFSET_IN_FIP=136
HASH_OFFSET_IN_FIP=$((HASH_OFFSET_IN_CERTIFICATE+CERTIFICATE_OFFSET_IN_FIP))
# Compute and patch the new hash
sha512sum $PATCHED_IMAGE | \
    cut -d ' ' -f 1 | \
    xxd -r -p - | \
    dd of=$1 bs=1 seek=$HASH_OFFSET_IN_FIP status=progress conv=notrunc
```

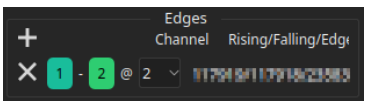
After applying these patches and repacking the whole firmware image, the technique discussed in Section 3.2 can be used to write the updated image in the persistent memory of the device.

6.4 Attack timing

The fault injection has to be extremely precise for the attack to work, thus it is not feasible to do it manually and/or randomly. A way to synchronize the fault injection with the device is needed and since BL1 does not produce any output on the UART interface, which would be the optimal synchronization method, we need to use something else. Since we have already soldered some jumpers on the eMMC SDIO interface, this one can be used to synchronize. More specifically, we used the `D0` (Data line 0) pin, by capturing some traces using a logic analyzer with both the original and the patched firmware. By looking at the differences in the traces, and where the device stops reading in the case of our patched firmware, we have identified where the signature check fails and by counting the rising edges of that data line, until the selected point in the trace, we had a reasonable precision in identifying when the CPU is making the check, by just passively reading from that test pin. In Figure 6.5a you can see the whole captured trace up to the point where the signature verification is performed and in Figure 6.5b is shown how to count the number of rising/falling/total edges, which has been blurred not to disclosed exact attack parameters.



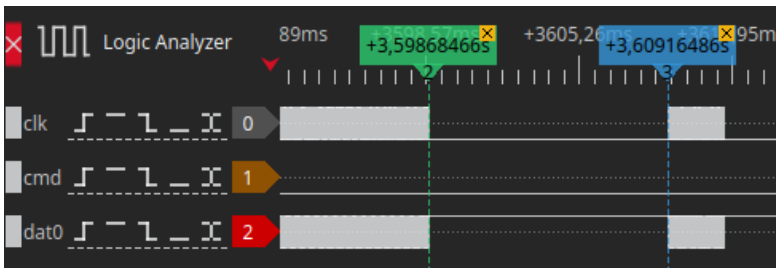
(a) Logic Analyzer trace



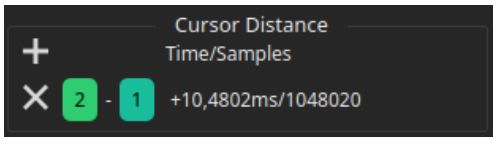
(b) Edges count

Figure 6.5: eMMC D0 capture during boot

In Figure 6.6a you can see the range between the end of the first read and the beginning of the second one, made only when the signature verification is successful, from Figure 6.6b you can see that the time between the two read operations is around 10 milliseconds, which is little enough to "brute force" (i.e. inject the glitch at random times during that time interval).



(a) Logic Analyzer trace



(b) Time range

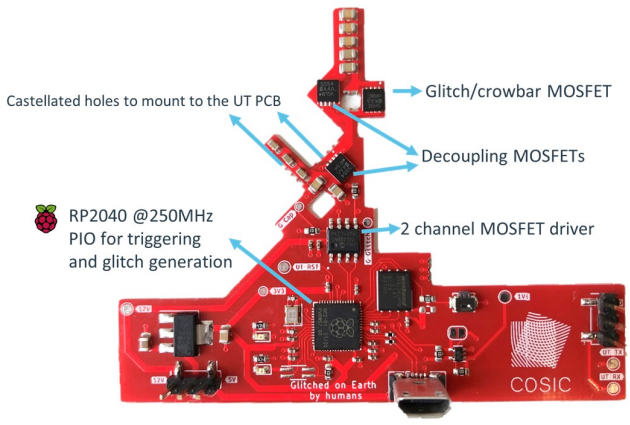
Figure 6.6: eMMC D0 capture during boot, zoomed in

6.5 Modchip

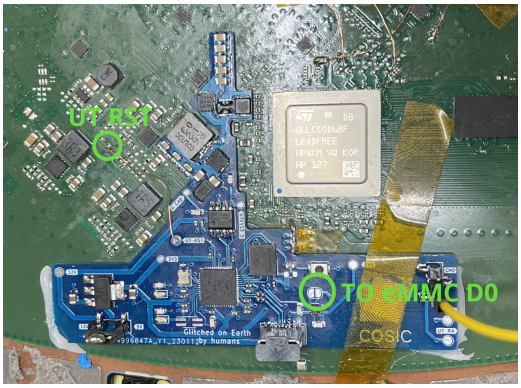
To perform the attack, we need something that:

1. Turns on the device.
2. Reads from the `D0` eMMC data line, counting rising edges until the trigger value is reached.
3. Performs the Voltage fault injection.
4. Detects whether the glitch was successful (i.e. the signature verification was successfully skipped).
5. Reboots the device and goes back to step 2 if the glitch was not successful.

Steps 1 and 5 can be achieved by setting a high or low value on the `RST` test pin of the CPU, which is shown in Figure 6.7a. Step 4 can be achieved by reading from the UART interface since we have patched the BL20 bootloader to enable the interface and print debug messages to it, so if we see something on that interface this means that the attack was successful. Step 3, the trickiest one, needs to be performed through the `Vin` pin of the CPU, but as seen in the introduction of this section, some hardware protections have been implemented to prevent errors in case of random interferences in the power supply. In this case, some decoupling capacitors have been put on this line, to stabilize the input voltage, thus we have to physically remove this protection for the attack to succeed.



(a) Modchip picture from COSIC research group [29]



(b) Our modchip mounted on the round User Terminal

Figure 6.7: Modchip designed by KU Leuven COSIC research group

Luckily, the research group at KU Leuven designed a modchip that does all this and fits perfectly on the round dish. The modchip mounts an RP2040, a microcontroller chip designed by Raspberry

Pi, and some interfaces to interact with the User Terminal and the controlling machine. It can be programmed to trigger on the D0 eMMC data line and send some pulses through the MOSFET driver, which disables the decoupling capacitors and shorts the SoC core voltage power supply to ground [29]. The original modchip is shown in Figure 6.7a. Thanks to this, we printed the PCD design and assembled our modchip, and then mounted it on the User Terminal. For more detailed instructions on how to mount it and how to assemble it, head to the GitHub page of the COSIC research group (link in references [29]). Figure 6.7b shows our modchip mounted on the dish.

6.6 Results

Unfortunately, when dealing with this kind of attack, a lot of things can go wrong, and burn some components of the board. Moreover, since the PCB schematics are not publicly available, it is very hard to find out what exactly is broken and fix it. This is what happened to us two times, the first one the device bricked while uploading the patched firmware, and the second one while performing the fault injection attack through the modchip. Due to these complications and time constraints, we were not able to get a third device so we gave up with this attack. Further dynamic testing has been performed by developing an emulated environment in which we could execute some components of the runtime, as shown in Chapter 8. Additionally, the SpaceX security team accepted us in their bug bounty program and gave us a new dish (Version 3) with root access to it. But again, due to time constraints, we weren't able to work much on that device.

7 Runtime

In this section, we will discuss what happens after the bootloader chain, starting from the Linux's init script to the processes that handle the runtime of the User Terminal.

7.1 Linux boot

U-Boot (BL33) is in charge of preparing the environment for Linux to boot, and then launching the kernel. The command line U-boot uses to launch the kernel contains some useful information, such as the path to the init script (`rdinit=/usr/sbin/sxruntime_start`) and how the eMMC is partitioned (`blkdevparts=mmcblk0:0x100000@0x00000000(BOOTFIP_0),...`), as shown in Section 4.1. Disk partitioning uses the embedded device command line partition parsing kernel option, which adds support for reading block device partition table from the kernel command line [5].

7.1.1 Init script

The init script, in Linux, is the first process that is started by the kernel and has usually a process identifier (PID) equal to 1. Its main task is to start all the other runtime processes that are needed for the system to be useful and remain in execution until the system is shut down. It will be the "oldest" ancestor of any other process and is also usually used to start, stop and configure daemons by the user, once the system is up and running. The most common init script you can find in end-user Linux distribution is `systemd`, which is a collection of tools to manage the whole runtime of the system (e.g. `systemctl`), among which there is also the init script.

SpaceX's people like to implement their software, in fact, they also have implemented their init script. `sxruntime_start` uses a custom formatted configuration file, which contains the instructions on which processes to start, how to start them and in which order.

Listing 7.1: /sx/local/runtime/dat/common/runtime

```
1 ########################################
2 # Create directory with rw permissions for all users. Copy temporary config file
3 # to new partition if tmp cfg exists.
```

```
4  #
5  # Depends on mount_dish_cfg being successful.
6  # This is so user_terminal_frontend can write to the dish_cfg partition.
7  ######################################
8  proc            /bin/sh
9  alias           make_dish_cfg_dir
10 startup_flags   wait_on_barrier barrier
11 custom_param    /sx/local/runtime/bin/dish_cfg_post_setup.sh
12 custom_param    /tmp/dish_cfg_setup_failed
13 custom_param    /mnt/dish_cfg/all
14 custom_param    /tmp/config
15
16 ######################################
17 # user terminal frontend
18 #
19 # Wait until dish config partition is set up before launching.
20 ######################################
21 proc            user_terminal_frontend
22 apparmor        user_terminal_frontend
23 start_if        $(cat /proc/device-tree/model) != 'spacex_satellite_starlink_transceiver'
24 startup_flags   wait_on_barrier
25 user            sx_packet
26 custom_param    --log_wrapped
```

Listing 7.1 shows a fragment of one of the configuration files that are used by the init script. These files are not just a list of programs to start, they are a bit more advanced, you can for example, define relationships between processes (e.g. wait for P1 before starting P2) or define conditions to be evaluated before starting a process. What follows is a detailed explanation of what this fragment means.

- The `proc` keyword defines the beginning of a process definition and takes the path of the program as an argument (lines 8, 21).
- The `alias` keyword sets an alias for the current process so that the process name is more explanatory and also creates a symbolic link to the process in `/tmp/runtime/<alias> -> /proc/<pid>` (line 9).
- With the `startup_flags` keyword it is possible to set multiple flags:
  - `barrier` sets a barrier that will be used by the `wait_on_barrier` flag.
  - `wait_on_barrier` waits for the last process marked with the `barrier` flag to finish before starting the current one.
  - `wait` simply waits for the current process to finish before continuing with the next one.
  - `no_logger` disables logging for the current process.
- The `apparmor` keyword chooses which apparmor profile to use for the current process (line 22).
- The `start_if` keyword evaluates a bash boolean expression to decide whether to start the current process or not (line 23).
- The `user` keyword will change the Linux user who is executing the binary (line 25).
- The `custom_param` will pass additional command line parameters to the process (lines 11-14, 26).

The two configuration files that are parsed by the init script can be found at `/etc/runtime_init` (ramdisk) and `/sx/local/runtime/dat/common/runtime` (runtime image). The first one handles system-level services and configurations, such as mounting partitions (e.g. the runtime), and setting up the network, the console and the logger service. The second one, instead, handles more high-level processes and configurations, such as starting all the processes contained in the runtime image and initializing encrypted devices.

7.2 System configuration

To understand what every process does and how it interacts with the underlying system and hardware, it is useful to analyze how the system is configured. This analysis is also needed for the next Chapter, in which we'll set up an emulated environment in which we'll perform some dynamic analysis and testing.

7.2.1 CPU scheduling

The init described in Section 7.1 doesn't just start processes following the mentioned configuration files, it also assigns priorities and specific CPU cores to those processes. This information is not included in the same configuration files, another one is parsed by the process which will then instruct the Linux kernel to schedule processes in specific ways. This file is also heavily documented, in Listing 7.2 you can see the header, which already contains a lot of useful information.

Listing 7.2: **/sx/local/runtime/dat/common/runtime_priorities**

```
#
# Associate processes into rules with explicit priority, scheduling policy,
# and CPU affinity.
#

############################
# System Information
############################
#
# The user terminal phased-array computers are Catson SoCs with a quad-core
# Cortex-A53.
#
# We dedicate one core to control, while leaving the other three to handle
# interrupts and auxiliary processes.
#
#   CPU 0: Control process.
#   CPU 1: Lower-MAC RX process.
#   CPU 2: PhyFW and Lower-MAC TX process.
#   CPU 3: utility core - interrupts, auxiliary processes, miscellaneous
```

After this, the file contains a detailed description of how to use/read this configuration file, and finally the set of rules that will be parsed by the init script, which will then instruct the kernel to schedule each process on a particular core, using a particular scheduling algorithm, with a predefined priority and so on. As an example, Listing 7.3 shows how priorities are assigned, the format of rules and, as an example, the rule for the "control" process.

Listing 7.3: **/sx/local/runtime/dat/common/runtime_priorities**

```
# Process priorities should be allocated as follows:
#
#   80-99: critical kernel processes and phyfw
#   60-79: time-critical flight software
#   40-59: normal kernel processes - default 40 (see "catch-all" at bottom)
#    1-39: normal userspace processes, deprioritized kernel processes
#       0: non real-time software
#

############################
# Using this file
############################
#
# Format per line is: <flags>:<sched>:<prio>:<affinity>:<regex>

:f:70:0x1:user_terminal_control
```

From the rule above, we can understand that a process named `user_terminal_control` is run with the following properties:

- Affinity: `0x1` → it will run on the first core (processor #0). The meaning of the affinity value indicates on which CPUs the process is allowed to run, by taking the binary representation of that number and selecting only the positions in which there is a one. As an example:

$$\texttt{0x1} = 2^0 \quad \rightarrow \quad \text{CPU } \#0$$
$$\texttt{0xD} = 2^0 + 2^2 + 2^3 \quad \rightarrow \quad \text{CPUs } \#0, \#2, \#3$$

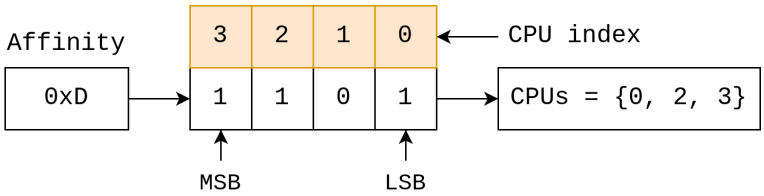


Figure 7.1: From affinity to CPU indexes

- Priority: `70` → it will have a priority value of 70, which will be used by the scheduling algorithm, and since it falls in the 60-79 range, we know it is a "time-critical flight software".
- Scheduler: `f` → it will use the First in First out (`FIFO`) scheduling algorithm. Other possible algorithms are batch (`'b'`) and round-robin (`'r'`).

We will talk more about this and other processes in Section 7.3.

7.2.2 Filesystem & Mount points

The root filesystem structure is a classical Linux structure, so we'll just focus on SpaceX-specific parts of the filesystem. First of all, the root filesystem is copied in RAM by the last bootloader (U-Boot, BL33), and is mounted at `/`. As seen in the introduction to this chapter, the eMMC is partitioned by the kernel by using a command line option. Partitions can be accessed by the system through the files `/dev/blk/mmcblk0pN` where `mmcblk0` is the name of the eMMC and `N` is the partition index, starting from 1. For convenience, a script will then create some symbolic links to these partitions to use a more explicit name, as shown in Listing 7.4.

Listing 7.4: `/bin/make_ut_blks`

```
# [...]
ln -s /dev/mmcblk0p1 /dev/blk/bootfip0
ln -s /dev/mmcblk0p2 /dev/blk/bootfip1
ln -s /dev/mmcblk0p3 /dev/blk/bootfip2
ln -s /dev/mmcblk0p4 /dev/blk/bootfip3
# [...]
```

Since almost every partition is duplicated, another script will then create additional links in the folders `/dev/blk/current` and `/dev/blk/other`, the first one containing the partitions that are currently being used by the system and second one containing the other ones, that will be used in case of a software update. The system knows which partitions have been used for the current boot by looking in `/proc/device-tree/chosen/linux_boot_slot` which is populated by the bootloader.

Then, the `runtime` partition is unpacked using `sxverity` (see Section 4.2.2) and the content is extracted to `/sx/local/runtime`. This partition contains two folders:

- `bin` contains binaries and executable scripts
- `dat` contains a lot of additional data like AppArmor rules, hardware-specific configurations and generic configuration files, such as `dat/common/runtime` which is the (second) configuration file used by the init script.

After these operations, the root filesystem is remounted with the `ro` (read-only) flag. Additional partitions are then mounted, such as:

- `/dev/blk/current/version_info` on `/mnt/version_info` (through `sxverity`).
- `/dev/blk/dish_cfg` on `/mnt/dish_cfg` (through `LUKS`).
- `/dev/blk/edr` on `/mnt/edr` (through `LUKS`).

7.2.3 Networking

Since the device we are analyzing is a network device, this section is expected to be the most interesting, but due to the problems we had with gaining root access to the device and the difficulties encountered in reverse engineering lower-level binaries, the network configuration of the dish is not entirely clear at the moment of writing. In Listing 7.5 you can see the output of the `ip` command that lists all the interfaces that are `UP`, with their corresponding addresses.

Listing 7.5: Output of the `ip` command on the physical device

```
[root@user1 ~]# ip a show up
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
    1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth_user: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN
    group default qlen 1000
    link/ether 26:12:ac:1a:80:01 brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.1/24 scope global eth_user
       valid_lft forever preferred_lft forever
    inet 34.120.255.244/32 scope global eth_user
       valid_lft forever preferred_lft forever
    inet6 fe80::2412:acff:fe1a:8001/64 scope link
       valid_lft forever preferred_lft forever
3: eth0: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1500 qdisc mq state UP group
    default qlen 1000
    link/ether 26:12:ac:1a:80:01 brd ff:ff:ff:ff:ff:ff
6: eth_prime@eth_user: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    group default qlen 1000
    link/ether 26:12:ac:1a:80:01 brd ff:ff:ff:ff:ff:ff
    inet 172.26.128.1/12 scope global eth_prime
       valid_lft forever preferred_lft forever
    inet6 fe80::2412:acff:fe1a:8001/64 scope link
       valid_lft forever preferred_lft forever
7: sx0: <POINTOPOINT,NOARP,UP,LOWER_UP> mtu 1300 qdisc pfifo_fast state UP group default
    qlen 500
    link/none
    inet 172.16.10.2/24 scope global sx0
       valid_lft forever preferred_lft forever
    inet6 2620:134:b000:bdf:9cf4:8e0::/48 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::a036:3c0b:cc0e:fdc/64 scope link stable-privacy
       valid_lft forever preferred_lft forever
```

The only thing we are sure about is that `eth_user` is the interface that faces the Wi-Fi router, hence it is the one receiving the traffic coming from the user. But other than that, we couldn't understand how the traffic was further routed and forwarded to the RF hardware. Even `tcpdump` fails to capture those packets, thus we guess that it is using a network stack that is not the one included in the Linux kernel.

7.3 Daemons

After the elaborate boot procedure and system configuration we finally reach a state in which some processes are running, each one with a unique task, to provide the user with the service the device is built for, i.e. a satellite internet connection. As you may guess, many things happen in the background to ensure a stable enough internet connection, such as sending and receiving traffic to and from satellites, choosing which satellite to connect to, continuously changing satellite when it has moved too far without disrupting the connection, handling user requests coming from the mobile application, etc... Furthermore, these processes need to continuously communicate with each other to synchronize and cooperate, and with Starlink's cloud services for backend functionalities.

The only sources of information about these processes and how they work are the binaries themselves, so we had to reverse-engineer them to find out their purpose in the system. All of them a stripped, meaning that they do not contain any symbol (e.g. function names), most of them are C++ binaries, some of them are even statically linked and one of them is a go binary. Reversing C++ binaries is quite a challenge, due to the complex low-level implementation of classes, particularly when inheritance or generic types are used. Moreover, without any symbols, it is really difficult to understand the high-level structure of the code. On top of that, in the case of statically linked binaries, you also need to reverse library functions. On the other hand, reversing the go binary was much easier, thanks to the debugging symbols included in the binary that are needed by the go runtime.

7.3.1 C++ binaries

In this section, we will describe the methodology used for reversing the C++ binaries, with a brief description of the most important ones. Ghidra, the open-source software from NSA, has been used for reverse engineering analysis, along with some plugins and other software that we will mention in this chapter. From a first look at the binaries, it became clear that we were handling C++ binaries, and from a quick analysis of the printable strings found in the binaries, we could also understand that a lot of code has been shared among all the runtime utilities.

Library functions identification

The first problem we tried to solve was to identify the libraries that were statically linked to the binaries under study because otherwise, the analysis would have been impossible to do in a reasonable time. To do that, we used the binary diffing technique, which consists, given two compiled programs, of identifying common functions in the two binaries. For the diffing to be successful we need the libraries, compiled with the same version of the compiler, with the same options and for the same architecture, so we either have to find them somewhere or recompile them ourselves. Since a lot of code is shared among the binaries, it is likely that also the library the processes are using are the same ones. And since there are some dynamically-linked binaries, the dynamically-linked libraries have to be somewhere in the filesystem. The libraries were stripped from symbols as well, but the exported functions (the ones you call from the program you are linking it to) have to be present. So we tried to diff the statically linked binaries with the available libraries (such as `libc`, `libcrypto`, `libssl`, etc.) using `BinDiff`, an open-source binary diffing tool.

Figure 7.2 shows the result of diffing a statically linked binary (connection_manager) with a very common library (libc). In this case, the tool has been able to match around 1800 functions from the libc, among which there are plenty of false positives, but also some true positives.

To include these results in Ghidra to proceed with the static analysis, a plugin can be used: BinDiffHelper. In our case, though, it didn't help because it was crashing while importing the symbols and, most importantly, it does not provide a way to filter out false positives. BinDiff's results include also some statistics about every matched function, such as the similarity and the confidence. These values can help us understand if the match is a true or a false positive, so we can use them to filter out bad results before importing them into Ghidra. Since this cannot be done using existing tools, we've implemented a Python script for Ghidra that can import symbols from BinDiff's outputs, while

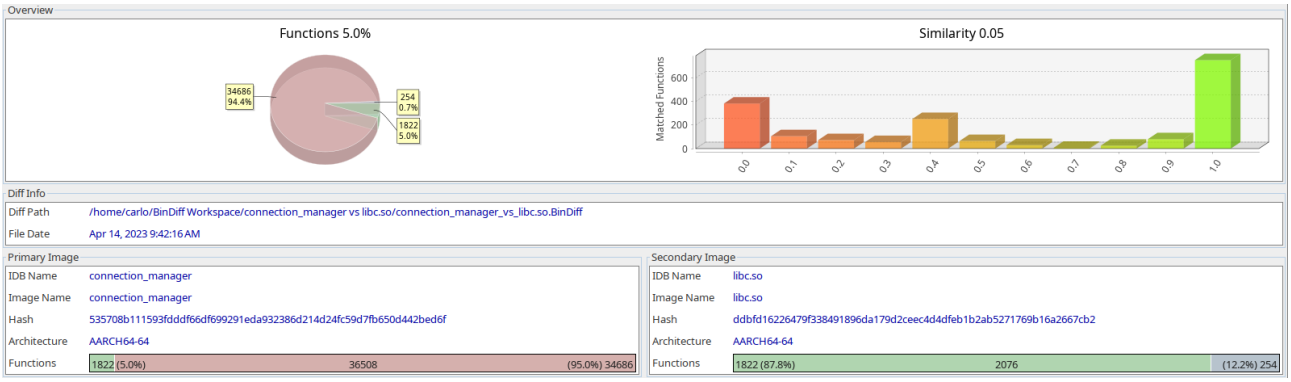


Figure 7.2: `bindiff` of `connection_manager` vs `libc`

also giving the user the capability of filtering matched functions by setting a lower threshold for the confidence and similarity or by defining more complex filters directly in Python [25]. This technique is not perfect because even if the compiled code of two functions is almost identical, it is not guaranteed that they are the same function, and the same function could be compiled differently in the two binaries.

Design pattern and related challenges

After recovering some symbols using the aforementioned technique, we started analyzing the binary in Ghidra. Since it is a C++ binary we also tried recovering information about classes and their methods. Some Ghidra plugins have been tested but we weren't able to find one that worked well. Most of them were just able to recover class names and namespaces (i.e. demangling), but none of them was capable of re-constructing the data types and resolving method calls. This is mainly because when using inheritance, multiple inheritance, virtual methods and generic types, the underlying structures that make all these features work are very complicated, and it is often difficult to understand statically which method will be called and what kind of subclasses we are dealing with. Moreover, it looks like these programs implement the State Machine Design Pattern [30], which heavily uses the aforementioned features of Object Oriented Programming. Dynamic analysis is often required to fully understand what the program does, which wasn't possible on-device until the very end of my research period, and was only partially possible in the emulated environment (see Chapter 8). Due to these reasons, the static analysis of C++ binaries did not produce many results, and thus we focused on the go binary, which will be discussed in the next section.

Overview of C++ binaries

Here is anyway a brief overview of the studied binaries, using the little information we could gather from them, by using an OSI-like structure of the network stack.

- `phyfw` (Physical Firmware, perhaps) handles the physical layer of the satellite communication, which includes modulation/demodulation of RF signals.
- `rx_lmac` and `tx_lmac` (rx/tx lower Medium Access Control, perhaps) fall in the data link layer, and handle the physical access to the medium, separately for receiving and transmitting.
- `umac` (upper Medium Access Control, perhaps), could represent the network layer. It handles the access to the medium, at a higher level, and coordinates between the transmission and reception of frames. It may also be in charge of choosing which satellite to connect to.
- `connection_manager` could represent the transport layer and, if it's the case, it handles stateful connections between the dish and satellites (or gateways directly, abstracting the underlying infrastructure), in which the traffic will be exchanged.

- `ut_packet_pipeline` is probably used to create an encrypted tunnel in which user traffic will be exchanged using the secure element on the dish for handshakes. This could be associated with the known protocols such as TLS, DTLS, IPsec or, again, a custom one.

Other than these network-related processes there also are some processes that handle system telemetry, software updates, system health status and outage detection/reporting and finally a "control" process that acts as an orchestrator for all the other processes.

7.3.2 Go binary

One of the runtime processes is a go binary, which is an open-source (compiled) programming language from Google that claims to provide memory safety guarantees and built-in concurrency, thanks to its runtime, which is included in every go program and handles the whole program status, garbage collection and safety checks during the execution while having a small impact on performance.

Go binaries are statically linked and they include the go runtime, so they are pretty big, but luckily they also include symbols that are used by the runtime for comprehensive runtime error reporting, which includes function names, source code line numbers and data structures. All this precious information can be recovered using a plugin for Ghidra called GolangAnalyzer which, as you can see in Figure 7.3, was quite effective. The extension also recovers complex data types and creates the corresponding C-like structures in Ghidra, which is extremely useful when working with an OOP language. Additional manual analysis is needed because of the custom calling convention used by Go, but after this, the resulting disassembled C code is easily readable.

Figure 7.3: Golang Analyzer analysis result

The binary we are analyzing is `user_terminal_frontend`, and its main task is acting as a back-end for the mobile application and the web interface accessible to the users. Thus, through this process, the user can edit some basic configurations of the dish and get some information from the dish such as statistics about the connection speed or an obstruction map in which (s)he can see if the antenna has a good visibility of the sky. Communications with the mobile app and the web interface use Google's RPC (gRPC) framework, but we'll dive into this in Chapter 9. Thanks to this process we've also understood some other components that are implemented also by C++ binaries, such as general configurations and inter-process communications (IPC) (see Section 9.2).

7.3.3 Architecture

To summarize this section, in Figure 7.4 you can see a sketch of the architecture of the runtime (not complete), in which you can see that processes at the bottom, "closer" to the hardware, are statically linked, probably for performance reasons, and communicate only with the control process, while the other ones also communicate with Starlink's cloud services, through gRPC. In Chapter 9 we will tackle all the communications shown by this picture in more detail. And finally, the is the go binary (which is technically statically linked as well, but just because of the language constraint), which communicates with frontend applications used by the user.

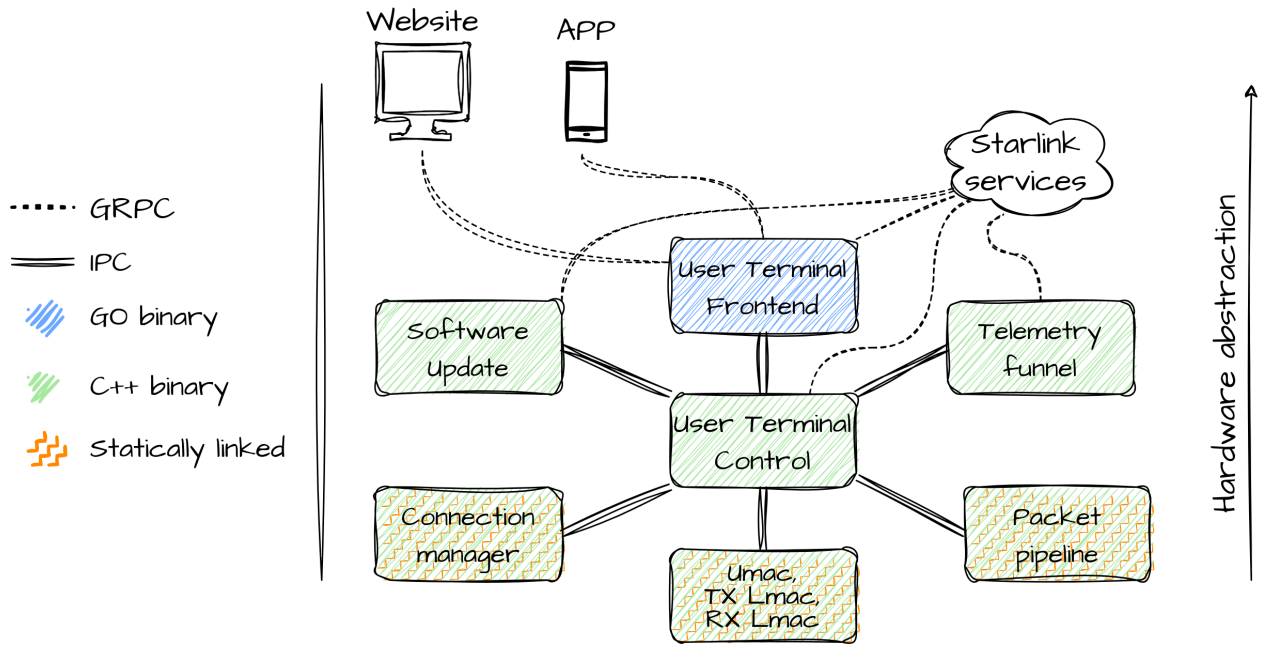


Figure 7.4: Sketch of the runtime's architecture

8 Runtime Emulation

Due to the negative results of the Fault injection attack (Chapter 6), we didn't have access to a live device on which to check our findings or perform some dynamic analysis. Thus, we tried to set up an emulated environment, as similar as possible to the real device, that would be capable to execute runtime binaries. We are emulating the entire system (full-system emulation), starting from the kernel, using QEMU as an emulator engine. The following sections will describe every challenge we had to deal with while setting up the environment and the final result, containing the ones we couldn't solve.

8.1 QEMU machine hardware

The first choice to be made is which hardware we want QEMU to emulate. When you want to emulate an IoT device using QEMU you usually look for the hardware implementation of the particular device for QEMU, which is usually available for common, off-the-shelf devices such as Arduino, Raspberry Pi, and less-known boards as well. The hardware implementation of our device was, of course, not available, so we used the `(aarch64) virt` machine, which is the most generic one. The proper way to emulate the whole device would have been to construct this machine specification for QEMU, as well as implement emulators for every piece of hardware that is present on the board. The problem is that most of the peripherals present on the device are not open hardware, and even if they did, implementing all of them in QEMU would have been a lot of work. Instead, by using the `virt` machine and tweaking the Device Tree it is much easier, at the cost of not having most of the hardware peripherals and thus some limitations, see Sections 8.3 and 8.6.

8.2 Kernel

Another problem was the choice of the kernel to run in QEMU. We tried with the original one, extracted from the FIT of the firmware, but that didn't work in the emulated environment. So we decided to

compile one ourselves. Unfortunately, the open-source version of Linux published by SpaceX is 5.10.90 [16], while the one found on the dish was 5.15.55, so we used the mainstream Linux kernel. A lot of tweaks in the compile-time configuration had to be made for it to boot, some of them required by QEMU and some of them required by Starlink's software. It is possible to extract this configuration from a compiled Kernel image, using the script `extract-ikconfig` from the Linux kernel repository, which was used to find differences between the default one and the one configured by SpaceX.

8.3 FDT

As we have seen in Section 4.1, the device tree of the device not only contains information about hardware peripherals but also data that is used by the runtime, such as public keys used by `sxverity`. Additionally, the U-Boot bootloader also populates the FDT before booting the Linux kernel, by adding, for example, which set of partitions have been used in the current boot, the name of the main network interface and more. All this information is of course not included in the FDT set by QEMU for the `virt` machine, thus we extracted this FDT, using the `dumpdtb` flag, and added the missing information, as shown in Listing 8.1, which can then be recompiled using the Device Tree compiler (`dtc`) and given to QEMU using the `-dtb` flag.

Listing 8.1: Customized FDT for QEMU

```
# ...
model = "spacex_satellite_user_terminal";
compatible = "st,gllcff";

chosen {
    linux_boot_slot = "0";
    ethprime = "eth_user";
    board_revision = "rev2_proto3";
    bootfip_slot = <0x00>;
    boot_slot = <0x0006>;
    boot_count = <0x0010>;
    # ...
};

security {
    dm-verity-pubkey = <REDACTED>;
    # ...
};
# ...
```

8.4 rootfs

As the root filesystem, we used the one we extracted from the dish, with some modifications.

- Since we want to have access to the emulated dish, it must think it is a development version so that password access is enabled, thus we patched the `is_production_hardware` script, just as we have seen in Chapter 6. This could have been done in multiple ways, such as directly editing the `/etc/shadow` file, or adding our public key to the SSH's `authorized_users` files, but it is more effective as we did because development hardware will also enable other debugging features.
- We also included the extracted runtime where it would have been mounted and removed that step from the `/etc/runtime_init` file so that it will skip the integrity check and we'll be able to also tamper with the content of that partition.
- In the `/etc/runtime_init` file we also have added some custom steps, for example, one that sets up our emulated network, and one that mounts read-write partitions as emulated volumes.

Additional patches will be needed for other programs to start in the emulated environment, here is one from `Starlink software update` as an example. Listing 8.2 shows a snippet of disassembled

code from the binary that was making the process crash in the emulator and in Listing 8.3 you can see the corresponding decompiled C code. In this case the function call to `FUN_001f61e0` was failing (returning `false`), and because of that the program panics. A simple patch for this kind of problem is to just skip the check after the call, replacing the instruction with a `nop` (No operation), which does nothing and continues with the normal execution of the next instruction, which in this case is the "good" branch. The patch is shown in Listing 8.4.

Listing 8.2: Code in Starlink software update that makes the program crash in the emulator

```
    001bf1dc 01 dc 00 94     bl          FUN_001f61e0
    001bf1e0 1f 1c 00 72     tst         param_1, #0xff
    001bf1e4 60 87 00 54     b.eq        LAB_001c02d0

LAB_001c02d0:
    ; Here an error message is printed and panic is called
```

Listing 8.3: Decompiled code of Listing 8.2

```
res = FUN_001f61e0(var1,3);
if (res == false) {
    panic("SacAbortIfNot","","","external/rocket/src/flight/sat/all/swupdate/
        starlink_software_update_bin.cc",0x118,"",0);
}
```

Listing 8.4: Patched code

```
    001bf1dc 01 dc 00 94     bl          FUN_001f61e0
    001bf1e0 1f 1c 00 72     tst         param_1, #0xff
    001bf1e4 1f 20 03 d5     nop
```

We have also included some additional software that will be used for testing purposes such as `gdbserver`. But for these programs to be able to run, we either had to cross-compile them using the same build toolchain, or cross-compile them statically. The modified root filesystem is then repacked into a `cpio` image and then compressed using the script shown in Listing 8.5.

Listing 8.5: `pack-rootfs.sh`

```
#!/bin/bash

echo "Copying filsystem folder..."
# we use sudo because here because we want root to be the owner of the content
sudo cp -r rootfs _rootfs
cd _rootfs

echo "Creating rootfs filesystem..."
find . | sudo cpio -o --format=newc > ../rootfs.cpio
cd ../

echo "Compressing filsystem image..."
sudo gzip -c rootfs.cpio > rootfs.cpio.gz

echo "Cleaning up..."
sudo rm rootfs.cpio
sudo rm -r _rootfs
```

8.5 Persistent memory & Networking

Even though both the root filesystem and the runtime are already placed in memory when the Linux kernel boots, a lot of processes directly access some partitions of the eMMC. So we've also instructed QEMU to create a raw virtual block device, containing the original image dumped from the physical board. But since it is not seen by the kernel as an eMMC chip, the assigned name is different from the

one assigned by the physical device. Because of this, we had to change every reference to `mmcblk0` in `vda`, which is the name assigned by the kernel in the emulator. Fortunately, as we have seen in Subsection 7.2.2, the device only uses the name of the device in a script that creates some symbolic links to every partition, so we just had to patch that script and the command line argument for the kernel. Partitions that are mounted with write permission, are instead mapped to a folder on the host so that it is possible to inspect their content afterward.

As for the network, it is not necessary to replicate the exact network configuration of the dish (as it is not very clear to us), we just need to have the right interface names and internet access. This has been done by using a tap interface, bridged to the host which will act as NAT, as shown in Listings 8.6 and 8.7

Listing 8.6: Network configuration on the host

```
#!/bin/bash

ifconfig $1 0.0.0.0 promisc up
brctl addbr virbr0
brctl addif virbr0 $1

ip link set dev virbr0 up
ip link set dev $1 up

ip addr add 192.168.100.2/24 dev virbr0
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -t nat -A POSTROUTING -o wlp0s20f3 -j MASQUERADE
```

Listing 8.7: Network configuration in the guest

```
#!/bin/sh

ip link set dev eth0 name eth_user
ip link set dev eth_user up
ip addr add 192.168.100.1/24 dev eth_user

route add -net 0.0.0.0 netmask 0.0.0.0 gw 192.168.100.2 dev eth_user
```

8.6 Results and Limitations

The resulting command line is shown in Listing 8.8. On line 5 you can see the command line arguments that will be passed to the kernel, some of them are the same ones the bootloader sets, such as `rdinit` (the path to the init script) and `blkdevparts` (with the modified device name). And others are custom, like `console` (to set the console defined in the FDT), `nokaslr` (for debugging binaries more easily). Parameters that have the `sx-` prefix are meant to be read by the init script, in this case, we have set a fixed public IP address (that won't be used) and the board's serial number.

The arguments of `qemu-system-aarch64` have already been explained in the previous sections, without actually naming them. The ones between lines 23 and 29 set the right type of emulated hardware, then we select the Linux kernel image (line 30), the ramdisk (line 31), we set the emulated hard drive (lines 32 and 33), the custom device tree (line 34), the emulated network device (lines 35 and 36) and finally some shared folders for persistency among emulations.

Listing 8.8: QEMU command line

```
#!/bin/bash

set -e

bootargs="
    console=ttyAMA0
    nokaslr
    panic=5
    ro
    root=/dev/ram
    rdinit=/usr/sbin/sxruntime_start
    mtdoops.mtddev=mtdoops
    trace_buf_size=5M
    rcutree.kthread_prio=80
    uio_pdrv_genirq.of_id=generic-uio
    audit=1
    blkdevparts=vda:0x100000@0x00000000(BOOTFIP_0),0x100000@0x100000(BOOTFIP_1),0
        x100000@0x200000(BOOTFIP_2),...
    sx-ipaddr=172.26.128.1
    sx-serialnum=123
"

qemu-system-aarch64 \
    -nographic \
    -no-reboot \
    -machine virt \
    -machine type=virt \
    -cpu cortex-a53 \
    -smp 4 \
    -m 2G \
    -kernel "linux-5.15.55/arch/arm64/boot/Image" \
    -initrd ./rootfs.cpio.gz \
    -blockdev driver=raw,file.filename=original.img,file.driver=file,node-name=vda \
    -device virtio-blk-device,drive=vda \
    -dtb ./FDTs/custom.dtb \
    -device e1000,netdev=vm0 \
    -netdev tap,id=vm0,script=./networking/tap0-up.sh,downscript=./networking/tap0-down.
        sh \
    -virtfs local,path=./disks/dish-cfg,mount_tag=dishcfg,security_model=mapped-xattr \
    -virtfs local,path=./rootfs,mount_tag=rootfs,security_model=mapped-xattr \
    --append "$bootargs"
```

As explained in previous sections, most of the hardware is not present in the emulated environment, so every component that tries to use it will fail. Lower-level processes, such as `phyfw` and `[rx/tx]_lmac`, whose main task is to interact with Starlink's hardware won't work in this environment. But also other binaries require some hardware to be present, the most common one is the secure element, which is used in most of the cryptographic exchanges. So for those binaries to work we patched every instruction that would make the program crash, but if the hardware is required for substantial parts of the process, this solution is pointless. In the end, we managed to emulate, among the main processes discussed in Section 7.3, only `user_terminal_frontend`, `starlink_software_update`, `umac` and the ones related to telemetry, along with smaller processes. Further work on this topic could incrementally add support for some peripherals of the board, hence being able to emulate more and more processes. In Listing 8.9 you can see the console output of the final phase of the boot process in the emulated environment.

Listing 8.9: QEMU output

```
# [...]
# kernel messages and (a lot of) errors from applications
# trying to access missing hardware peripherals
# [...]

setup_console: Development login enabled: yes

SpaceX User Terminal.
user1 login: root
Password:

                                                                *
                                                 +
                                       +    +
                                +     +
                           +      +
+ + + + +              +     +
  +        +       +     +
     +       + +      +
        +   +      +
          +      + +
      +      +        +
   +       +    +        +
 +       +         +        +
+ + + + +             + + + + +


The Flight Software does not log to the console. If you wish to view
the output of the binaries, you can use:

tail -f /var/log/messages

Or view the viceroy telemetry stream.

[root@user1 ~]#
```

9 Communications

In IoT devices, communication with other devices is often a crucial function, if not the main one. This is also the case for our device, which is basically an internet gateway and, as you can imagine, in this case, communication between the device and the satellites is the main function of the User Terminal. We briefly discussed the physical layer of this communication in the first chapters, but we didn't focus on that. On a higher layer, the communication with satellites is split into two "planes":

- The Data Plane, which contains user traffic to and from the Internet;
- The Control Plane, which contains every other kind of traffic, such as control messages between the antenna and the satellite (e.g. Connection handshake).

But these are not the only communications happening in the device, in the following sections we will also see how the device interacts with user front-end applications and also how the different components inside the device communicate with each other.

9.1 External communications

The User Terminal heavily interacts with external devices, which can be split into two major categories:

- Devices that reside in the internal network that the dish creates, i.e. user-owned devices, which can run front-end applications such as the mobile app or the web interface.
- All the other devices, which include satellites and backend services inside the SpaceX private network.

These interactions use, as the application layer, Google's Remote Procedure Calls (`gRPC`), which underneath uses `protobuf`, and finally `TCP`.

9.1.1 Satellites & Starlink's cloud services

Many binaries in the runtime need to communicate with backend services to, for example, ensure that the user has an active subscription. All these communications happen in the Control plane, and all the devices are in the private network of Starlink, only reachable through a User Terminal. Due to this, we couldn't test these interactions in the emulated environment, thus we don't have many details on this.

Luckily, this time SpaceX did not choose to implement a custom protocol, they are instead using `gRPC`. Messages and services are defined by using the protobuf's proto definition, which is a human-readable representation of the format of each message and the available services that each server is exposing. In Listing 9.1 you can see an example message, defined using the proto definition format.

Listing 9.1: A proto definition

```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

After defining messages and services, these files are compiled using the protocol buffer compiler, which can also generate stubs in many programming languages, such as C++, Python and Go. As an example, in Listing 9.2 you can see a C++ snippet that makes use of the message defined in Listing 9.1. The parsing and encoding of messages is handled by the automatically generated stubs, which is very convenient since you can share these definitions among programs written in multiple languages, without the need of re-implementing everything from scratch, but rather by just focusing on the actual application code.

Listing 9.2: C++ snipped that uses the protobuf stubs

```
int main(int argc, char ** argv) {
    Person john;
    fstream input(argv[1], ios::in | ios::binary);
    john.ParseFromIstream(&input);
    id = john.id();
    name = john.name();
    email = john.email();
    // ...
}
```

This protocol is very convenient for us security researchers as well since the original message definitions are often included in the compiled binaries for `reflection`, which is a feature of the protocol with which you can "ask" a server process which kind of messages and requests it supports, that will be sent as proto definitions. Thanks to this we could extract the original, human-readable definitions from the binaries using the protobuf toolkit (`pbtk`) [11]. As an example, in Listing 9.3 you can see the definition of a message extracted from the `umac` binary.

Listing 9.3: Protobuf definition of a message extracted from the `umac` binary

```
message SatIneHandshakeRequest {
    uint32 satellite_id = 1;
    uint32 gateway_id = 2;
    uint32 channel_id = 3;
    uint32 sequence_num = 4;
    uint32 interval_start_sec = 5;
    string antenna_id = 6;
    SpaceX.API.Common.SpaceTime.GPSTimestamp tx_deadline = 7;
    uint32 mac_software_version = 8;
    SpaceX.API.Common.SpaceTime.GPSTimestamp curr_time = 9;
}
```

Luckily, the names of messages, fields and requests are self-explanatory. The message shown in Listing 9.3 is probably the request sent from the dish to the satellite to initialize a stateful connection, containing which satellite we want to connect to, which gateway we want to use to reach the internet, which radio channel we want to use, from which antenna the handshake is coming from and more. From the information gathered from these message definitions, we could understand how the dish interacts with backend services, such as how information about the user is gathered and how information about available software updates is requested. But, as explained in the introduction to this chapter, we could not test these in the emulated environment, thus we didn't focus on this. This is also because all the communications are using mutual authentication (TLS), by using the secure element on the dish, so it would be very difficult to, for example, impersonate Starlink backend services to sniff the traffic or tamper with it for exploitation.

9.1.2 Front-end applications

The communication with front-end applications is handled by the process `user_terminal_frontend`, which we were able to both run in the emulated environment and reverse-engineer, thanks to the language it has been implemented in (Go). As explained in Subsection 7.3.2, this binary is in charge of handling the communication between the front-end applications, which are the mobile application and the web interface. From these applications, the user can see some statistics of the device and can change some high-level settings such as Wi-Fi credentials, reboot or stow the dish, etc. These interactions are again using gRPC, but do not require mutual authentication. Protobuf definitions can be gathered either by extracting them from the binary (using `pbtk`) or by asking the reflection server of the process itself (using `grpcurl`, for example). Some tools have been implemented as alternative front-end applications that use this protocol [18, 19]. The aforementioned applications, implemented in Python, make use of the gRPC APIs exposed by the frontend binary, providing the user with an alternative user interface to inspect the statistics of the dish. The authors of these applications have probably gathered the protocol definitions from the mobile application or by using the reflection server.

Listing 9.4 shows the definition of the main service. The interesting function is `Handle`, which takes a `Request` as an argument and returns a `Response`. The other function is just a wrapper to call the `Handle` function using streams.

Listing 9.4: Front-end service definition

```
service Device {
    rpc Stream(stream ToDevice) returns (stream FromDevice);
    rpc Handle(Request) returns (Response);
}
```

In Listing 9.5 you can see the (partial) definition of the `Request` message, which contains a few ids and one specific request, among the ones listed. Every inner request has its definition, with the parameters the server needs to process the request, and a corresponding response that will hold the result.

Listing 9.5: Partial definition of the `Request` message

```
message Request {
    uint64 id = 1;
    uint64 epoch_id = 14;
    string target_id = 13;

    oneof request {
        GetNextIdRequest get_next_id = 1006;
        AuthenticateRequest authenticate = 1005;
        EnableDebugTelemRequest enable_debug_telem = 1034;
        FactoryResetRequest factory_reset = 1011;
        GetDeviceInfoRequest get_device_info = 1008;
        GetHistoryRequest get_history = 1007;
        GetLogRequest get_log = 1012;
        GetNetworkInterfacesRequest get_network_interfaces = 1015;
        GetPingRequest get_ping = 1009;
        PingHostRequest ping_host = 1016;
        GetStatusRequest get_status = 1004;
        RebootRequest reboot = 1001;
        SetSkuRequest set_sku = 1013;
        SetTrustedKeysRequest set_trusted_keys = 1010;
        SpeedTestRequest speed_test = 1003;
        SoftwareUpdateRequest software_update = 1033;
        DishStowRequest dish_stow = 2002;
        StartDishSelfTestRequest start_dish_self_test = 2012;
        DishGetContextRequest dish_get_context = 2003;
        DishGetObstructionMapRequest dish_get_obstruction_map = 2008;
        DishSetEmcRequest dish_set_emc = 2007;
        DishGetEmcRequest dish_get_emc = 2009;
        DishSetConfigRequest dish_set_config = 2010;
        DishGetConfigRequest dish_get_config = 2011;
        // [...]
    }
}
```

With this information, we can now interact with the server by using the `grpc_cli`, which comes with gRPC and it's a command line interface that lets you make gRPC requests; or by implementing a client in one of the many supported languages, such as Python, after "compiling" the protocol definition and generating the stub code for the chosen language. In Listing 9.6 you can see a simple Python script that sends the `GetDeviceInfoRequest` and prints out the response.

Listing 9.6: Python script using gRPC to send the `GetDeviceInfoRequest`

```
def get_device_info():
    channel = grpc.insecure_channel(TARGET)
    stub = device_pb2_grpc.DeviceStub(channel)

    request = device_pb2.Request(
        id=1,
        epoch_id=1,
        target_id="unknown",
        get_device_info=device_pb2.GetDeviceInfoRequest()
    )

    response = stub.Handle(request)
    return response

print(get_device_info())

######################## Example output (from emulator) ########################
# id: 1                                                                        #
# api_version: 6                                                               #
# get_device_info {                                                            #
#   device_info {                                                              #
#     id: "uta5cf81ba-115e-44a2-a62d-e189d7fe42df"                             #
```

```
#     hardware_version: "rev2_proto3"                                          #
#     software_version: "c36a30b4-93ac-4a14-b663-0622ef7ed944.uterm.release" #
#     is_dev: true                                                             #
#     generation_number: 1674091226                                            #
#   }                                                                          #
# }                                                                            #
################################################################################
```

There are two ways of communicating with this gRPC, either by using an insecure channel, like Listing 9.6 shows, or by using a secure channel, which involves TLS and mutual authentication by using certificates stored in the secure element. The mobile application and the web interface both use the insecure channel, so the encrypted one must be used by something else. The secure channel requires a private key which is not available in the dish, and the public key is stored in the secure element, so to test this feature we would have to patch the program in multiple places.

Among the requests that can be made to the server, many of them are meant for front-end applications, a few examples are:

- `FactoryResetRequest`, which requests a factory reset of the dish
- `GetDeviceInfoRequest`, which is the one shown in Listing 9.6
- `GetStatusRequest`, which requests the status of the dish
- `RebootRequest`, which asks the dish to reboot

But some requests do not look like they are used by those applications, such as:

- `SetTrustedKeysRequest`, shown in the Listing 9.7, which supposedly sets the provided public keys for future use by the process or by the SSH agent.[1]

  Listing 9.7: Message definition for `SetTrustedKeysRequest`

  ```
  message SetTrustedKeysRequest {
      repeated PublicKey keys = 1;
  }
  ```

- `GetHeapDumpRequest`, which supposedly returns a dump of the Heap section of the process.
- `SoftwareUpdateRequest`, which supposedly initiates a software update with the provided update bundle. This request will be further described in Section 10.1.

Unfortunately, most of these requests are not implemented in the binary we were analyzing (e.g. `SetTrustedKeysRequest`, `GetHeapDumpRequest`), and some of them require authentication (e.g. `DishGetContextRequest`, `DishGetEmcRequest`) both on the transport layer (secure gRPC channel) and application layer (by using the `AuthenticateRequest`). We are not entirely sure who is supposed to use these requests and why most of them are not implemented in the binary, they could be used by another Stalink product such as the WiFi router, or by Starlink support in the case of a partially bricked device, for remote assistance. The most interesting request that is both implemented and does not require authentication is the `SoftwareUpdateRequest`, which will be explained better in Section 10.1. As an example, here is how the `SetTrustedKeysRequest` works.

`SetTrustedKeysRequest`

As shown in Listing 9.7, the request contains an array of `PublicKey`s, whose definition is shown in Listing 9.8. A public key is composed of the public key encoded in hexadecimal and preceded by the name of the encryption algorithm that has to be used, along with a list of capabilities the key should enable, shown in Listing 9.9.

[1] We are not sure how these keys will be used because the implementation of this method is missing

Listing 9.8: Definition of message `PublicKey`

```
message PublicKey {
    string key = 1;
    repeated Capability capabilities = 2;
}
```

Listing 9.9: Definition of enum `Capability`

```
enum Capability {
    READ = 0;
    READ_INTERNAL = 13;
    READ_PRIVATE = 7;
    LOCAL = 14;
    WRITE = 1;
    WRITE_PERSISTENT = 11;
    DEBUG = 2;
    ADMIN = 3;
    SETUP = 4;
    SET_SKU = 5;
    REFRESH = 6;
    FUSE = 8;
    RESET = 9;
    TEST = 10;
    SSH = 12;
}
```

Listing 9.10 shows how a message of this kind can be constructed and sent to the server. As you can see our public key has been generated with the elliptic curve `ecdsa256` and the chosen capability is just SSH access. We set all the other parameters, i.e. `id`, `epoch_id` and `target_id`, to random values for the moment.

Listing 9.10: Python script constructing the `SetTrustedKeysRequest` message

```
request = device_pb2.Request(
    id=0,
    epoch_id=0,
    target_id="unknown",
    set_trusted_keys=device_pb2.SetTrustedKeysRequest(
        keys = [
            command_pb2.PublicKey(
                key = "ecdsa256:" + hexkey,
                capabilities = [
                    command_pb2.Capability.SSH
                ]
            )
        ]
    )
)
```

Sending this request results in an error from the server, saying that the `target_id` does not match the one of the dish, and provides the correct one. After correcting the target id, the error says that also the `id` and `epoch_id` are not correct, without saying what value we should send. To get those missing ids, we need first to send another request, containing the `GetNextIdRequest`, which is an empty request. This request is shown in Listing 9.11, this time the fields `id` and `epoch_id` are ignored by the server.

Listing 9.11: Python script constructing the `GetNextIdRequest` message

```
request = device_pb2.Request(
    id=1,
    epoch_id=1,
    target_id=DEV_ID,
    get_next_id=common_pb2.GetNextIdRequest()
)
```

This request makes the server return an error as well, this time saying that the request is not implemented. But by further analyzing the binary and by debugging the server we found that the same error is returned also when a request is expected to be signed, while it isn't. Thus, the request has to be wrapped by a `signed_request`, which is of type `SignedData`. This request contains a

serialized "inner" request, with its ecdsa256 signature. The public key used for verification is hard-coded in the binary, so we patched it and inserted a newly generated key, for testing purposes. In Listing 9.12, you can see the Python code that generates the `SignedData` message from an unsigned message.

Listing 9.12: Python script that signs a message

```
def sign(message):
    sk = ecdsa.SigningKey.from_pem(key_pem, hashfunc=hashlib.sha256)
    data = message.SerializeToString()
    signature = sk.sign(data,
        hashfunc=hashlib.sha256,
        sigencode=ecdsa.util.sigencode_der)
    return common_pb2.SignedData(data=data, signature=signature)
```

After successfully signing the `GetNextIdRequest` we were able to get the right values to put in the `SetTrustedKeysRequest`, just to find out that this request is not implemented. This time it's really unimplemented, by analyzing the handler code in the binary, we found the code that verifies that all the parameters are correct and in the right format, but then the code that was supposed to write the provided key somewhere is just missing.

Further work on this topic would be to analyze further every request handler for bugs or, better to fuzz them since there exist effective mutators for the protobuf protocol, such as `libprotobuf-mutator`.

9.2 Inter-Process Communication

According to Figure 7.4, Inter-Process Communication (IPC) is the last type of communication to be discussed. Every process running in the runtime continuously shares information with other processes, to collaborate and share statistics. As you can see from the figure, every process only communicates with the User Terminal Control, which acts as an orchestrator for the whole runtime. The protocol they are using has been designed by SpaceX, and it's called `Slate Sharing`.

Transport Layer

It uses `UDP` for transport, and every process starts listening on a different port, on the loopback interface and will receive messages from the control process on that port. On the other hand, the control process starts listening on multiple ports, one for every process that needs to communicate with it, so that the communication is bidirectional and without conflicts between processes. Port numbers are configured through a configuration file that can be found at `/sx/local/runtime/-common/service_directory`, in Listing 9.13 you can see a part of it, which lists the port numbers for communications between software update and control, and between frontend and control.

Listing 9.13: Service Directory configuration file

```
################################################################################
# Software update
software_update_to_control                localhost   27012       udp_proto
control_to_software_update                localhost   27013       udp_proto

################################################################################
# Frontend
control_to_frontend                       localhost   6500        udp_proto
frontend_to_control                       localhost   6501        udp_proto
```

Application Layer

Every couple of processes exchange different kinds of data, and messages do not contain any information about the content nor the structure of the data they transport, which is not the case for protocols

such as JSON or XML. Thus, to understand the content of messages we had to reverse engineer one of the binaries that makes use of the protocol, and in our case, the best choice was once again the user terminal frontend, which is the one implemented in Go.

Data is exchanged in binary form, by sending raw packed (no padding) C structures, in big-endian. Every message contains a header, holding some information about the message, and a body, containing the actual data to be shared. In Listing 9.14 you can see the data structure representing the message header, from the GolangAnalyzer Ghidra plugin. With the same technique, we also extracted the structure of the body of messages between the frontend process and control.

Listing 9.14: `SlateHeader` structure definition

```
*************************************************************
* Name: sx_slate.SlateHeader                                *
* Struct:                                                   *
*   +   0x0    0x4 uint BwpType                             *
*   +   0x4    0x4 uint Crc                                 *
*   +   0x8    0x8 longlong Seq                             *
*   +  0x10    0x4 uint Frame                               *
*************************************************************
```

The header contains:

- `BwpType`, which is a fixed value, acts as a "magic number" for the protocol (`00 00 01 20`).
- `Crc`, whose name suggests it is a Cyclic Redundancy Check, so a sort of error-detecting code for the message body, but by reverse engineering and sniffing messages, this field seems to be fixed as well, but it is different for every couple of messages.
- `Seq`, which is a sequence number that is incremented every message, but the protocol does not include any acknowledgment mechanisms to resend lost messages.
- `Frame`, which is used in case of fragmentation, i.e. when the message is bigger than the MTU (Maximum Transmission Unit), which is usually set to 1500 bytes. In this case, the body of the message is split into multiple frames, each one having an identical header, apart from the `Frame` field, which is incremented each frame, starting from 0.

In Listing 9.15, you can see, as an example, the header of a message sent from the frontend process to the control. The resulting payload is 20 bytes long, the first two double words are the BwpType and the Crc, then the next quad-word is the Seq number and finally, the last double word is the Frame number.

Listing 9.15: Header of a Slate message

```
00000000  00 00 01 20  24 5d 06 67  | BwpType, Crc
00000008  00 00 00 00  00 00 00 b6  | Seq
00000010  00 00 00 00               | Frame
```

The body of the message is encoded in the same way, for example in Listing 9.16 you can see part of the structure of messages sent by the frontend process to control, while in Listing 9.17 you can see a real message.

Listing 9.16: `FrontendToControl` structure definition

```
*************************************************************
* Name: slate.FrontendToControl                             *
* Struct:                                                   *
*   +   0x0    0x1 bool AppReboot                           *
*   +   0x1    0x1 bool TiltToStowed                        *
*   +   0x4    0x4 uint StartDishCableTestRequests          *
*   +   0x8    0x1 bool IfLoopbackTest                      *
```

```
*   +  0x10    0x8 longlong Now                                *
*   +  0x18    0x8 longlong StarlinkWifiLastConnected          *
[...]
```

Listing 9.17: Body of a Slate message

```
00000000  00 00 01 20  24 5d 06 67  | Header
00000008  00 00 00 00  00 00 00 b6  | Header
00000010  00 00 00 00  00 00 00 00  | Header, AppReboot
00000018  00 00 00 00  00 00 00 00  | TiltToStowed, StartDishCableTestRequests
00000020  00 00 00 00  00 00 00 00  | IfLoopbackTest, Now (High)
00000028  64 D4 D5 5A  00 00 00 00  | Now (Low), StarlinkWifiLastConnected (High)
00000030  64 D4 D4 7A               | StarlinkWifiLastConnected (Low)
```

If one clicks the "Tilt to Stowed" button on Starlink's application, or the web interface, or directly sends the gRPC message, the process would send the next Slate message with the `TiltToStowed` set to `true`, and the control process would process the user request by actually tilting the antenna. As you may have noticed, offsets and sizes of structure fields in Go do not match the ones shown in the payload. This is because Go handles some datatypes and struct field alignment differently from C. As an example, in Go a `bool` only takes 1 byte, while in C it takes 4 bytes, and padding between structure fields (alignment) has been disabled (structure packing) to be more memory efficient while transferring data.

Thanks to this information we would already be able to implement a message decoder (which we did), but this would only work for the communication between the frontend process and control. And to also be able to decode other communications we would need to manually reverse engineer every other binary to find out the structure of messages, perhaps without even finding field names, but as explained in Section 7.3, it is very hard to get useful information from C++ binaries. In Listing 9.18 you can see how we were parsing the header of Slate messages in Python, using the `ctypes` package.

Listing 9.18: Parsing of the Slate header in Python

```python
class DumpableStructure(BigEndianStructure):
    def dump(self):
        for key, dtype in self._fields_:
            print(f"{key:<30}: {getattr(self, key)}")

class SlateHeader(DumpableStructure):
    _pack_ = 1
    _fields_ = [
        ('BwpType', c_uint32),
        ('Crc', c_uint32),
        ('Seq', c_longlong),
        ('Frame', c_uint32)
    ]
```

Then, to understand how the protocol is handled in C++ binaries, we tried looking for some fields of the structures we knew (the ones of the frontend process) in the control process, which should have them in order to decode those incoming messages. We couldn't find them in the binary, but we found something way better, in the folder `/sx/local/runtime/common` there are a set of configuration files, such as `fontend_to_control`, which contain the structure of every message exchanged by processes. A snippet of `frontend_to_control` is shown in Listing 9.19.

Listing 9.19: Protocol definition for messages from frontend to control

```
# Slate share message from gRPC frontend process to main control process.
app_reboot                          bool
tilt_to_stowed                      bool
start_dish_cable_test_requests      uint32
if_loopback_test.enabled            bool
now                                 int64
starlink_wifi_last_connected        int64
# [...]
```

With this, it is much easier to implement a more generic decoder that parses these protocol definitions and decodes messages accordingly. Such a tool has been developed and will be discussed in the next section.

9.2.1 Slate sniffer/injector

Slate messages are sent very fast, thus it is hard to understand what is happening without a proper visualization tool. That is why we've implemented the Slate sniffer, a tool that sniffs for UDP packets on the loopback interface and decodes them on the fly, by highlighting differences between consecutive messages. The first version was only designed for the communication between the frontend process and control because that was the only protocol definition we could get by reverse engineering the binaries. In Figure 9.1 you can see a screenshot of the tool. It was very basic and later on, we also found out that the decoding of messages was not working properly due to the differences in structure fields from Go to C, explained in the previous section.

Attribute / Time ->	0	-1	-2	-3	-4	-5
BwpType	288	288	288	288	288	288
Crc	610076263	610076263	610076263	610076263	610076263	610076263
Seq	536	534	532	530	528	526
Frame	0	0	0	0	0	0
AppReboot	0	0	0	0	0	0
TiltToStowed	0	0	0	0	0	0
StartDishCableTestRequests	0	0	0	0	0	0
IfLoopbackTest	0	0	0	0	0	0
Now	0	0	0	0	0	0

Figure 9.1: Screenshot of the Slate sniffer version 1

After the protocol definition files were found, we implemented a second version, much more elaborate, that would be able to decode messages exchanged between any couple of processes and present them in a more readable way. In Figure 9.2 you can see the overall architecture of the new tool, which we'll describe in more detail. This tool has been implemented for the emulated environment, but we designed it by keeping in mind that it would need to work also on the real dish. For this reason, most of the work is done in the Sniffer, which is not in the device, and all the communications between the sniffer and the dish happen through SSH.

The first component that is used when starting the sniffer, is the protocol definition parser, which will parse configuration files:

- `service_directory` to know which "services" (i.e. message definitions) are available and on which UDP ports they will communicate.
- `[P1]_to_[P2]` for every available couple of processes `P1` and `P2`, to know the format of messages that will be exchanged.

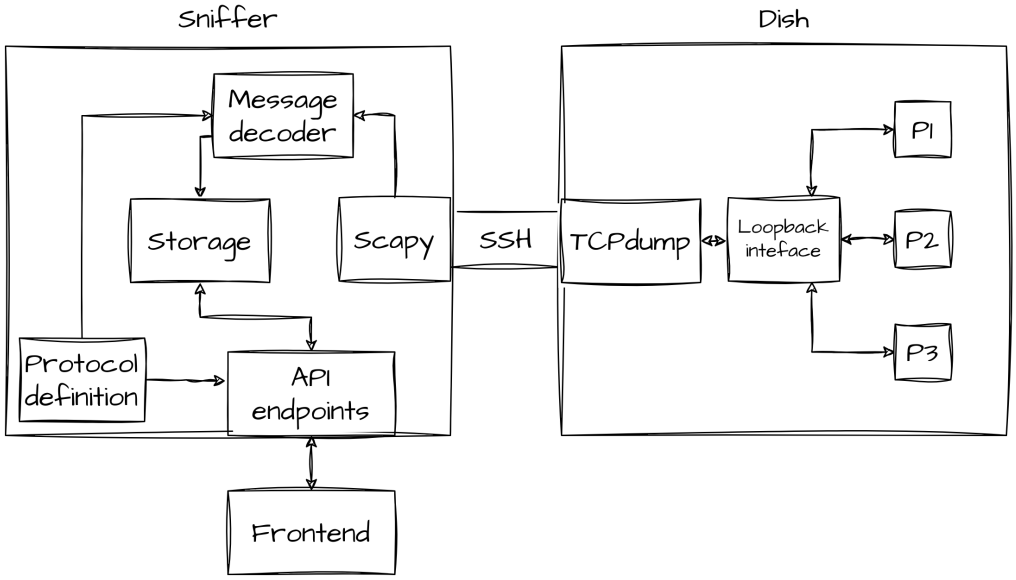


Figure 9.2: Architecture of the Slate sniffer version 2

This component will create `SlateMessageParser` objects, that will later be used to decode messages. The decoding is made using the `struct` python package.

After this, `TCPdump` will be launched on the dish, through SSH, and will listen on the loopback interface, only capturing UDP packets that have as the destination port one of the ones found by the parser. The output of `TCPdump` is piped to `Scapy` which will decode the packet, understand which service it comes from, by reading the destination port, and will then extract the UDP payload and pass it to the message decoder. Listing 9.20 shows a simplified version of it, that does not take into account message fragmentation.

Listing 9.20: Simplified version of the sniffer code

```
for packet in reader:
    try:
        if packet[UDP]:
            port = packet[UDP].dport
            if port in self.slates:
                payload = bytes(packet[UDP].payload)
                slate: Slate = self.slates.get(port)
                message = slate.parse_message(curr)
                handler(slate.service.name, message)
    except Exception as e:
        print(e, file=sys.stderr)
```

When a message is parsed correctly, it will be stored in the Storage component, which in this case is a simple in-memory database, only holding recent messages (configurable, based on available memory). On top of all this, there is a Flask server, exposing some APIs, to know which services are available, to know the schema of a message and, of course, to fetch messages. We also implemented a front end, as a simple web interface, which is shown in Figure 9.3. From the current front end, it is possible to see messages in real-time, spotting differences thanks to the highlighted changes, selecting which fields to show and filtering or ordering them. The frontend module is easily replaceable thanks to the exposed APIs, thus more complex interfaces can be integrated into the sniffer, to inspect the acquired data in more ways.

Search:

#	BwpType	Crc	Seq	Frame	timestamp	runtime_second	mac_software_version	active_beam_pairs
0	288	2077763449	18876	0	1375715244566127400	1184070144	54	0
1	288	2077763449	18877	0	1375715245566834200	1184070656	54	0
2	288	2077763449	18878	0	1375715246567356700	1184071168	54	0
3	288	2077763449	18879	0	1375715247566792400	1184071680	54	0
4	288	2077763449	18880	0	1375715248566717700	1184072192	54	0
5	288	2077763449	18881	0	1375715249566626600	1184072704	54	0
6	288	2077763449	18882	0	1375715250566787800	1184073216	54	0

Figure 9.3: Screenshot of the current frontend interface for the sniffer

After having a way to see messages, we thought it would be interesting to be able to inject custom messages, by editing the ones we are receiving or by creating new ones from scratch. For this reason, we implemented the Slate injector, which shares most of the codebase with the sniffer. The architecture of this tool is shown in Figure 9.4. Messages, to be received by processes, need to come from the loopback interface, thus we cannot send them directly from the Injector. This is why the injector will start a `socat` server on the dish, which will listen for UDP messages on the "external" network interface, and will then forward them to the right UDP port, by changing the source address to localhost.[2] Some API endpoints have been implemented to be able to inject messages from the front end and the current interface lets you edit and send a message or create a new one, Figure 9.5 shows a screenshot of what the interface looks like.

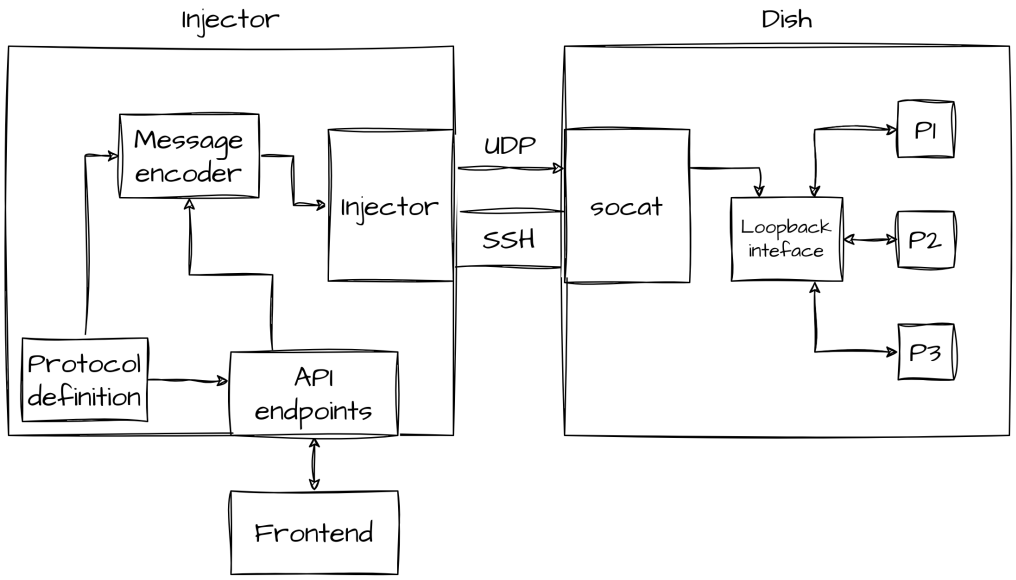


Figure 9.4: Architecture of the Slate injector

Being able to inspect messages between processes helped us a lot in understanding what each process does without being able to fully reverse-engineer them. Additionally, having the protocol definition and an easy way to inject messages, a natural development of the project is to fuzz the protocol, which will be tackled in Section 10.2. The full code of the Slate sniffer, injector and fuzzer is available on Quarkslab's GitHub [27].

[2] On production hardware, incoming UDP traffic is dropped by the firewall, so you'll need to either delete some `iptables` rules or use TCP between the injector and `socat`.

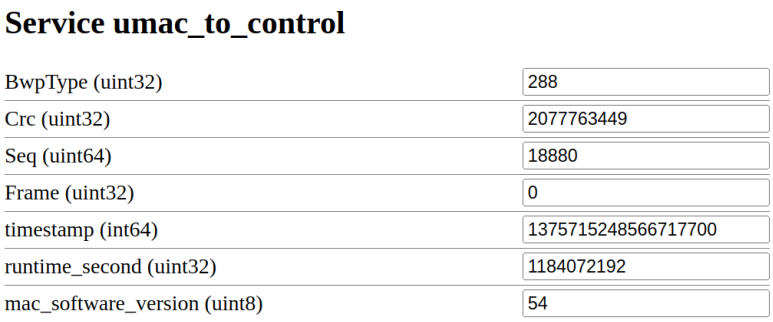


Figure 9.5: Screenshot of the current frontend for the injector

10 Fuzzing

In the last part of my research period, we identified which parts of the software had been analyzed enough and were more suitable for fuzz testing. To find a good fuzzable target, we had to take into account multiple aspects:

- The possible attack vector in the case a bug was found:
  - Can the bug be triggered by an authenticated user?
  - Does the attacker need to be connected to the wifi network of the dish?
  - Can the bug be triggered directly from the Internet?
  - Does the attacker need to already have access to the dish? In this case, the bug could be used for lateral movement inside the dish or privilege escalation.
- What is the impact of a possible bug in the target?
- How easily fuzzable is the target, and which kind of fuzzing is the most suitable for the target:
  - How is the input provided?
  - How isolated can the program be run? If a program makes use of many hardware peripherals and/or interacts with other components of the runtime, it will be hard to fuzz it in an isolated environment.
  - How deeply have we analyzed the target to know its internal workings?

10.1 Side-load Software update

As we have seen in Subsection 9.1.2, from the internal network the dish creates, it is possible to trigger a software update by sending a `SoftwareUpdateRequest` to the frontend process, through gRPC. This is an interesting request because it is the only one that does not look like it is meant for the user and does not require authentication. Furthermore, the input of this request is the update bundle, which can be very big, while inputs for other requests are often empty or very simple. The update bundle has to be split into chunks before being sent to the dish, Listing 10.1 shows a Python script that sends this message.

Listing 10.1: Python script that sends a software update bundle to the dish

```
CHUNK_SIZE = 16384

def update(data):
    channel = grpc.insecure_channel('192.168.100.1:9200')
    stub = device_pb2_grpc.DeviceStub(channel)

    stream_id = int.from_bytes(os.urandom(4), byteorder='little')

    for i in range(0, len(data), CHUNK_SIZE):
        chunk = data[i:min(len(data), i + CHUNK_SIZE)]
        request = device_pb2.Request(id = 1, epoch_id = 1, target_id = "unknown",
            software_update = common_pb2.SoftwareUpdateRequest(
                stream_id = stream_id,
                data = chunk,
                open = i == 0,
                close = i + CHUNK_SIZE >= len(data)
            )
        )
```

Every message needs to have the same `stream_id`, which can be randomly generated, then the first message has the `open` flag, while the last one has the `close` flag and all the ones in between don't have any of them. The receiver of the message is the frontend process, which will save the update bundle in a temporary folder, without reading the content of it, so without performing any kind of input sanitization, it will just check that the size of the bundle does not reach an hardcoded threshold. After that, the frontend process will notify the control process that a sideload update is ready to be applied, through a Slate message (see Section 9.2), and the control process will do the same for the software update process. Once the latter receives the message, the update is ready to be started, Figure 10.1 shows the overall flow of messages and actions the `SoftwareUpdateRequest` triggers.

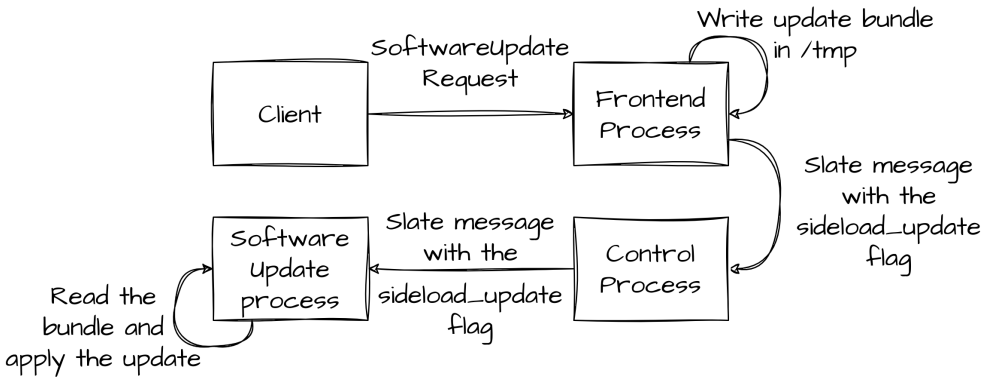


Figure 10.1: Sideload update message flow

After the software update process is notified that a software update bundle is ready to be applied, the update procedure starts. From this moment, there is no difference between this kind of software update and the standard one, which is downloading the update bundle from Starlink's backend. The update bundle is an `sxverity` image (see Section 4.2.2), which will be verified by the program with the same name and the inner `rom1fs` filesystem will be mounted. Once mounted, the software update process will look for partition images in the mount point. Every partition image will also have a SHA512 hashsum for additional integrity verification. Finally, each available partition image will be flashed on the corresponding `/dev/blk/other/*` eMMC logical partition (see Subsection 7.2.2). Figure 10.2 summarizes this with a diagram.

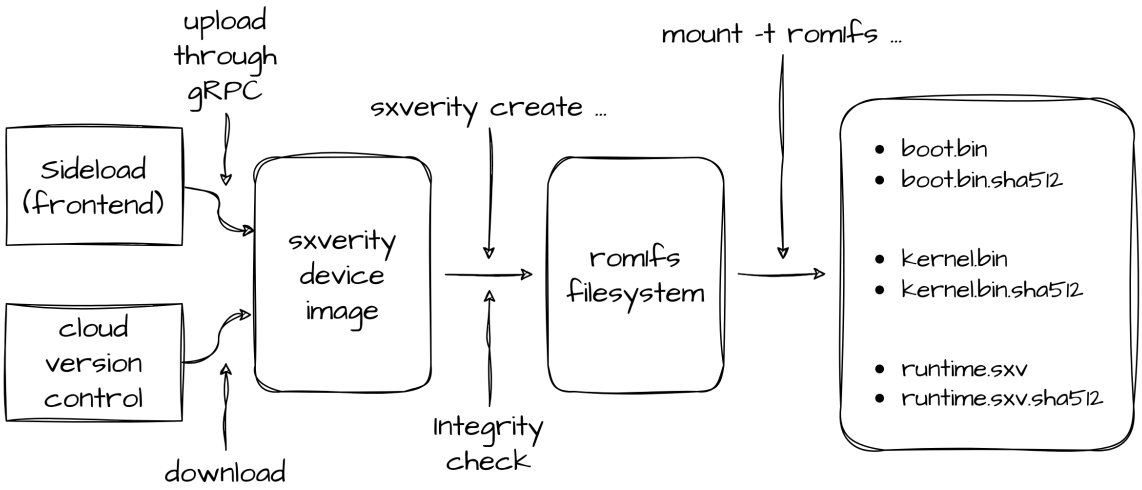


Figure 10.2: Software update bundle path

The update bundle is not accessed directly by the software update process, so the first process reading the content of the provided input is `sxverity`. Thus any fuzz test can be performed directly on that binary, skipping all previous steps. We've already analyzed how the verification of `sxverity` works in Section 4.2.2. As you can see in Figure 10.3, the fuzzable code inside `sxverity` is very limited because the signature verification is made by a library, which is out of scope, and anything happening after a successful signature verification is to be considered unreachable for us because if we can reach that state, it means we were able to craft an update package that would be flashed so we don't need to find other bugs there.

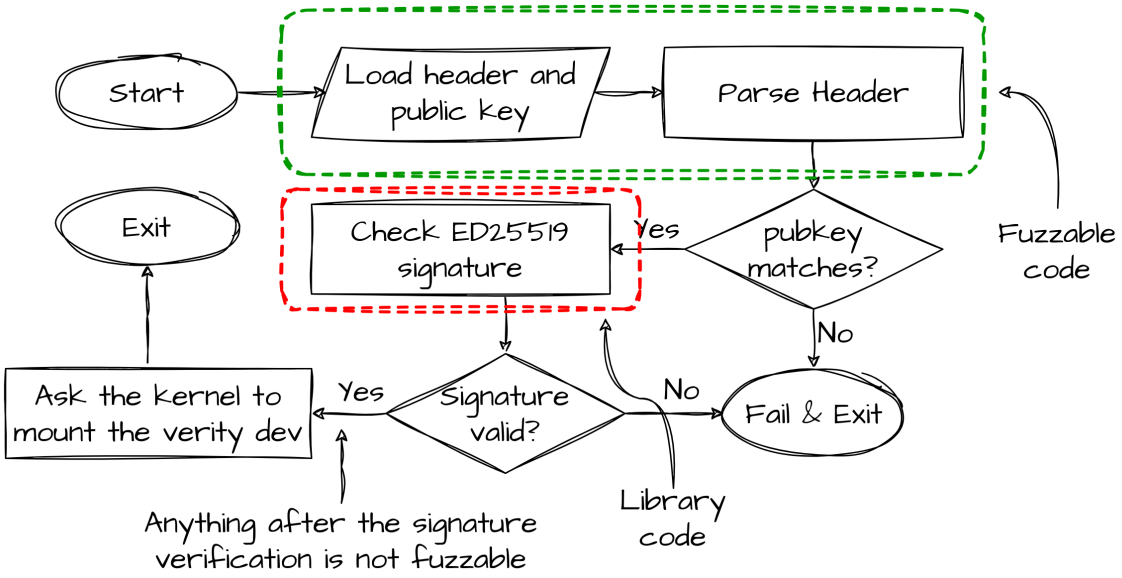


Figure 10.3: Fuzzable code inside `sxverity`

The only part of the input that will be parsed by the code under test is the header of the `sxverity` image, thus that will be the only part of the input that the fuzzer will mutate. Since that part of the program can be executed completely in complete isolation, we've fuzzed it inside `unicorn`, a lightweight CPU emulator that can be instructed from Python, by using its bindings. The first step was being able to emulate the code we want to test and setting up the harness for our fuzzer, which includes:

- Loading the binary.

- Identifying a good starting point in the code, in which it is easy to set up the whole environment, such as placing the input in the right memory location and setting up every other memory structure that will be used by the code under testing. As an example, Listing 10.2 shows how the input is placed in memory and how registers holding the addresses to the input locations are set.

Listing 10.2: Function `place_input_cb` of the harness

```
def place_input_cb(mu: Uc, content: bytes, persistent_round, data):
    content_size = len(content)

    if content_size < INPUT_MIN_LEN:
        return False

    pubkey, header = key_and_header_from_data(content)

    # write data in memory
    mu.mem_write(PUBKEY_ADDR, pubkey)
    mu.mem_write(HEADER_ADDR, header)

    # prepare function arguments
    mu.reg_write(UC_ARM64_REG_X2, PUBKEY_ADDR) # pubkey address
    mu.reg_write(UC_ARM64_REG_X3, 0x40) # nblocks
    mu.reg_write(UC_ARM64_REG_X4, HEADER_ADDR) # header buffer address

    return True
```

- Identifying function calls, hooking them and emulating them in Python, so that we do not spend time testing library code, nor do we have to load them in memory and handle dynamically loaded libraries. As an example, Listing 10.3 shows how the libc function `memcpy` is hooked and emulated.

Listing 10.3: `memcpy` hook in the harness

```
if address == MEMCPY_ADDR:
    # read arguments from registers
    dest = mu.reg_read(UC_ARM64_REG_X0)
    src = mu.reg_read(UC_ARM64_REG_X1)
    n = mu.reg_read(UC_ARM64_REG_X2)

    # read the data from src
    data = mu.mem_read(src, n)
    # write data in dst
    mu.mem_write(dest, bytes(data))
    # return the address of dest
    mu.reg_write(UC_ARM64_REG_X0, dest)
    # jump to the return address
    lr = mu.reg_read(UC_ARM64_REG_LR)
    mu.reg_write(UC_ARM64_REG_PC, lr)
```

- Identifying an ending point, which has to be a point in the program in which we stop the emulation because the run was successful (no bugs).

The nicest thing about using Unicorn, other than being pretty easy to configure and instruct, is the flawless support with `AFL++` (American Fuzzy Lop plus plus) which we used as fuzzer. `AFL++` working with `Unicorn` can detect crashes and most importantly, gather coverage information, in a transparent manner, so that it can perform coverage-guided mutations. Setting up the fuzzer with Unicorn, is pretty straightforward, as you can see in Listing 10.4. The fuzzer also needs some initial test cases, called seeds, for that we've used some valid headers, taken from `sxverity` images found in the dish, and some randomly generated headers.

Listing 10.4: AFL++ set up inside Unicorn

```
# set the starting address
mu.reg_write(UC_ARM64_REG_PC, START_ADDRESS)
# start the fuzzer
uc_afl_fuzz(
    uc=mu,
    input_file="@@",
    place_input_callback=place_input_cb,
    exits=[END_ADDRESS]
)
```

10.1.1 Results

The fuzzer ran for around 24 hours, performing more than one million executions, but unfortunately, no crash was recorded. This was expected since the tested codebase was very limited and the structure of the input was very simple, not having complex data structures or variable length fields, the most common memory-related bugs are avoided.

10.2 IPC fuzzer

The other component we tested with fuzzing was Inter-Process Communication (IPC) - which was deeply analyzed in Section 9.2 - since we already had developed a set of tools to analyze and tamper with this communication. In this case, we are not going to fuzz a single binary, but rather the whole set of processes that form the runtime of the device, since every one of them is using Slate messages to communicate. The fuzzing approach was completely different from the one we used for `sxverity`, which was gray-box fuzzing, because:

- The codebase we are trying to test is enormous
- We weren't able to exactly identify the code that handles slate messages in every binary and, more importantly, bugs can be also found outside this code, because of some inconsistencies in the program state caused by wrongly interpreted inputs.
- Binaries need to run in a dish-like environment because they continuously interact with other components of the system, most of them don't even run in our emulated environment.
- Also recording coverage would have been challenging, because for that we need to instruct the binaries since we don't have the source code to recompile them, and the fuzzer would need to be run on the dish.

For the above reasons, we used black-box fuzzing, without coverage-guided mutations, which is usually called "dumb" fuzzing. `Boofuzz` was used as fuzzer, it is a simple and easy-to-use fuzzer specifically designed for network protocols, which was a perfect fit for what we were looking for. `Boofuzz` does not generate input in a completely random way because you are giving it the protocol definition to be used in the communication, with the possibility of defining sequences of messages using a finite state machine. In our case, every message was disconnected from the others (apart from the sequence number), so defining the format of messages was enough. The fuzzer will then mutate every field of the message, by trying some values that could trigger a bug, e.g. for an `int32` the fuzzer will try values such as `{0, 1, -1, INT_MAX, -INT_MAX, ...}`. As an example, Listing 10.5 shows how some fields of Slate messages are "translated" to Boofuzz protocol definition.

Listing 10.5: Boofuzz protocol definition for some data types

```
if param.dtype.name == "BOOL":
    return Simple(
        name=param.name,
        default_value=b"\x00\x00\x00\x00",
        fuzz_values=[b"\x00\x00\x00\x00", b"\x00\x00\x00\x01"],
    )
if param.dtype.name == "INT8" or param.dtype.name == "UINT8":
```

```
    return Byte(name=param.name)
if param.dtype.name == "INT32" or param.dtype.name == "UINT32" or param.dtype.name == "
    FLOAT":
    return DWord(name=param.name)
```

Every data type used in the Slate message protocol could be encoded using Boofuzz's standard types, apart from the Sequence number, which needs to store an internal state to increment itself every iteration, you can see its implementation in Listing 10.6. Static fields such as the `Bwptype` and `Crc` can be encoded using the `Static` type from Boofuzz.

Listing 10.6: `SequenceNumber` data type implementation

```
class SequenceNumber(Fuzzable):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs, fuzzable=False, default_value=0)
        self._curr = 0

    def encode(self, value: bytes = None, mutation_context=None) -> bytes:
        curr = self._curr
        self._curr += 1
        return int.to_bytes(curr, length=8, byteorder="big")
```

Once the message structure has been defined, the fuzzer can use the code from the Slate injector (see Section 9.2.1) to send the messages. The only component that needs to be implemented at this point is something that can detect if a program has crashed after a message was sent. At first, we were issuing the `pgrep` command through SSH, but this was adding an overhead that was slowing the fuzzer. So we've implemented a simple script that runs on the dish, opening a TCP socket and waiting for a connection which will then be used to directly communicate with the fuzzer. The part of the process monitor that will run on the client (fuzzer machine) can be integrated into the fuzzer, by inheriting Boofuzz's `BaseMonitor` and implementing its methods, such as `alive` (check if the target process is still alive) and `restart_target` (restart the target process). The resulting architecture is shown in Figure 10.4.

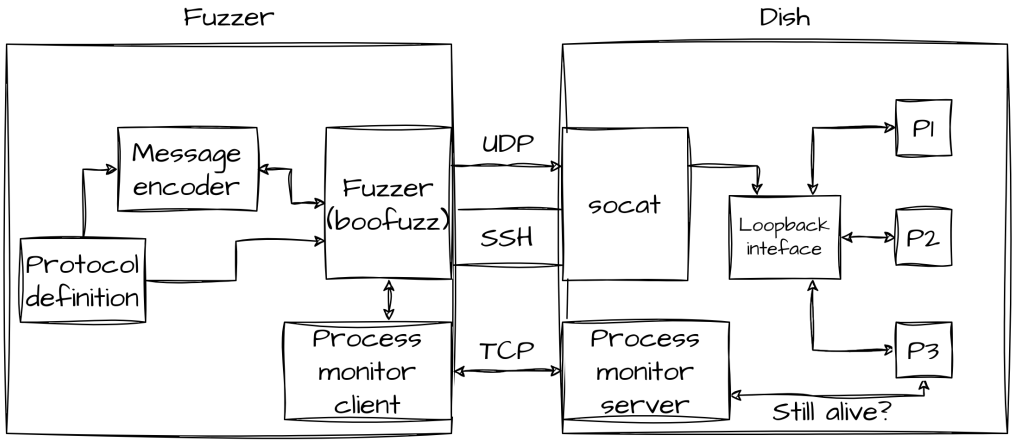


Figure 10.4: Architecture of the fuzzer codebase

10.2.1 Results

Some crashes were found by fuzzing the communication `control_to_frontend`, so messages that are supposedly coming from the control process and going to the frontend process. But none of them appeared to be exploitable in ways other than simply crashing the program, causing a Denial of Service for the frontend applications. This is because the frontend process is a Go binary and the Go runtime is the piece of software that makes the process crash (through the `panic` function) because it detects that something fishy is going on.

As an example, the following are the details of one of these crashes. In Listing 10.7 you can see part of the stack trace produced by the Go runtime upon the crash, from which you can understand that the crash is caused by the function `UpdateObstructionMap`, which tries to allocate too much memory.

Listing 10.7: Crash synopsis from the frontend process

```
fatal error: runtime: out of memory

goroutine 5 [running]:
[...]
runtime.(*mheap).alloc(0x50468000?, 0x28234?, 0xd0?)
[...]
main.(*DishControl).UpdateObstructionMap(0x40003be000, {0x7a90f8?, 0x4000580380?})
[...]
```

By inspecting further this function, we understood how the obstruction map is transferred to the frontend process. First of all, the obstruction map is a 3D map of the sky above the antenna, indicating whether the antenna has a clear view of the sky or is obstructed by an obstacle such as a tree or other buildings, that the user can see from the frontend applications. This map is not produced by the frontend process, thus it has to be sent to it through Slate messages.

Listing 10.8: Part of message definition from `control_to_frontend`

```
obstruction_map.config.num_rows                    uint32
obstruction_map.config.num_cols                    uint32
obstruction_map.current.obstructed                 bool
obstruction_map.current.index                      uint32
```

In Listing 10.8 you can see part of the message structure definition that carries information about the obstruction map. The obstruction map is represented in memory as a matrix, in which each point can be obstructed or not. The control process sends this information by sending one Slate message for each point in the matrix, by setting the right `index` and setting `obstructed` to `true` or `false`. The size of the matrix is not fixed, and its dimensions can be set by the control process by using the `num_rows` and `num_cols` fields in the message. This is where the bug resides, in fact when sending big values in these two fields, the program tries to allocate enough memory for the matrix and panics for this reason.

Listing 10.9: Decompiled code that handles the size of the obstruction map

```
1 len = ObstructionMapConfigNumCols * ObstructionMapConfigNumRows;
2 if (len == (this->obstructionMap).snr.__count)
3     goto LAB_0050b7f4;
4 (this->obstructionMap).numRows = ObstructionMapConfigNumRows;
5 (this->obstructionMap).numCols = ObstructionMapConfigNumCols;
6 puVar5 = runtime.makeslice(&datatype.Float32.float32,len,len);
```

Listing 10.9 shows the decompiled and annotated code of the frontend binary which handles the size of the obstruction upon the reception of a slate message. Line 1 computes the size of the matrix and Line 2 compares it with the current size of the matrix the program has in memory, if the two differ then the dimensions are updated in the internal memory structure on Lines 4 and 5, and then the new matrix is allocated using the `makeslice` method of the Go runtime on Line 6. As you can see, no checks are performed on the size it is being allocated, nor on the result of the multiplication between the two given dimensions. This would be very dangerous in C, but the Go runtime handles all the corner cases automatically, by checking that the size of the asked memory is positive and not too big. The Go runtime also checks every array access, otherwise, an arbitrary write would probably be possible by playing with the index and the size of the matrix.

Note that this bug can only be triggered by sending crafted UDP packets to a service bound to localhost only. Therefore it is not possible to trigger it from an external network. Additionally, the

`iptables` configuration of the UT filters out incoming UDP packets, so spoofing packets with a localhost source IP would not work either. Therefore we did not consider this a vulnerability but merely a bug.

After we implemented the fuzzer and used it in the emulator, we were provided a rooted UT by Starlink, then we confirmed the presence of the aforementioned bug on a real device and fuzzed some more processes that weren't working in the emulator.

11 Conclusion

A lighter, less detailed, version of this work can be found on Quarkslab's blog [26], while the discussed tools can be found on Quarkslab's GitHub [27].

This work and the tools we have published are meant to be reused for further research on Starlink's User Terminal. Unfortunately, due to some technical issues and time constraints, we did not manage to fully inspect the network stack and protocols used in the satellite communications, but hopefully, this knowledge base of the higher-level management function of the runtime can be used in the future to assist in that effort. I encourage research on this topic because SpaceX's security team is there to help you and they also offer some juicy bounties [15].

Further work on this topic includes (but is not limited to):

- Fuzz the gRPC communications using `libprotobuf-mutator` [9].
- Analyze better the content of Slate messages to understand their meanings and perhaps find logic bugs.
- Investigating how the network stack is implemented:
  - How the traffic is forwarded from the Linux's network interface to the RF hardware.
  - How the system interacts with the RF hardware.
  - How the Physical and Data Link layers of the RF communications are implemented.
- And many other things, the firmware is huge and we've probably just scratched the surface here.

Bibliography

[1] Arm trusted firmware-a. https://github.com/ARM-software/arm-trusted-firmware. last access 24/07/2023.

[2] Device tree what it is. https://elinux.org/Device_Tree_What_It_Is. last access 24/07/2023.

[3] DishyPowa. https://dishypowa.com/. last access 19/07/2023.

[4] dm-verity. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/verity.html. last access 24/07/2023.

[5] Embedded device command line partition parsing. https://www.kernel.org/doc/Documentation/block/cmdline-partition.txt. last access 27/07/2023.

[6] Fault injection attacks. https://www.appluslaboratories.com/global/en/news/publications/new-fault-injection-attacks. last access 28/07/2023.

[7] Gateways locations. https://starlinkinsider.com/starlink-gateway-locations/. last access 19/07/2023.

[8] Implementing dm-verity. https://source.android.com/docs/security/features/verifiedboot/dm-verity. last access 26/07/2023.

[9] libprotobuf-mutator. https://github.com/google/libprotobuf-mutator. last access 28/08/2023.

[10] Ookla. https://www.ookla.com/articles/starlink-hughesnet-viasat-performance-q3-2022. last access 18/07/2023.

[11] pbtk - reverse engineering protobuf apps. https://github.com/marin-m/pbtk. last access 08/08/2023.

[12] Russia tests secretive weapon to target spacex's starlink in ukraine. https://www.washingtonpost.com/national-security/2023/04/18/discord-leaks-starlink-ukraine/. last access 28/08/2023.

[13] Satellite map. https://satellitemap.space/?constellation=starlink. last access 18/07/2023.

[14] Sd card specification. https://forums.futura-sciences.com/attachments/electronique/61778d1227430362-carte-sd-courant-max-sd_sdio_specsv1.pdf. last access 20/07/2023.

[15] Spacex bugcrowd campaign. https://bugcrowd.com/spacex. last access 28/08/2023.

[16] Spacex's linux modifications. https://github.com/SpaceExplorationTechnologies/linux. last access 24/07/2023.

[17] Starlink. https://www.starlink.com/. last access 18/07/2023.

[18] Starlink cli. https://github.com/starlink-community/starlink-cli. last access 08/08/2023.

[19] Starlink grpc tools. https://github.com/sparky8512/starlink-grpc-tools/. last access 08/08/2023.

[20] Starlink satellite internet. https://satoms.com/starlink-satellite-internet/. last access 19/07/2023.

[21] Starlink satellite services in ukraine. https://en.wikipedia.org/wiki/Starlink_satellite_services_in_Ukraine. last access 28/08/2023.

[22] Starlink's tweet about the number of subscribers in may 2023. https://twitter.com/Starlink/status/1654673695007457280. last access 28/08/2023.

[23] Ken Keiter. Starlink Teardown: DISHY DESTROYED! https://youtu.be/iOmdQnIlnRo. last access 19/07/2023.

[24] MikeOnSpace. Starlink Dish TEARDOWN! - Part 1. https://youtu.be/QudtSo5tpLk. last access 19/07/2023.

[25] Carlo Ramponi. Bindiff-ghidra importer. https://github.com/CarloRamponi/bindiff-ghidra-importer. last access 03/08/2023.

[26] Carlo Ramponi. Diving into starlink's user terminal firmware. https://blog.quarkslab.com/starlink.html. last access 29/08/2023.

[27] Carlo Ramponi. Starlink reverse-engineering scripts. https://github.com/quarkslab/starlink-tools/. last access 29/08/2023.

[28] COSIC research group. Dumping and extracting the spacex starlink user terminal firmware. https://www.esat.kuleuven.be/cosic/blog/dumping-and-extracting-the-spacex-starlink-user-terminal-firmware/. last access 20/07/2023.

[29] COSIC research group. Starlink user terminal modchip. https://github.com/KULeuven-COSIC/Starlink-FI. last access 31/07/2023.

[30] Anatoly Shalyto, Nikita Shamgunov, and Georgy Korneev. State machine design pattern. *Proc. of the 4th International Conference on .NET Technologies*, pages 51–57, 2006.

[31] SpaceX. U-boot. https://github.com/SpaceExplorationTechnologies/u-boot. last access 20/07/2023.

[32] The Signal Path. Starlink Dish Phased Array Design, Architecture & RF In-depth Analysis. https://youtu.be/h6MfM8EFkGg. last access 19/07/2023.

Appendix A Attachments

A.1 Memory blocks extractor

```python
import sys
import os

# Every element of this array is a tuple:
# (NAME, SIZE (Kb)), Addresses start from 0.
MEMORY_LAYOUT = [
    (f"bootfip{i}",             1024) for i in range(4)
] + [
    ("bootterm1",               512),
    ("bootmask1",               512),
    ("bootterm2",               512),
    ("bootmask2",               512),
    ("fip_a.0",                 1024),
    ("fip_b.0",                 1024),
    ("fip_a.1",                 1024),
    ("fip_b.1",                 1024),
    ("fipterm1",                1024),
    ("fipterm2",                1024),
    ("unused",                  2816),
    ("per_vehicle_config_a",    128),
    ("per_vehicle_config_b",    128),
    ("mtdoops",                 192),
    ("version_a",               128),
    ("version_b",               128),
    ("secrets_a",               128),
    ("secrets_b",               128),
    ("sxid",                    320),
    ("linux_a",                 32 * 1024),
    ("linux_b",                 32 * 1024),
    ("sx_a",                    24 * 1024),
    ("sx_b",                    24 * 1024),
    ("edr",                     151367),
    ("dish_config",             32 * 1024)
]

def main():
    input_file = sys.argv[1]
    output_folder = sys.argv[2]

    os.mkdir(output_folder)
    with open(input_file, "rb") as file_in:
        handle_file(file_in, output_folder)

def handle_file(file_in, output_folder):
    for file in MEMORY_LAYOUT:
        with open(os.path.join(output_folder, file[0]), "wb") as file_out:
            file_out.write(file_in.read(file[1] * 1024))

if __name__ == "__main__":
    main()
```

A.2 ECC data stripper

```python
# this script removes ECC data from a file
import sys

NPAR = 32
ECC_BLOCK_SIZE = 255
ECC_MD5_LEN = 16
ECC_EXTENSION = b"ecc"
ECC_FILE_MAGIC = b"SXECCv"
ECC_FILE_VERSION = b"1"
ECC_FILE_MAGIC_LEN = len(ECC_FILE_MAGIC)
ECC_FILE_HEADER_LEN = ECC_FILE_MAGIC_LEN + len(ECC_FILE_VERSION)
ECC_FILE_FOOTER_LEN = 1 + 4 + ECC_MD5_LEN + NPAR
ECC_DAT_SIZE = ECC_BLOCK_SIZE - NPAR - 1
ECC_FIRST_DAT_SIZE = ECC_BLOCK_SIZE - ECC_FILE_HEADER_LEN - NPAR - 1
ECC_BLOCK_TYPE_DATA = ord('*')
ECC_BLOCK_TYPE_LAST = ord('$')
ECC_BLOCK_TYPE_FOOTER = ord('!')
ECC_BLOCK_TYPE_INDEX = ECC_BLOCK_SIZE - NPAR - 1

def unecc(input_file, output_file):
    with open(input_file, "rb") as file_in:
        with open(output_file, "wb") as file_out:
            size = 0
            # read the first block
            block = file_in.read(ECC_BLOCK_SIZE)
            file_out.write(handle_first_block(block))
            size += ECC_FIRST_DAT_SIZE

            # read blocks until we reach the last one
            while block[ECC_BLOCK_TYPE_INDEX] == ECC_BLOCK_TYPE_DATA:
                block = file_in.read(ECC_BLOCK_SIZE)
                # if it's the last block, read the footer to compute the padding length
                if block[ECC_BLOCK_TYPE_INDEX] == ECC_BLOCK_TYPE_LAST:
                    footer = file_in.read(ECC_FILE_FOOTER_LEN)
                    payload_size = int.from_bytes(footer[1:5], "big")
                    last_block_size = payload_size - size
                    file_out.write(handle_block(block)[:last_block_size])
                else:
                    file_out.write(handle_block(block))
                    size += ECC_DAT_SIZE

def handle_first_block(data):
    assert(data.startswith(ECC_FILE_MAGIC + ECC_FILE_VERSION))
    assert(data[ECC_BLOCK_TYPE_INDEX] == ECC_BLOCK_TYPE_DATA
        or data[ECC_BLOCK_TYPE_INDEX] == ECC_BLOCK_TYPE_LAST)
    # remove magic bytes, version byte and th ECC data
    return data[ECC_FILE_HEADER_LEN:ECC_FILE_HEADER_LEN + ECC_FIRST_DAT_SIZE]

def handle_block(data):
    assert(data[ECC_BLOCK_TYPE_INDEX] == ECC_BLOCK_TYPE_DATA
        or data[ECC_BLOCK_TYPE_INDEX] == ECC_BLOCK_TYPE_LAST)
    return data[:ECC_DAT_SIZE]

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print(f"Usage: python {sys.argv[0]} input_file output_file", file=sys.stderr)
    input_file = sys.argv[1]
    output_file = sys.argv[2]
    unecc(input_file, output_file)
```