RUHR-UNIVERSITÄT BOCHUM

Hacking the Stars: A Fuzzing Based Security Assessment of CubeSat Firmware

Florian Göhler

Master's Thesis – December 22, 2022.
Chair for System Security.

1st Supervisor: Prof. Dr. Thorsten Holz
2nd Supervisor: M.Sc. Johannes Willbold

Abstract

There were never more satellites in space than today and their number is steadily increasing. At the same time, satellites and their infrastructure on the ground progressively shift into the focus of cybercriminals and other threat actors, as was impressively shown in early 2022. This issue is intensified by a lack of security research on the onboard software and firmware of space systems, as past research mostly focused on the security of other components. This is especially true for small CubeSats that are often built with commercial off-the-shelf parts. OPS-SAT provides an example of such a CubeSat. OPS-SAT is an experimental satellite that is currently in space. In this thesis, we assess the security of the OPS-SAT onboard firmware by using fuzzing. Therefore, we implement an emulator for the AVR32 CPU architecture that allows us to rehost the satellite's firmware on a more powerful computer. Next, we will analyze the firmware and define the code segments that will be evaluated in the assessment. Our work will show that rehosting can be used to effectively use fuzzing to identify vulnerabilities in satellite firmware. The results of the fuzzing will show that the OPS-SAT suffers from at least two security vulnerabilities that potentially allow attackers to gain control of the satellites onboard computer.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

22.12.2022	
Datum	Unterschrift

Contents

1 Introduction

The introduction explains the motivation for this thesis. The motivation points out why we consider this topic to be an interesting research area. The introduction also describes the three main contributions of this thesis. In the third section, the introduction details how the thesis is organized.

1.1 Motivation

There were never more satellites in space than today and their number is steadily increasing [Nib21]. In 2020 more than 1200 satellites were sent into space [Nib21]. It is expected that the total number of satellites grows up to 8000 in the year 2024 [CMT20]. Many of these satellites are part of a satellite *constellation* [Kul20]. Constellations consist of multiple satellites that are similar to each other and have the same purpose, for example, to provide communication via satellite phones [Kul21]. The number of individual satellites per constellation varies. As an illustration, there are constellations with 20 planned satellites [Kul21]. On the other hand, the *Starlink* constellation is planned to consist of more than 4000 satellites, with 1600 already being in space [Kul21].

One reason for the increasing number of satellite launches is a drastic decrease in launch costs that happened in the last decade. The cost of satellite lunches was reduced because commercial companies developed more cost-efficient spacecrafts [Jon18]. To give an example of this development, we consider the costs for a *Space Shuttle* launch and a launch of the newer *Falcon 9 plus Dragon*. In 2018, the costs for a *Space Shuttle* launch were about 1.7 billion dollars, while a launch of the *Falcon 9 plus Dragon* only costs about 150 million dollars [Jon18]. Many companies see this situation as a chance to develop new market concepts. Therefore, the number of companies that want to build satellite constellations is increasing every year. Since 2015 more than 100 constellation companies were founded [Kul21]. Combined, these companies plan to start hundreds of satellites every year [Kul20].

Most of the existing and planned satellite constellations consist of *CubeSats* [Kul21]. CubeSats are small satellites that usually fly in a Low Earth Orbit (LEO). A LEO is any orbit with a distance of 350 to 2000 kilometers to Earth's surface [Kul21]. CubeSats are used for various purposes, like weather observation, communication, or scientific measurements. Like every spacecraft, CubeSats face various challenges in space. These challenges include exposure to solar radiation and a limited power supply. Additionally, repairs are impossible if a CubeSat is physically damaged when in space. Because of that, engineers that build CubeSats need to have special skills, like physics, electrical engineering, and applied science [NE19]. Also, software engineering is needed to build a CubeSat. However, experts in spacecraft development are usually no experts in computer security [Fal18]. CubeSat engineers often lack awareness about computer security issues [Fal18]. Additionally, computer security is often not considered important when building a CubeSat, because the mentioned physical challenges are more pressing issues [Fal18] [Fri13]. Because of this, there are also financial constraints that affect the security of space-systems [Fal18].

The development of spacecrafts often requires a complex project-organization, as various possible participants are involved in such projects [Fal18]. Besides scientific stakeholders, like universities, there are commercial companies, vendors and contractors from the private sector, and public space agencies that need to work together to send a satellite into space. Usually, these participants are from multiple countries and each participant has it's own responsibilities during the project [Fal18]. This is relevant for the security of CubeStats, as complex structures increase the risk of security issues [Fal18]. The reason for this is that the responsibility for the security of the satellite is often not clearly assigned to every relevant party [Fal18]. Hence, there could be a situation where one party is not addressing a potential security issue in the satellite's software because they expect another party to do so. However, the second party also expects another party to focus on the security aspects of the satellite, as this responsibility was not assigned to them. In such cases, vulnerabilities may stay unnoticed because no party is explicitly responsible for security-related issues [Fal18]. Another aspect is that there is no international authority or regulation that enforces rules for secure spacecraft development [Fal18]. Therefore, many spacecraft developers only focus the functionality and safety concerns without considering security.

The increasing number of CubeSats makes them an interesting target for cybercriminals [FVS21]. Recent events showed that satellites and the related infrastructure in general are catching the attention of cybercriminals and other threat actors. In early 2022 the KA-SAT satellite network was attacked [Via22]. The attackers used malware to disable multiple thousand satellite modems on the ground [Via22]. These systems could no longer be used for communication and the attack resulted in various side effects. For example, more than 5000 wind power stations in Germany could no longer be accessed remotely [Bun22].

Attacks on satellites are not a new phenomenon and have been reported in the past decades [PM20]. These reports include various types of attacks, like attacks on satellite control systems on the ground and attacks on communication links [Fri13]. As an illustration, there were incidents in which radio signals from Global Positioning System (GPS)-satellites were jammed to prevent receivers on the ground to use GPS [Fri13]. Other reports show that attackers can eavesdrop on communications links between satellites and ground stations[PM20]. There are also reports about incidents, in which attackers were able to gain control of satellites in space [Fri13].

Because of incidents like the ones described above and the general development in the space industry, it is expected that satellites and the related infrastructure will be targeted more frequently by groups with malicious intent in the future [Fal18] [PM20]. One of the main reasons for this assumption is, that attacks on satellites allow thread actors to impact a high number of assets [Fal18]. Besides damages to the satellites themselves, there can be consequences for various industries [Fal18]. CubeSats in particular are vulnerable to various attacks [FVS21].

Despite this situation, there is a lack of research on the security of satellite systems [Fal18]. One reason for this is that the private sector is often not cooperating with researchers who wish to analyze the security of a certain space-system [Fal18]. Because many CubeSats are build with commercial off-the-shelf parts, including software development kits, this leads to a situation where a vulnerability in a single product can cause security issues in multiple types of CubeSats that were built by different vendors [Fal18]. Hence, such an incident could affect a whole satellite constellation or even multiple constellations. A specific issue is that the security of satellite firmware is not as well researched as other areas.

The situation described above shows that the security of satellite systems is an interesting research area with many aspects. Some of these aspects are areas with little previous research. One example of such an area is the security of CubeSat firmware. Because of that, we want to use this thesis to investigate the security of CubeSat firmware. More specifically, we want to do a fuzzing-based security assessment of the firmware of the OPS-SAT, a CubeSat that currently is in space.

The OPS-SAT in particular is an interesting research subject because we already know that it suffers at least from one security vulnerability. Hence, we want to use fuzzing to evaluate if there are other vulnerabilities that are currently not known.

1.2 Contribution

This thesis will provide the following contributions:

- Implementation of an AVR32 emulator
- Fuzzing-based security assessment of the OPS-SAT firmware
- Collection of hurdles and issues, that come up when fuzzing satellite firmware

The primary contribution of this thesis will be a fuzzing-based security assessment of OPS-SAT's firmware. We will use the well-known AFL++ fuzzing tool during the assessment. The assessment will focus on the security of functions that are involved in the communication with ground stations. However, before we start with the assessment, we need to overcome an important issue.

The Central Processing Unit (CPU) that is used in the OPS-SAT is based on the AVR32 architecture. This architecture is intended to be used in embedded devices, like the OPS-SAT. The particular CPU in the OPS-SAT is not powerful enough to do meaningful fuzzing. Hence, we will use *rehosting* to execute the firmware on a more powerful desktop computer CPU. This way, we can improve the performance of the fuzzing process. However, no suitable AVR32 emulator is available to the public at this time. Therefore, we need to implement an AVR32 emulator. Hence, the second contribution of this thesis will be a basic AVR32 emulator that later can be used by other researchers that plan to analyze AVR32-based software. To implement the emulator, we will extend the QEMU emulation software [QEMb].

As a third contribution, we will note issues and hurdles that we come across during the implementation of the emulator and the fuzzing. For example, we expect difficulties because the firmware needs to interact with peripheral hardware. As we want to reduce the complexity of the emulation, we will not emulate the full functionality of peripheral devices. Therefore, we need to find workarounds for such situations. These issues and their respective solutions likely apply to the firmware of other satellites and hence might be useful if similar projects are done in the future.

1.3 Organization of this Thesis

This thesis has three main parts that are further divided into 9 chapters. The first part explains the theoretical background of this work. Part two can be considered as the practical phase, where we implement the AVR32 emulator and started the fuzzing process. The third and final part consists of the evaluation and discusses the results of the practical phase. In this section, we will briefly explain the contents of each chapter.

As already mentioned, the thesis is divided into 9 chapters. In this first chapter, we talked about our motivation for the thesis. We also named the three primary contributions that will be provided by this thesis. In chapter two, we will introduce the relevant background topics that are necessary to understand certain aspects of this thesis. This means that we will briefly describe the OPS-SAT, the FreeRTOS operating system, the AVR32 architecture, and the QEMU project. We will also introduce the relevant aspects of fuzzing. Following that, we will explain our approach to achieve the research goals in chapter three. More specifically, chapter three explains our approach to the implementation of the AVR32-emulator and the fuzzing process.

Chapter 4 marks the start of the practical phase of this thesis. In particular, chapter 4 will explain how the QMEU project can be extended to add a new architecture to the project. Further, we will show how we implemented the architecture-specific parts that are necessary to emulate AVR32-based firmware. To this end, we will describe in deep how the translation of CPU instructions is done and how hardware devices can be emulated. Thereafter, in chapter 5, we will show how we searched for functions that can be targeted by fuzzing and how we selected one of these functions for the assessment. Additionally, we will describe the implementation of a connection between QEMU and the fuzzing tool. Another aspect that we cover is the creation of useful *input seeds*, which are needed by the fuzzing tool.

In the third part of the thesis, we will analyze the results of the fuzzing phase. This part starts with the evaluation in chapter 6. The evaluation includes an overview of the coverage that was achieved. Furthermore, we will describe how we improved the initial coverage for some parts of the firmware. Next, we evaluate the performance of our fuzzing implementation. We will also address an experiment that was done to improve the performance if only specific parts of the firmware should be targeted by fuzzing.

Following that, we describe a vulnerability that was found during the fuzzing phase and how we built an exploit for it. We will also demonstrate how we used our AVR32-emulator to verify a second vulnerability that was known before to this thesis. Additionally, we illustrate how we developed an exploit for this vulnerability and how we tested it with the emulator.

In chapter 7, we will discuss the results of our work. First, we will start with the discussion of the security of the OPS-SAT. After that, we will list different hurdles and issues that we came across during our work. Chapter 7 also includes a short list of *lessons learned*. This list contains things that we learned during this thesis and would like to have known before starting this work as that would have resulted in a much more efficient approach. Chapter 8 lists other publications and explains how they are related to this thesis. In chapter 9, we find a conclusion. The conclusion starts with a summary of this thesis. In the end, we will name future research questions that came up during this work.

2 Background

The background chapter provides a brief overview about the topics that this thesis is based on. We first describe the OPS-SAT and introduce important aspects of the FreeRTOS operating system. Thereafter, we explain the AVR32 CPU architecture that is used by the on-board computer of OPS-SAT. Next, we cover the QEMU emulator that we will use to emulate the firmware. In the last section, we will explain fuzzing, our primary tool for the security assessment.

2.1 The OPS-SAT

The OPS-SAT is an experimental space mission by the European Space Agency (ESA). The satellite was launched into space in 2019 and is currently flying 515 kilometers above the earth [Eura]. The goal of the OPS-SAT mission is to provide an experimentation platform that can be used by experimenters to test new concepts for satellite-based applications in a real-world setting [Eva16].

Regular satellite missions often take years to prepare and are worth millions of euros. To prevent potential safety issues that could result in the loss of a satellite, software that controls ESA satellites is strictly tested before being send to an in-flight-system. While such safety tests are critical for the integrity of a satellite, they are a hurdle when it comes to experiments and new approaches that need to be tested in real-world settings. Therefore, software experiments are difficult to prepare and often will not be carried out because of safety concerns. Additionally, satellites often lack a powerful processor. Because of that, many processing operations can not be done effectively on satellites [Eva16]. The OPS-SAT mission has the objective to provide a solution for these issues. The OPS-SAT provides a new kind of processing hardware that is said to be ten times more powerful than any other CPU used in ESA spacecrafts [Eura].

The OPS-SAT consists of two important, divided parts. The first part is the CubeSat bus, the other part is the Satellite Experimental Processing Platform (SEPP).

The CubeSat bus consists of a NanoMind A3200 board, a power control system, and a UHF communication system. The CubeSat bus also contains other utility systems that are needed to operate a satellite, like a GPS receiver. The CubeSat bus is responsible to monitor the satellite's state. It sends status information to the ground station and it can control the execution of experiments on the SEPP.

The firmware that is used on the NanoMind board will be the focus of this thesis.

The NanoMind A3200 is a System-On-Chip (SOC) that is designed for applications in space [Gom21]. The core of the SOC is a AT32UC3C microcontroller that is based on the AVR32 architecture. The A3200 also provides multiple subsystems, like an I2C interface and a Controller Area Network (CAN) bus interface. The A3200 offers a 32 MB SDRAM and a 120 MB Flash memory as well as a timing co-processor.

The second part, the payload, of the OPS-SAT is the SEPP. The SEPP is the main part of the OPS-SAT and contains the processing platform that is used to execute the experiments. The processing platform is based on an ARM A-9 dual-core CPU. It also provides various sub-systems and devices that can be used by experimenters:

- Camera
- Optical receiver for laser communication
- Software defined radio
- Second GPS receiver
- S-Band Transponder
- Fine Attitude Determination Control System

The OPS-SAT mission allows institutions and companies to apply for an experimentation slot [Eurb]. At this time, over 100 institutions applied to perform their experiments on the OPS-SAT. During an experiment, any system of the SEPP can be used by the experimenter software and experimenters are allowed to communicate with their software in real-time. This way, they can control the different sub-systems of the OPS-SAT [Eva16].

The CubeSat bus, the SEPP, and the various sub-systems are connected by different interfaces. A simplified diagram of structure of OPS-SAT can be seen in Figure 2.1. On important fact is that the SEPP and the onboard computer are connected by a CAN line and an I2C line that allow the two systems to exchange data.

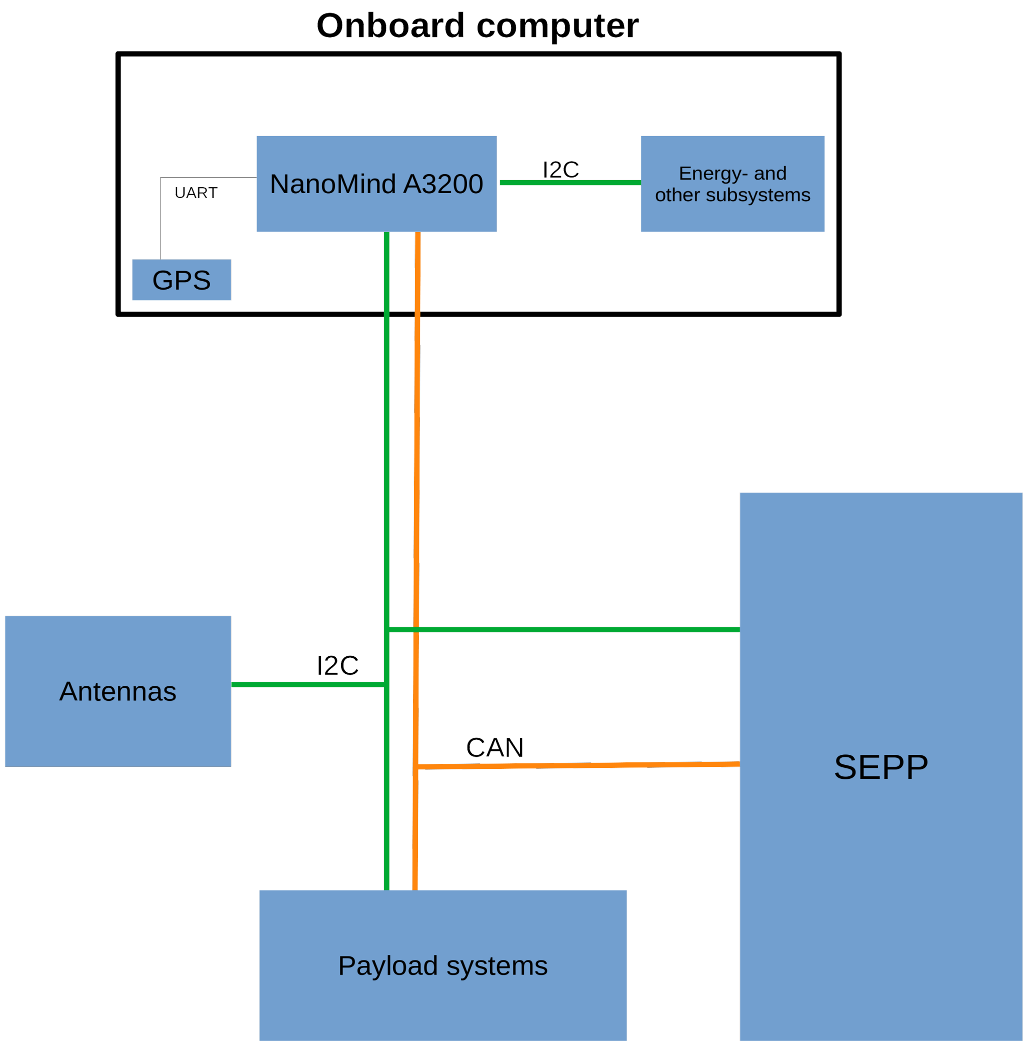


Figure 2.1: Simplified system structure of the OPS-SAT.

2.1.1 Additional motivation - Vulnerabilities

The OPS-SAT suffers from at least one security vulnerability. A function that works on input data that is loaded from an external connection suffers from a buffer overflow. Such vulnerabilities can be severe, because they may allow overwriting of values on the stack. If a code pointer is stored on the stack, it potentially can be overwritten and an attacker can gain control of the system. We will show that this is possible for this specific vulnerability by using the emulator that we will implement. However, identifying such vulnerabilities by static analysis can be a onerous task, as every function that works on buffers needs to be evaluated. This was an additional motivation to use fuzzing because fuzzing is a tool that is capable to detect such vulnerabilities more quickly.

2.2 FreeRTOS

FreeRTOS is a Real Time Operating System (RTOS) that supports the AVR32 architecture [Freb]. It is intended to be used on microcontrollers and is used by the OPS-SAT.

FreeRTOS uses a scheduling-based approach to perform multitasking on a single-thread CPU [Frec]. A firmware that is based on FreeRTOS consists of multiple independent *tasks* that are managed by a scheduler [Free]. The scheduler ensures that every task receives CPU time, depending the priority of the task.

A task has a *context* that consists of the CPU register values and the stack of the task. During it's execution, the task operates on the stack and the CPU registers, like a regular application would do. But, if a task used up its CPU time or performs an operation that is *blocking*, for example, reading input from a hardware device, it gets *suspended* until the blocking time is over and no task with a higher *priority* is being executed. This context switch is done by storing the CPU state on the stack and then loading the CPU state of the next task into the corresponding registers [Frea].

The context switches can be triggered by different events. As mentioned, a context switch occurs if a task goes into a blocking or waiting state. However, most context switches are initiated by an external event, mostly by a timer interrupt. If a timer interrupt occurs, the current task is suspended and a control task becomes active. Among other things, this task decides which task is the next to become active, depending on the state and priority of the tasks [Fred]. This way, different operations can be executed on a single-thread CPU, like the one in the OPS-SAT.

2.2.1 Security Mechanisms

FreeRTOS provides two different techniques to detect a stack overflow. The first method evaluates if the stack pointer is outside the allowed range of a task, after the task is suspended. After a task is suspended, the stack likely is at its lowest point and therefore a stack overflow could be detected [Fref].

The second method is a padding of known bytes that is written into the stack, after a task is created. If a task is suspended, FreeRTOS checks if the last 16 bytes in the stack are equal to the known bytes.

Both techniques are optional and only checked when the *vApplicationStackOverflowHook* function is called.

2.3 The AVR32 Architecture

AVR32 is a 32-bit load/store processor architecture that was developed by Atmel [Atmb]. It is intended to be used by microcontrollers and embedded systems.

AVR32 stores data in big-endian, hence the most significant bit is stored at the lowest address in memory. AVR32 can load and store 8 bit long *bytes*, 16 bit long *half-words*, 32 bit long *words* and 64 bit long *double-words*. AVR32 supports signed and unsigned bytes and half-words.

2.3.1 Registers

AVR32 provides 13 general-purpose registers. Additionally, there are registers for the stack pointer, the link register and the program counter. Besides that, AVR32 has a 32 bit long status register that stores information about the current CPU state. For example, the status register contains flags that are set if the result of an arithmetic operation is negative or zero. The status register also contains three *mode bits* that indicate the current operation mode of the CPU. For example, the CPU usually operates in the *application* mode. However, if an interrupt occurs, the mode changes and the mode bits are modified, according to the priority of the interrupt.

2.3.2 Instructions

AVR32 provides over 150 instructions that can be grouped into:

- Arithmetic operations

- Logic operations
- Bit operations
- Arithmetic and logical shift operations
- Instruction flow operations
- Data transfer and load and store operations.
- System control operations

Additionally, AVR32 supports operations on co-processors and Java code execution. Because the AR32UC3C processor that is used in the OPSSAT does not support java, the corresponding instructions are not further addressed in this work.

Instruction Structure

AVR32 instructions are either 16 or 32 bit long. A 32-bit instruction always starts with a bit sequence of *111*. This fact was handy when QEMU needs to decode instructions, as we explain in section 4.2. In general, an instruction consists of several fixed bits that indicate what instruction it is. Those bits are called *opcode* and they are used by the processor to identify what operation needs to be performed.

Besides the opcode, an instruction can have one or more operands. An operand can be an immediate value or the number of a register. AVR32 has the following operands:

- Register numbers and register lists
- Immediate values
- Conditions and bit flags
- Displacements
- Shit amounts
- Bit positions and widths of bit fields

In some cases, an operator is split into multiple parts that are distributed to different bit fields within the instruction, for example 21 bit immediate values. In total AVR32 has 58 different instruction formats that are described in the AVR32 architecture document [Atmb].

It should be noted that many instructions exist in multiple formats. For example, the *subtract* instruction can be used with two registers, performing the operation $rd \leftarrow rs - rd$, or with one register and an immediate value, performing the operation $rd \leftarrow rd - 0x10$, where rd and rs are register and *0x10* is the immediate values

16. Because of this, an AVR32 compiler can use the compact 16-bit format of instruction, if the operators are suitable. This way, the size of the compiled binary can be smaller. If the operators are not suitable, for example, because an immediate value needs 21 bits, it can still use the less compact 32-bit format. Another advantage of this concept is that some instructions can perform multiple operations so that a compiler can spare a whole instruction. For example, if a program segment needs to perform the operation $rd \leftarrow rd + (rs << 2)$, this can be assembled as one *shift* instruction and one *add* instruction. But AVR32 also provides the *add* instruction in a format that includes an operand shift. Therefore, the resulting binary is more compact.

2.4 The QEMU Emulator

The Quick Emulator (QEMU) is an open source software project. QEMU allows rehosting of software [QEMb]. Rehosting means that software that was compiled for different architecture than the host processor architecture can be executed. Rehosting will be used to emulate the execution of the OPS-SAT firmware. This is necessary because the processor of the OPS-SAT is not fast enough to support meaningful fuzzing. Because the AVR32 architecture is supposed to be used in embedded systems, AVR32-based processors in general are likely to be significantly slower than the CPU of a regular desktop computer. By rehosting the target firmware, these limitations can be circumvented [MSK+18a].

QEMU will be used as an emulator, because it is a well-known project that provides lots of features, including full-system-emulation. Because the OPS-SAT firmware runs a real-time operating system, the emulator needs to emulate a full virtual CPU with memory access and related operations. Currently, there is no AVR32 support in QEMU. The only publicly available AVR32 implementation is a fork that was not updated since 2012 [tur]. This fork did not implement actual CPU instructions that could be emulated. Because we could not find any other system emulator with AVR32 support, we will implement basic AVR32 support into QEMU.

2.4.1 General Concepts

QEMU uses the Tiny Code Generator (TCG) to translate instructions from one architecture into another. This is done by parsing an instruction and then generating host architecture code that performs the operation of the instruction. The TCG is executed in a loop and reads CPU instructions from the emulated memory. Every instruction is translated into one or more intermediate code instructions until a full basic block is translated. A basic block is a segment of instructions that ends after a branch instruction [QEMe]. The intermediate code is then translated into host architecture instructions and executed. QEMU next uses the Program Counter Register (PC) to find out at which address the execution should continue. The TCG may perform code optimizations. For example, an operation that calculates a value that is just overwritten in the next operation will be ignored.

QEMU also provides memory emulation. A virtual CPU can access an address in the virtual memory that QEMU then maps to a physical memory location in the host memory. Besides that, QEMU also supports the emulation of peripheral hardware, like CAN buses or UART lines that can be accessed by a virtual memory address.

2.5 Fuzzing

Fuzzing is a technique that is used to find vulnerabilities in applications [SWS+16]. When fuzzing an application, input is automatically generated and send to the application. The application either executes in an intended way and ends with a regular exit code or the input provoked unexpected behaviour and the application ends with a crash, for example, a segmentation fault. The fuzzer keeps track of the execution flow of the application and changes the input before the application is executed again. The fuzzer tries to generate input that results in as much covered code as possible. This approach is called coverage-based fuzzing. *Coverage* refers to the code segments that were executed. The goal of fuzzing is to find an input that results in a crash of the target application. A crash indicates that a potential vulnerability is present in the code segment that caused the crash [Oeh05]. Fuzzing is usually done in a *fuzzing loop*. The fuzzer will execute the target application and send the generated input to it. The fuzzer then keeps track of the coverage and crashes, if any occur. After the application ended its execution, the fuzzer will use the coverage and crash information to generate a new input. Next, the application is started again and the new input is send to it. A fuzzing tool that uses coverage-based fuzzing is the American Fuzzy Lop (AFL) [pro]. This tool will be used in this thesis.

Fuzzing can be divided into 3 categories: white box fuzzing, gray box fuzzing, and black box fuzzing [ZDY+19]. Black box fuzzing only needs the executable binary of an application. The fuzzing tool is not using any output from the application to improve the next input. On the other hand, white box fuzzing uses advanced binary analysis techniques like symbolic execution. Gray box fuzzing is a technique that lies between the to other categories. When doing gray box fuzzing, the fuzzing tool is using a small amount of information to improve the inputs. In our case, the fuzzing tool will use the coverage to generate new inputs. Gray box fuzzing is considered to be effective in real-world applications [ZDY+19]. Because we have no access to the source code of the OPS-SAT firmware but can gain coverage through the QEMU emulation, we will use gray box fuzzing for the security assessment.

3 Approach

In this chapter, we will shortly explain the approaches that we will take to achieve the three research goals: the implementation of the AVR32 emulator, the security assessment and the identification of issues. The first section explains how we aim to implement the AVR32 emulator with QEMU. The second section will highlight how we plan to do the fuzzing.

3.1 Emulator Implementation

The QEMU project provides different descriptions of the inner workings of the emulator. There are also some articles on implementing new architectures into QEMU. We will use these resources to implement the functions that are needed by QEMU to run a virtual CPU. We will then use the AVR32 instruction set manual to implement all instructions that are part of the OPS-SAT firmware. This will be done in an iterative approach. We will implement the first few instructions of the firmware and then execute the emulator. We expect the emulation to stop after a short time, as the emulator will not be able to execute the first unknown instruction. Then we can look up that instruction in the manual and implement it. This will be done until the firmware is executed without finding new instructions and therefore is an ongoing task.

The iterative approach is taken because this way the complexity can be reduced, as only relevant instructions will be be implemented. We also will save time because not every instruction is found in the firmware and therefore not all instructions need to be implemented. For example, some instructions execute Java code. However, the CPU in the NanoMind board does not support Java, hence the related instructions will be left out.

Because we expect a noticeable amount of complications that result from missing hardware input, we will look for code segments that try to read data from peripheral devices. If the emulation does not continue properly without the external input, we will implement a workaround. For example, the instructions in question could be skipped. If this solution is not enough, we will implement a basic hardware device as a last resort and try to determine the expected input values by reverse engineering the relevant code segments. Because the QEMU projects does not provide a suitable manual on the implementation of virtual devices, we will analyze the code of other hardware devices in the QEMU repository and use it as a reference.

3.2 Fuzzing Approach

We decided to use AFL++ as a fuzzing tool for this thesis. The tool was chosen because it is under active development, it provides a plugin API, and a QEMU mode for fuzzing of emulated binaries [FMEH20]. However, currently AFL++ does not support the QEMU version that our emulator is based on. Therefore, we will analyze how AFL++ supported QEMU in previous versions and implement the necessary connection functions into our emulator.

The fuzzing will be done in a loop. This *fuzzing loop* consists of the following steps:

1. The fuzzing tool generates input and sends it to a command handler.
2. QEMU will emulate the input handling.
3. QEMU will report coverage information to AFL.
4. QEMU will report an exit code to AFL after the handler function ends or the firmware crashes.

After the last step, the fuzzing loop starts again with new input data.

For the first step, we will implement a connection between AFL and QEMU that allows AFL to send input data into the emulated memory if the emulation reaches a specific code segment. We plan to write the input data into a buffer that is used to store input data before the data is used by an input handler.

Next, QEMU will emulate the input handler function. To do so, no further implementation is needed, as the emulator will be capable to emulate all necessary instructions beforehand. However, we need to identify a suitable command handler function for the fuzzing input. To find an adequate target for the fuzzing, we will reverse engineer the OPS-SAT firmware. We will try to find a function that is involved in outside communication. Because we have an Executable and Linkable Format (ELF)-file of the firmware, we can search for function names that imply that a function is used for outside communication. Functions that are of particular interest are those that are used to start the execution of operations on the satellite and are executed in a loop. We likely need to further reverse engineer parts of the firmware that are responsible for the input handling to determine how the inputs are handled and if the fuzzing input seeds need to be optimized. We will do this by statically analyzing the firmware image. After the emulator is implemented we also will be able to dynamically analyze the firmware.

QEMU will report coverage information to AFL during the emulation of the handler function. To provide this information, we will implement a patch to QEMU that calls a reporting function at the beginning of every code block that is executed. To be able to evaluate the effectiveness of the fuzzing, we will also collect the coverage information in a log file.

The last step in the fuzzing loop is the reporting of an exit code. We will implement a patch to QEMU that reports the exit code *zero* to AFL if the end of the input handler function is executed. This exit code indicates that no error occurred. We will also identify error handling functions in the firmware, again by statically analyzing the firmware image. If one of these functions is called during the input handling, this may indicate that the input triggered a crash of the firmware. We will implement a patch to QEMU that reports an error exit code to AFL if one of the error handling functions is executed and then restarts the emulation.

4 Implementing an AVR32 Emulator

This chapter covers the first part of our practical work. In the first three sections, we describe how we implemented the AVR32 architecture in QEMU and how we build a virtual NanoMind board. Then we discuss different hardware components that were emulated in QEMU. In the last section, we explain how we build a testing framework that was used to validate the AVR32 emulation.

4.1 QEMU code translation

We explained the general concept of QEMU in section 2.4. Now, we will describe how binary data of one CPU architecture is translated into another and then is executed by QEMU.

The architecture specific function that translates binary data into QEMU's intermediate code, is set in the `.translate_insn` parameter of the `TranslatorOps` in the *cpu.c* file. For our implementation, this is the `avr32_translate_insn` function.

The `avr32_translate_insn` function first sets the emulated program counter register to the value of the current program address. After that, the `decode_insn_load` function is called. This function is included from an external file that is generated by QEMU's *decodetree.py* script during QEMU's compilation. The *decodetree.py* script reads the instruction patterns for the AVR32 architecture, which are described in section 4.2, and generates corresponding C code that allows QEMU to interpret binary data as target architecture instructions.

4.2 The Decodetree

AVR32 instructions are always 2 or 4 bytes long. A 4 bytes long instruction is indicated by bit mask of `111` at the start of the instruction. After 2 or 4 bytes of binary data are loaded, QEMU needs to decode the data, to identify which instruction needs to be executed. QEMU uses a *decodetree* that consists of instruction *patterns* to identify how a specific sequence of bits should be interpreted [QEMa]. The decode tree is created at QEMU's build time. The creation is done in the *decodetree.py* script that is part of QEMU. The following parts of this section will discuss the decode tree patterns and how they are defined for the AVR32 instruction set.

4.2.1 Patterns

The QEMU *decodetree* contains patterns for every instruction QEMU needs to emulate. In the AVR32 implementation, the decode patterns were specified in the *insn.decode* file.

A decodetree pattern is a sequence of bit values and wildcards that are characteristic for an instruction. Depending on the instruction and the architecture, there can be patterns of different lengths. The AVR32 architecture has instructions with 16 bits length and instructions with 32 bits length. Therefore, every pattern must be 16 or 32 bits long. A pattern starts with an identifier that is followed by a sequence of bit values, called *opcode*, and wildcards. A minus indicates an irrelevant value and the bit at such a position will be ignored. A dot indicates a dynamic value, for example, an immediate value that was set when a program was compiled. These values will be evaluated and used in so called *fields*. The identifier of the pattern is also the name of the function that QEMU calls to generate the intermediate code for the instruction.

As an example, the patterns of the `AND` instruction that performs a logical *and* operation and the pattern of the `ADD` instruction that performs an arithmetical *add* operation are shown below:

```
AND_rr        000 .... 00110 ....                 @op_rs_rd
ADD_rd_rs     000 .... 00000 ....                 @op_rs_rd
```

Listing 4.1: The patterns of the AND and ADD instruction.

Both patterns start with an identifier on the left side. In the middle, the opcode can be seen. It should be noted that both patterns do **not** start with a sequence of `111` bits, as the respective instructions are 16 bits long. A pattern can have none, one, or multiple *fields* that will be passed to the TCG. Each field can contain an immediate value, the number of a register or sub-opcodes. The latter is not used in the AVR32 architecture. For example, the `AND` and the `ADD` instruction each contain two fields, which hold the numbers of registers that are used for the operations. Because these values are set during the compilation of a program, they are represented by the dot wildcard.

The final part of a pattern definition is the identifier of a *format* that helps to avoid redundant definitions if a specific order of fields is used in multiple instructions. Because both instructions use two fields with the same length at the same position, they both use the `@op_rs_rd` format.

QEMU will perform pattern matching to find the corresponding decode pattern for a sequence of bytes in a binary file. After that, the instruction is passed to the format.

4.2.2 Formats

As mentioned earlier, instruction patterns can have *fields* that contain values that are set at compile time. In the AVR32 architecture, most instructions have multiple fields, for example, to specify the destination and the source registers of an operation or to specify which condition needs to be evaluated for a conditional operation. Many instructions have fields of the same length at the same position. Often these fields also have the same purpose, for example, to specify the destination register. QEMU provides *formats* to easily work with the values of such fields and to prevent redundant code. These formats are a handy tool to work with the 58 different instruction formats that are part of AVR32.

For example, the AND instruction has two fields: one for the source register and one for the destination register of the operation. The AND instruction also uses two fields at the same positions, to specify the source and destination registers of the operation. To use these fields, QEMU passes their bit sequence to the `@op_rs_rd` format that is also defined in the *insn.decode* file:

```
@op_rs_rd          ... rs:4 ..... rd:4                     &rs_rd
```

Listing 4.2: The definition of the op_rs_rd format.

The format starts with its identifier `@op_rs_rd`. It is followed by its characteristic field sequence. As the fixed values of the opcode are not relevant to the format, they are replaced by the dot wildcard. Deviating from a pattern, the fields are not replaced by wildcards but by a name and their length in bits. In the example above, there are two fields: the `rs` field, short for **r**egister **s**cource (rs), with 4 bits length, and the `rd` field, shot for **r**egister **d**estination (rd), with also 4 bits length.

A field definition can end with the name of an *argument set.* The argument set specifies the names of the fields in a format. These names are later used to access the values of the fields in the translation functions. If no set is given, the field names will be used as arguments. For the AVR32 implementation, we used argument sets for every format definition, as they may be useful in future updates of the AVR32 implementation.

```
  &rs_rd            rs rd
```

Listing 4.3: The `&rs_rd` argument set. The fields will be available as the variables `rs` and `rd` later.

The argument set starts with the an identifier and then has a name for every field in the format.

To implement the AVR32 instruction set for QEMU, we took the opcode format for every instruction that occurred in the OPSSAT firmware from the architecture document [Atmb] and noted it in the way described above.

4.3 Intermediate Representation

After QEMU interpreted binary data as an instruction, it calls the corresponding handler function with the prefix `trans_` to translate the operation of the instruction into an Intermediate Representation (IR), for example `trans_ADD_rd_rs`.

The handler functions are defined in the *translate.c* file. The handler functions translate the operation of their corresponding instruction into an IR [QEMe]. For example, the AVR32 architecture document defines the operation of the ADD instruction like this:

$$Rd \leftarrow Rd + Rs, \{d, s\} \in \{0, 1, \dots, 15\} \tag{4.1}$$

`Rd` is the register for the first argument and the result of the addition. `Rs` is the register for the second argument of the addition. `Rd` and `Rs` can represent the same registers, like *add r5, r5*. In this case, the CPU adds the value of register 5 to register 5 itself. The specific registers are set by the compiler, when the program is compiled into a binary file. Additionally, the ADD instruction also modifies the status register, depending on the result of the operation. For example, the *zero flag* (z-flag), which indicates if the result of an operation is zero, is set to 1, if the result of the addition is 0. If the result of the addition is **not** zero, the z-flag is set to 0.

When adding a new architecture to QEMU, the behavior of every instruction needs to be implemented with QEMU's *front-end* operators [QEMd]. Front-end operators are used to generate the code of the IR in the handler functions. Front-end operators do not modify the contents of the emulated registers or the virtual memory directly, but generate the *IR* code that is later used to perform the actual operations.

The ADD instruction can be translated with front-end operators to IR as follows:

```
static bool trans_ADD_rd_rs(DisasContext *ctx, arg_ADD_rd_rs *a){
  TCGv res = tcg_temp_new_i32();
  TCGv Rd = tcg_temp_new_i32();
  TCGv Rs = tcg_temp_new_i32();
  tcg_gen_mov_i32(Rd, cpu_r[a->rd]);
  tcg_gen_mov_i32(Rs, cpu_r[a->rs]);

  tcg_gen_add_i32(res, Rd, Rs);
```

```
 9    tcg_gen_add_i32(cpu_r[a->rd], cpu_r[a->rd], cpu_r[a->rs]);
10
11    // Setting the status register
12
13    if(a->rd == PC_REG){
14      ctx->base.is_jmp = DISAS_JUMP;
15    }
16
17    tcg_temp_free_i32(res);
18    tcg_temp_free_i32(Rd);
19    tcg_temp_free_i32(Rs);
20    //...
21
22    ctx->base.pc_next += 2;
23    return true;
24  }
```

Listing 4.4: The trans_ADD_rd_rs function in translate.c.

In lines 2 to 4 we initiate a 3 temporary variables, which are needed later. Temporary variables are used to perform operations that should not change the content of an emulated register. This is necessary, because many instructions perform multiple operations at once. For example, the ADD instruction changes the destination register and the status register. However, QEMU can not perform this operation simultaneously. Therefore, temporary variables are needed to keep certain values until the instruction is completed. In case of the ADD instruction, the original content of the destination register and the result of the addition are needed to set the status register.

In line 5 the value in the destination register is moved into the `Rd` temporary variable. In line 6 the same is done for the source register. The `cpu_r` variable is a global array that holds the values of all CPU registers. These values are also `TCGv` variables. A special aspect of the TCG variables is that their value can not be accessed by the translation function directly. As mentioned before, QEMU first translates a basic block to IR and then executes it. Therefore, the values in the TGCv variables are not known when the translation function is executed.

The variable `a` holds the content of fields, which can be accessed by the names set in *insn.decode*. In this case, `a` holds the specific number of the registers `rs` and `rd`. As these numbers are decoded form the loaded binary data, they are known during the execution of the translation function and directly accessible as a C integer.

In line 9, we call the TCGs `gen_add` function, with `res` as the result variable and `Rd` and `Rs` as arguments. In line 10 we do the same, but we use the real emulated registers as arguments. The temporary variables are necessary, because the original value in `cpu_r[r->rd]` is needed to correctly modify the status register.

If the destination register is the PC register, the CPU would jump to another location within the program. Therefore, we check if `a->rd` is the PC registers in line 13 and tell QEMU that the translation needs to continue at another translation block in line 15, if `rd` is the PC register.

In line 18 to 20 the temporary variables are deleted. This is important because QEMU exits with an error after the translation block was executed and temporary variables are not cleared. At last, the program counter is increased by 2, as the ADD instruction is 16 bit long. The function returns true, to tell the translation loop that the instructions was translated successfully.

QEMU calls a translation function for every instruction in a basic block. Before QEMU can finally execute the IR of the translated block, the IR is translated to instructions of the host architecture that QEMU is executed on. This is done by existing QEMU functions that are not specific for the guest architecture.

4.3.1 Branch instructions and conditions

Some instructions perform conditional operations. If the condition depends on a value from an instruction field, the condition can be simply evaluated by the translation function. For example, the ANDL operation that performs a logical *and* operation on the lower half of a register has a flag that determines if the upper half of the register should be set to zero. If the flag is set to `1`, the upper 16 bits of the register are cleared. Otherwise, the upper half of the register is not affected by the instruction. Because the bit in the flag is set during compile time, it is known when QEMU translates the instruction. Hence, we can utilize a regular if statement from C code, as the instructions handler will not change during runtime.

However, if the condition depends on the content of a register, a more complex approach is needed. Because values in registers are not known during the translation, they need to be evaluated with frontend operators. Because QEMU has no frontend operators that work like an if-statement, implementers need to use so called *TCGLabels* to perform jump operations based on register evaluations. First, the condition is evaluated. Then, a conditional branch is executed within the instruction, depending on the result of the evaluation. The labels then are used to jump past operations that should not be executed because of the evaluated condition. An example of such instructions is the conditional branch (BR) instruction:

```c
static bool trans_BR_rd(DisasContext *ctx, arg_BR_rd *a){
  int disp = sign_extend_8(a->disp) << 1;

  TCGLabel *no_branch = gen_new_label();
  TCGv reg = tcg_temp_new_i32();
  int val = checkCondition(a->rd, reg);

  tcg_gen_brcondi_i32(TCG_COND_NE, reg, val, no_branch);
  gen_goto_tb(ctx, 0, ctx->base.pc_next+disp);

11    gen_set_label(no_branch);
12
13    tcg_temp_free_i32(reg);
14    ctx->base.pc_next += 2;
15    ctx->base.is_jmp = DISAS_CHAIN;
16    return true;
17  }
```

Listing 4.5: The trans_BR_rd function in translate.c.

The BR instruction uses a displacement to calculate the destination address of the branch. This displacement is calculated in line 2. To do so, the displacement is first signed extended, as it is defined in the AVR32 instruction set manual. Then the extended displacement is logically shifted one bit to the left, again as stated in the instruction set manual. In line 4 a `TCGLable` is defined. The label will be used later, to skip parts of the IR. Line 5 is used to declare a temporary variable. In line 6 the `checkCondition` function is called. The first argument of the function is the condition that needs to be checked. The condition is given as a 4 bit value in a filed of the instruction. The second argument is the temporary variable. The `checkCondition` function then evaluates the passed condition based on the content of the status register. For example, a condition could be the *equal* condition.

The *equal* condition is true, if the zero flag in the status register is `1`. In this case, the temporary variable is filled with the value of the zero flag from the status register and the return value of the function is `1`, as the zero flag needs to be `1` for the condition to be true. The actual evaluation is done in line 8. The `tcg_gen_brcondi_i32` frontend operator is used to check if the value in the temporary variable is **not equal** to the return value of `checkCondition`. If the evaluation is true, the condition of the branch instruction is **not** true and **no** branch should be done. Therefore, `tcg_gen_brcondi_i32` will jump to the `no_branch` label that is given as the third argument. The label is set in line 11. If the result of the evaluation is **not** true, the condition of the branch instruction **is true** and a branch should be performed. In line 10, this branch is done by calling the `gen_goto_tb` function. The branch target is given as the third argument, with the current program counter and the displacement. In line 13 to 16 the instruction epilogue is given.

QEMU creates the intermediate representation sequentially, in the same order as the frontend operators that are used in the translation function. Because the branch that the BR instruction should perform if the condition is true is set in line 10, a jump to the label that is set in line 11 would cause the emulator to skip the branch operation in line 10. This way, QEMU can evaluate conditions that depend on register values.

The approach that was explained above has the disadvantage that it increases the complexity of the translation function, as it is more complex than a simple if-statement. In the example above, only one condition needs to be evaluated and a single label is enough to reproduce the operation of the branch instruction. However, there are AVR32 instructions that perform multiple evaluations. For example, the POPM instruction, which loads multiple registers from the stack, contains multiple nested if-statements. The implementation of the corresponding translation function was a complex task and the risk for implementation errors was higher than for other instructions. We explain how such instructions can be tested for validity in section 4.6.

4.4 Virtual Hardware

Besides the virtual AVR32 CPU, which executes the instructions, QEMU also needs a virtual hardware machine that is used to operate the virtual CPU. When starting QEMU, the user needs to specify which virtual machine should be used to emulate an application. Hence, we implemented a virtual NanoMind A3200 board in the `hw/avr32` folder. This folder contains all files that define virtual hardware devices that make use of the AVR32 architecture. The virtual NanoMind creates another virtual device, to emulate the AT32UC3C microcontroller, during its initialization. The virtual AT32UC3C is the key component of the hardware emulation. It holds the definitions of the memory areas that the firmware uses. These are the Static random-access memory (SRAM), the Flash memory, and the Synchronous Dynamic Random Access Memory (SDRAM). The memory areas have the same address spaces as a physical AT32UC3C would have.

The AT32UC3C class also controls the virtual *sysbus*. The virtual sysbus is used to connect different virtual hardware devices. One example is the virtual Universal Asynchronous Receiver Transmitter (UART) device that we implemented. The firmware uses a UART interface to print log information. By reverse engineering the relevant parts of the firmware, we found the memory address mapping for the text output. The virtual UART device was then used to print any text output to a log file. This information was useful to further understand the firmware's behavior, especially because some error messages are printed to the UART device.

Another important hardware component is the Interrupt Controller, which is explained in section 4.5. We also implemented different other hardware components, like a Serial Peripheral Interface (SPI) interface or a CAN bus. Most of these devices were only implemented in a rudimentary way to allow the firmware to pass certain checks or to observe the communication to the respective device. For example, we implemented a virtual FM33256b Ferroelectric Random Access Memory (FRAM) device. At one point, the firmware tries to read a value from this device to determine if the antennas of the satellite were automatically deployed. Without this information, the firmware did not continue its execution and therefore we decided to implement the virtual FM33256b. To do so, we added a memory mapping in the virtual AT32UC3C device. The firmware tries to read or write values to certain memory addresses, when it wants to communicate with peripheral hardware. QEMU will redirect the communication to the virtual device that was registered for this address space. With the virtual device, we can observe the input that the firmware sends and respond with output of our choice. In case of the virtual FM33256b, we respond with the value `1`, if the firmware tries to read data from the address that holds the antenna status value.

In case of most of the other virtual devices, like the SPI and CAN interface, we implemented status return functions and otherwise only observed the input form the firmware. Because we did not have access to images of the memory devices, we were not able to respond with meaningful data and decided to not implement more emulation functionality. However, this can be used as a base for further improvements of the emulator in the future.

4.5 Timer Interrupts

During initialization of the OPS-SAT firmware, multiple tasks for the FreeRTOS operating system are created. Among others, the firmware creates the `prvIdleTask` at the end of the main function. The Idle Task has the lowest priority of all tasks. It checks if any other task is ready via supervisor calls and calls the sleep instruction, if no task is ready. This results in a halt of the CPU. The CPU continues execution if an interrupt occurs.

The other tasks that are created during the firmware initialization all contain operations that need to wait for a defined time. For example, the `init_task` calls the `init_adcs()` function that calls other functions until it reaches a call to `vTaskDelay`. The `vTaskDelay`function sets a task to inactive until a certain amount of clock cycles has passed. Because no external clock was emulated, tasks like this were never reactivated. After the `prvIdleTask` called the sleep instruction, the emulation stopped and the firmware was not further executed by QEMU. Therefore we needed to implement a clock device that periodically causes interrupts to wake up the CPU and increase the tick counter. This way, the emulator is able to perform timed operations.

As mentioned earlier, virtual hardware devices need a reference to the virtual sysbus device. This is also true for the timer, as it needs to interact with the virtual CPU. The timer also has a virtual *interrupt request line*. The interrupt request line is connected to the virtual CPU and is used to trigger an interrupt that stops the current operation of the CPU and starts the execution of a special code segment that handles the interrupt.

The main function of the timer device is a separate thread that is created during the timer initialization when QEMU is started. The timer thread waits until it is activated by a *pthread* signal. This is done because the firmware first needs to set up the interrupt handler functions during its initialization phase. After that, the timer thread executes a while loop every 100 microseconds. Each time the loop is executed, the timer creates an interrupt on the virtual interrupt request line. The speed of the emulation depends on the sleep-value of the timer thread. We noticed that the emulation is running faster for lower values, but slows down for very low values. A value of 100 microseconds was determined to be fast-working. It was used during the fuzzing process.

To start the timer, we inject a patch into the MCALL instruction that calls a QEMU helper function at `0xd00c43f6`. At this address, the interrupt handlers are just prepared and the firmware can handle interrupts. The helper function sends a *pthread* signal to the timer thread that then becomes active.

```
1   d00c43ea 49 7c        LDDPC     R12=>vTick,->vTick
2   d00c43ec f0 1f 00 17  MCALL     PC[->INTC_register_interrupt]
3   ...
4   d00c43f6 f0 1f 00 16  MCALL     PC[->tc_init_waveform]
```

Listing 4.6: The code segment before address `0xd00c43f6`.

Listing 4.6 shows the code segment before address `0xd00c43f6`. In line 1, the firmware loads the address of the `vTick` function into register 12. In line 2 the `INTC_register_interrupt` function is called. This function is used to register a function as an interrupt handler in the interrupt handler table. After the function call returns, `vTick` can be used for interrupt handling. In line 4 MCALL is used at address `0xd00c43f6`. When this address is executed, the QEMU helper starts the timer. We decided to apply the patch to MCALL at this address because the MCALL instruction already contained patches that were needed to emulate the firmware. This way no other translation function needed to be modified.

4.5.1 Consuming interrupts

As mentioned in the previous subsection, the timer is connected to the virtual CPU by an interrupt request line. If an interrupt is send through the interrupt request line, the virtual interrupt controller that we implemented is used to further handle the interrupt. During the initialization of the AT32UC3C device, a function was set as an interrupt handler. If an interrupt is send to the AT32UC3 device, the interrupt handler function is responsible to set the CPU state accordingly. Specifically, the interrupt controller needs to set the interrupt level and the *autovector* [Atma].

The interrupt handler function first checks if an interrupt mask is set in the status register. If this is the case, the interrupt handling is disabled by the CPU and any interrupt is to be ignored. For example, the interrupt masks are set if the CPU already is handling an interrupt of the same level.

Next, the function determines which interrupt request line caused the interrupt. The timer device is connected via line 10. The interrupt controller has multiple status registers that hold information about the interrupt level and the interrupt cause, the *interrupt cause registers*. The `_get_interrupt_handler` function of the firmware reads out these registers and uses their values to calculate the address of the appropriate interrupt handler. The addresses of the interrupt handlers are stored in tables that are filled during the initialization of the firmware. Because of this, we performed the calculations of `_get_interrupt_handler` in reverse, to get a set of values that result in a call to the `vTick` function at address `0xd00c4478`. The real values for the interrupt cause registers may be different in the real OPS-SAT.

The last step is a call to QEMUs `cpu_interrupt` function that invokes a call to the `avr32_cpu_exec_interrupt` function that we implemented. This function again checks the interrupt masks and initiates a call to the `avr32_cpu_do_interrupt` function.

```
//Store regsiters r8 to r12 on the stack.
*(--SP SYS ) = LR;
*(--SP SYS ) = PC of first noncompleted instruction;
*(--SP SYS ) = SR;
SR[M2:M0] = 010;
PC = EVBA + INTERRUPT_VECTOR_OFFSET;
```

Listing 4.7: Extract of the INT0 exception pseudocode according to AVR32 instruction set manual page 73.

The `avr32_cpu_do_interrupt` function saves the registers r8 to r12, Link Register (LR), and PC to the stack and sets the interrupt masks, as defined in the AVR32UC manual. The function also sets the PC register to the address of the firmware's exception handler. The exception handler will call the `_get_interrupt_handler` function and then continue at the `vTick` function.

Inside the vTick function, the firmwares TickCounter will be increased by one. After that, the firmware continues with the highest priority task that is in the *ready* state or, if no other task is ready, with the previous task. To do so, the firmware uses the RETE instruction that returns from an event handler.

4.5.2 The RETE instruction

The implementation of the RETE instruction was an onerous task because the firmware fell to undefined behavior, if RETE was implemented as shown in the AVR32 manual. When an interrupt occurs, the CPU stores the status register on the stack, as explained above and shown in Listing 4.7. During the normal execution, the mode bits in the status register have the sequence `000`. This indicates that the CPU is in the *application* mode. If an interrupt with the priority `0` occurs, the mode sequence in the status register is changed to `010`, **after** the status register is saved to the stack. The first operation of the RETE instruction is to restore the status register from the stack, as shown in Listing 4.8. After that, the instruction checks if the current mode is `010`, `011`, `100`, or `101`. If this is the case, the registers that were saved to the stack are to be restored. Otherwise, the registers keep their current values and the stack pointer is not changed.

```
SR = *(SP SYS ++)
PC = *(SP SYS ++)
If ( SR[M2:M0] == {010, 011, 100, 101}){
  LR = *(SP SYS ++)
  //Restore R12 to R8
}
```

Listing 4.8: Extract of the RETE operation pseudo code according to AVR32 instruction set manual page 306.

If the instruction is implemented as shown in Listing 4.8, the emulation crashed. This is, because the mode check is performed against the status register value that was stored on the stack **before** the exception mode was set. If the interrupt occurs in the *application* mode, the check will never pass, because the RETE instruction already restored the `000` mode sequence. Hence, the firmware continues to add values to the stack during each interrupt that are never removed from the stack. After some time, the increasing stack size in the tasks context will provoke that other values are overwritten. In some cases, the FreeRTOS stack overflow protection noticed this situation and triggered a CPU rest.

Even after a significant time of studying the AVR32 instruction set manual, no reason for the above contradiction could be identified. We conclude that the description of the operation of RETE is liable to be misunderstood and that the mode check intended be performed against the status register values at the beginning of the RETE instruction. After we changed the implementation of the instruction so that the status register value is first checked and then restored, the firmware was executed as expected. The stack was properly cleared after every interrupt and no values were overwritten without intend.

4.6 Testing Framework

During the implementation of the AVR32 architecture in QEMU, implementation errors occurred frequently. One reason for that is that there are no automated checks for application logic errors. Assume we want to perform an operation $rd \leftarrow rx + ry$ and implemented the following code:

```
1   //Correct
2   tcg_gen_add_i32(cpu_r[a->rd], cpu_r[a->rx], cpu_r[a->ry]);
3
4   //Wrong
5   tcg_gen_add_i32(cpu_r[a->rd], cpu_r[a->rx], cpu_r[a->rx]);
```

Listing 4.9: A little logic error that is not automaticly detected.

In the example above, the operation is done by using the `add_i32` front-end operator that performs an addition on two 32-bit `TCGv` variables. The operator is called correctly in line 2. The second argument of the add function is the `ry` register. On the other hand, in line 5 the `rx` register was passed as first **and** second argument to the operator. As the code has a correct syntax and the `rx` register is a valid input, neither the compiler nor QEMU itself complain about this issue. A mistake like this can occur because a wrong key was pressed, `x` instead of `y`, or the code was copied from a similar function and not edited correctly.

As this error is purely in the application logic of the emulator, the error will stay unnoticed until the emulated program behaves in a way that is not expected. As this might be very deep into the emulation, the source of the error is not easy to determine. While errors like this are also possible when developing 'regular' applications, such errors are more likely to occur when implementing a QEMU extension. This is because the use of the intermediate representation adds another level of abstraction to the application. Also, the debugging process for QEMU is more difficult, as the emulated values are not directly accessible when QEMU is debugged. Because the debugging takes place when the translation function is executed, the values in the virtual registers are not known. Multiply days of try and error were spent to identify faulty parts of the implementation that were caused by minor errors like this.

To identify errors like the above, we developed a semi-automatic testing framework. The framework consists of various test files that verify AVR32 instructions. The test files also state the correct register contents of the virtual CPU after the test is done. The tests are then compiled and executed by a series of *python3* scripts.

4.6.1 Test generation

Each test case is defined and stored in an individual python file. The test cases consist of two parts: AVR32 assembler code, with the instructions that should be executed by the emulator and the expected register contents after the test cases finished their execution. Usually, the test cases need to set up one or more registers, before the instruction of interest can be used. Therefore, the test case uses multiple other instructions that may interfere with the status register. When implementing new tests, this needs to be considered, to ensure that the test result actually shows the result of the intended instruction and is not affected by the test preparation code.

The assembler code is stored in a string in the test file. A python *dictionary* is used to define the expected results of the test. For each relevant register, the expected value is given in the dictionary. Registers without defined result values are ignored for the test's evaluation. The expected results can also include the flags from the status register.

Below an example for a test of the AND instruction is given. We expect that the AND instruction performs the following operation: $r5 \leftarrow r5 \wedge r4$.

```
 1  TEST = """
 2  mov r4, 0x1020
 3  mov r5, 0x1020
 4  and r5, r4
 5  """
 6
 7  EXPECTED_RESULTS = {
 8    "r0": 0,
 9    "r4": 0x00001020,
10    "r5": 0x00001020,
11    "sregZ": 0
12  }
```

Listing 4.10: A test for the AND instruction (test AND_f1_2).

In lines one to four, the code of the test is written down. The test first moves the value 0x1020 into the `r4` register and then moves the same value into the r5 register. Next, the AND instruction is used in line 4. Because `r4` and `r5` contain the same value, it is expected that `r5`, as the destination of the operation, still contains this value after the test. Furthermore, the zero-flag of the status register should hold the value `0`, as the result of the operation **is not** `0`, but `0x1020`. The expected results are given in line 7 to 11. Because `r0` is not filled with any value in the test, it should contain the value `0` after the test. Theoretically, it is not necessary to include this register, however, we did so, to show that the `r0` register is not changed by the involved instructions.

The tests are generated with a python3 script. The script first generates a file that only contains assembler code that is needed for the test. To do so, the script first writes a preamble to the assembler file. The preamble contains the section header and other information that is needed to assemble the binary file for the test. Also, a postamble is added to the test. The postamble consists of an unconditional branch that forces QEMU to output the CPU status after the test code was emulated. This output is later used to evaluate the test results.

The preamble, the assembler code, and the postamble are combined by the script and stored in a separate file. The script then calls an AVR32 assembler that builds an executable ELF file out of the source file. Thereafter, a second script is called that reads the ELF files section table and extracts the *.text* section from the file. The .text section is then written to another file. This step is necessary, because our QEMU implementation can not load ELF files for now and hence needs raw binary input.

Finally, QEMU is stated with the extracted binary file as input. The script reads the QEMU output that contains the register contents and compares them to the expected results that are loaded from the test case file. At last, the script prints out if the test was successful or not.

The described process was automated with a wrapper script that performs all necessary steps. The wrapper scripts allows the user to name a single test case that should be executed or to execute all tests at once. We will evaluate the testing framework in section 6.5.

5 Fuzzing the OPS-SAT

In this chapter, we describe how we used fuzzing to find security vulnerabilities in the OPS-SAT firmware. First, we discuss what parts of the firmware are a potential target for fuzzing. Thereafter, we explain how we integrated the fuzzing into the QEMU emulation. In the last section, we cover how we could improve the fuzzing input seeds, so that relevant code segments would be executed in a reasonable time.

5.1 Fuzzing Target

As the goal of the security assessment is to execute arbitrary code on the OPS-SAT, we decided to focus the fuzzing on functions that are involved in the communication with ground stations. The security of satellite communication links is a better-researched area of space security, hence we will take up this research and continue it on the satellite itself [PM20].

To identify functions that read input data from an external source, we performed a static analysis of the firmware's ELF-file. The ELF-file contains symbols for the functions and objects, so that we can search for function names that imply an association with input handling. For example, a function name that includes the substring *_rx* suggests that the function is responsible to receive data.

By searching for different words, we found multiple functions that are candidates for the fuzzing.

The first function that comes into question is the `CSP_ServerCycle`. It reads a space packet from an input connection [fSDS20]. However, there is no direct interaction with the contents of the packet, as it is redirected to another location in the end. Because of this, we decided to not focus on this function if other functions can be found that perform operations on the received input data.

Another function that reads input data is `can_rx_task_gmv`. This function reads packets from the CAN bus and writes them into different buffers. The OPS-SAT receives space packets via the CAN bus, therefore this function could be of interest. However, the function is rather complex and we did not implement a virtual CAN device that is capable of providing the needed input. Hence, we will not include this function in the fuzzing process.

There is also the `csp_i2c_rx` function that is used to read a space packet from the I2C bus. We did not include this function in the fuzzing, because it is used by the `i2c_ISR` function. This function is an interrupt handler, therefore we would need to implement another virtual device that generates interrupts. For example, we could start a new thread that generates an interrupt every n milliseconds and then sends the fuzzing input through the virtual I2C interface. However, this approach would increase the complexity of the fuzzing loop and hence was dismissed for the practical phase.

We decided to use fuzzing against a function that directly works on the input data, without any hardware interaction. Such a function is the `TCTA_Cycle`. The `TCTA_Cycle` reads a space packet from an input buffer and constructs a *telecommand* from it. The telecommand is then executed by a telecommand handler. Depending on the performed operation, a response packet is send. The `TCTA_Cycle` is a *cyclic* task that is executed until the firmware is restarted. Because the input data is read from a buffer, no additional virtual hardware needs to be implemented for the fuzzing.

In the following parts of this section, we will describe the structure of a telecommand and how the OPS-SAT reads and executes telecommands.

5.1.1 Telecommand structure

The basis of any telecommand is a space packet [fSDS20]. The space packet can be loaded from the data structure `sppBuffer` or the CAN bus.

A space packet starts with a 6 bytes long primary header. The header contains various metadata about the packet. By reverse engineering the `TCTA_Cycle` functions, we determined that the OPS-SAT firmware expects certain values to be set in the upper two bytes of the header.

Table 5.1: Telecommand primary header, upper two bytes

Header values expected by the firmware.

Offset	**Size**	**Expected Content**	**Description**
0xd	3 bit	0	Packet Version Number
0xc	1 bit	1	Packet Type
0xb	1 bit	1	Secondary Header Flag
0x0	10 bit	1010	Application Process ID

The Packet Version number is expected to be zero. The Packet Type bit determines if a space packet is a telemetry packet or a telecommand packet. A telecommand packet is identified by a `1` bit in the Packet Type flag. The Secondary Header Flag determines if the packet has a second header that can be filled with implementer defined data. The OPS-SAT expects space packets to contain a secondary header, therefore the flag must be set to `1`. The Application Process ID is used to identify which application a space packet needs to be routed to. A telecommand space packet needs to have the value `0xa`. Overall, the first two bytes of every telecommand space packet need to have the value `0x180a`.

Following these two bytes, a space packet's primary header contains a Packet Sequence Control Field. This field is not evaluated in the `TCTA_Cycle` and therefore was not in focus during our work. The last two bytes of the primary header contain the size of the user data field. At different locations in the `TCTA_Cycle`, the firmware checks if the user data field is not longer than `0xff` bytes.

The secondary header of a telecommand space packet is 0x34 bytes long. The first few bytes of this header are used to determine the telecommands service area and operation. This information is 7 bytes long and evaluated by multiple functions. At offset 0x30 in the telecommand, there is a priority byte that needs to have the value `0xc0`.

After the secondary header ends, the user data field starts. A regular user data field can be up to 0xff bytes long. One important value in this field can be found at offset 0x144. At this location, a half-word contains the size of the user data field that is set after the space packet was converted into a telecommand.

Table 5.2: Structure of a telecommand

Offset	**Size**	**Description**
0x8	4 bytes	Status value
0xc	4 bytes	Status value
0x10	6 bytes	Space Packet primary header
0x18	-	Start of Space Packet secondary header
0x1a	2 bytes	Service area
0x1c	2 bytes	Service
0x1e	2 bytes	Operation
0x20	2 bytes	Area version (only upper byte used)
0x30	1 byte	Packet priority
0x34	4	Timestamp
0x44	-	Start of user data

Every telecommand has an additional status header that is located before the primary header of the space packet. This header is 0x10 bytes long and contains two important integers. At offset 0xc is a status number that determines if the telecommand was just received, passed a check as valid, or is currently executed. At offset 0x8 is another status value, however, it is only checked at two locations. The status value at offset 0xc is updated after every execution step.

A telecommand is stored in the `gTelecommandsInProcess` array that is 1000 bytes long. Because every telecommand is 0x14c bytes long in total, the array can hold up to 20 telecommands.

5.1.2 TCTA_Cycle program flow

The `TCTA_Cycle` consists of 4 functions:

- TCMA_CleanTCBuffers
- TCMA_ReadCommand
- TCMA_VerifyCommand
- TCMA_ExecuteCommand

In the following subsections, we will explain the relevant parts of these functions. Because we simplified some aspects of the command handling during the fuzzing, we will focus on the function parts that are relevant to understand later parts of this thesis. A graphical overview of the `TCTA_Cycle` is shown in Figure 5.1.

The first function is responsible for the cleanup of a telecommand's memory area. Before a new telecommand is loaded from an input buffer, this function sets every byte in the telecommand's status header and the user data area to zero.

5.1.3 Reading a Telecommand

The second function in the cycle, `TCMA_ReadCommand`, is used to read a space packet from an input buffer and copy it into the telecommand array. To do so, the `COMTT_GetReceivedTCPacket` function is called. The first argument of this function call is a pointer to the `gRawPacketBuffer`, where the space packet will be copied to. The second argument of this function call is the value `0xff`. It is used as a maximum size value.

`COMTT_GetReceivedTCPacket` first tires to load a space packet from the space packet protocol buffer (`sppBuffer`) and then writes it into the `gRawPacketBuffer` by calling the function `CSP_GetReceivedPacket`. If `sppBuffer` contains a bigger size value than 0xff, the packet is not copied to the telecommand array.

If no space packed was copied from `sppBuffer`, `COMTT_Get ReceivedTCPacket` tries other options to load a space packet. One of these options is a call to the function `CAN_GetLastFrame`. This function loads a space packet from the `CanStore` to the memory area that is passed as a call argument. It should be noted that the size of the space packet in the `CanStore` is **not** checked against the value 0xff. In theory, `CAN_GetLastFrame` will copy packets to the `gRawPacketBuffer` that are longer than 0xff bytes.

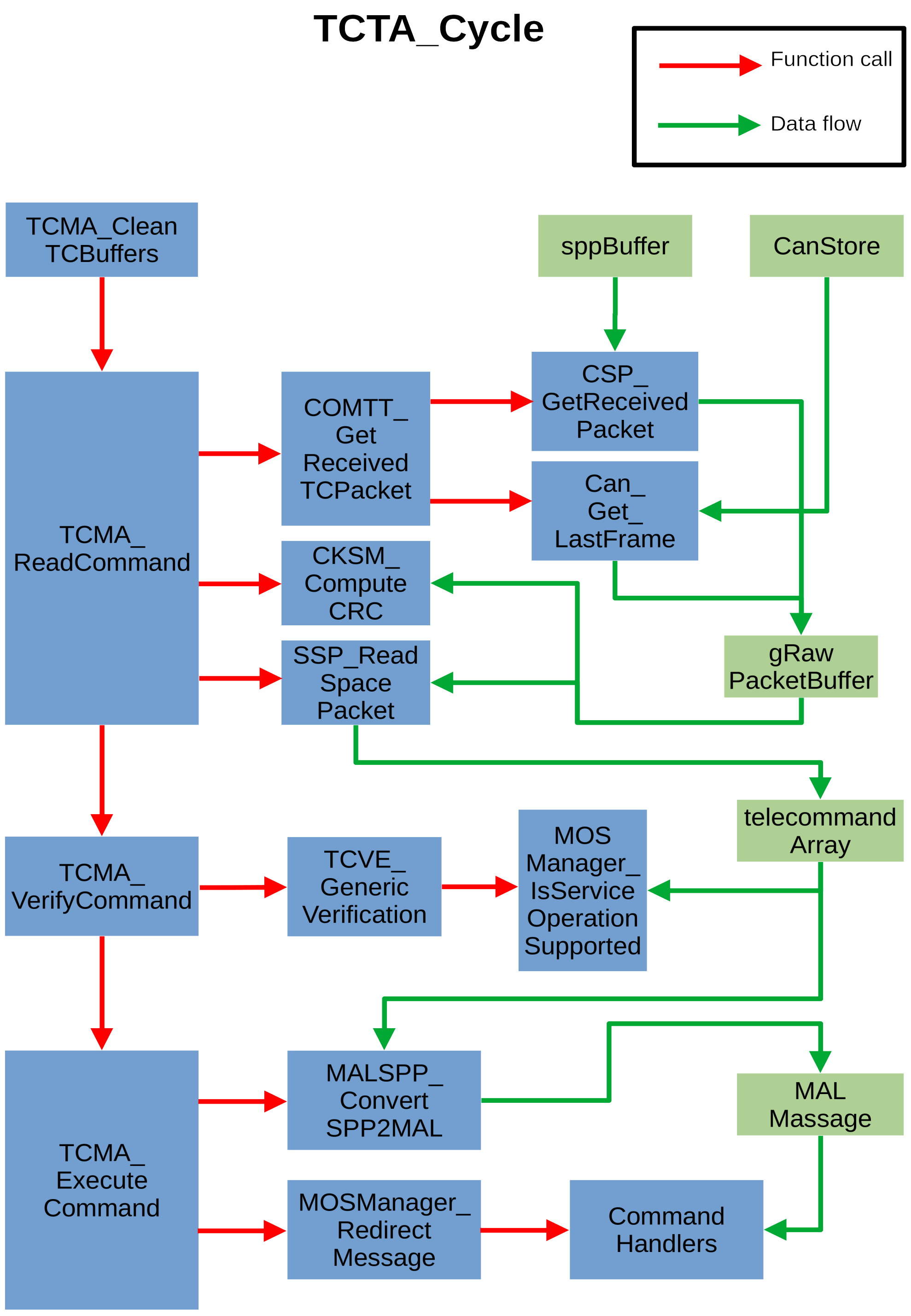


Figure 5.1: Simplified overview of the execution and data flow in the `TCTA_Cycle` The red lines represent function calls, the green lines represent the data flow. Return values or function arguments are not shown. The blue boxes represent functions, the green boxes represent data buffers. Some aspects of the `TCTA_Cycle` are left out, to simplify the illustration.

If `COMTT_GetReceivedTCPacket` returns and a packet was copied, a Cyclic Redundancy Check (CRC) of the space packet is calculated and compared to the last two bytes of the copied packet. If the values are matching, the function `SPP_ReadSpacePacket` is used to copy the space packet from the `gRawPacketBuffer` to the telecommand array. The last 2 bytes, the CRC check value, are left out. After the user data is written to the telecommand, the size of the user data area is stored at offset 0x144 in the telecommand.

5.1.4 Telecommand verification

The `TCMA_VerifyCommand` function is used to determine if a telecommand is valid. The contents of the telecommand are checked in the `TCVE_GenericVerification` function. The first operation is to check if the telecommands primary header starts with the value `0x180a` and if the application process identifier is `0xa`. It is also checked if the value at offset 0x30 is equal to `0xc`. If these checks are passed, the values for the service area and the service, as well as the operation and service area version from the secondary header, are passed to the function `MOSManager_IsServiceOperationSupported`. If this function returns the correct result value, the telecommand contains a requests for a valid operation and `TCMA_VerifyCommand` returns.

5.1.5 Telecommand execution

The last function in the `TCTA_Cycle` is `TCMA_ExecuteCommand`. If the status integer of a telecommand equals `4`, the space packet from in telecommand is passed to the function `MALSPP_ConvertSPP2MAL`. Inside this function, several values from the telecommand are copied to a *Mission Operations Message Abstraction Layer* message object at a fixed memory location [fSDS15]. The user data field of the space packet is copied to the message body field. If these operations are successful, the message is passed to the `MOSManager_RedirectMessage` function. Here, the service area and the operation in the message are evaluated and the message is send to the respective telecommand handler. There are 61 telecommand handlers that are grouped into multiple service areas. Some of the handlers are protected and will operate if the `MOS_Globals_isCriticalCommandEnabled` flag is set to `1`.

Most of the telecommand handlers will build another message object that is converted into a space packet and send back with the results of the operation. After `MOSManager_RedirectMessage` returns and if no other telecommand is ready for execution, `TCMA_ExecuteCommand` returns and the `TCTA_Caycle` starts again.

5.2 QEMU Fuzzing Integration

We decided to use AFL++ as a fuzzing tool for this thesis. The version of the QEMU build that we used for this thesis is 6.0.93. Currently, there is no support for this version in AFL++ or classic AFL. Therefore, we needed to implement a connection to AFL into QEMU. The `avr32a_cpu_class_init` function was patched to perform a call to a setup-function that sets up the fuzzing integration after the virtual CPU is initialized. The setup-function prepares a shared memory area, where coverage information is written to later. The function also informs AFL that the fuzzing target is ready.

We decided to inject the fuzzing test cases directly into the telecommand array, to reduce the complexity of the emulation and avoid the emulation of peripheral communication devices. The test cases were injected just before the `TCMA_VerifyCommand` function is called. By including the `TCMA_Verify Command` function in the fuzzing loop, only test cases that passed the formal validity tests were executed. This is a trade-off between low complexity and realistic inputs.

To inject the fuzzing input, we added a patch to the MCALL instruction. MCALL is used to perform a function call. If MCALL is executed at address `0xd00a0652`, a QEMU-helper function is used to read a new test case from AFL. This test case is then written into the start of the telecommand array at address `0xd0ec2f10`.

```
if(ctx->base.next_pc == 0x0xd00a0652){
  gen_helper_fuzzer_insert(cpu_env);
}
```

Listing 5.1: The patch that is applied to the MCALL instruction.

The patch first checks if the current program counter is equal to `0xd00a0652`. If that is the case, the helper function is called.

We also implemented a patch that changes the value of the `MOS_Globals_isCritical CommandEnabled` flag to `1` for every testcase, to ensure that protected telecommands are also executed. Otherwise, it is unlikely that the AFL produces one input that enables critical commands and then another input that executes a critical command, before a reset of the emulation occurs.

AFL needs to be informed when the execution of the fuzzing target ended in a regular manner, without a crash. The regular end of an application is usually indicated by the exit code `0`. Because our emulator does a full system emulation, the firmware has no regular 'exit point', where it stops with an exit code like a regular program would do. We decided to treat the end of the `TCTA_Cycle` as the execution end for this purpose. We added a patch to the POPM instruction at address `0xd00a065a`, where the `TCTA_Cycle` ends. The patch notifies AFL that the execution ended without an error. We expect that a crash would prevent the firmware from reaching the end of the `TCTA_Cycle`. As the `TCTA_Cycle` task is executed periodically, the execution will continue at the start of the `TCTA_Cycle` later.

To ensure that unnoticed errors, which were caused by the fuzzing, are not persistent, we added a call to QEMU's `cpu_reset` function after every 10,000 executions. The call causes QEMU to restart the firmware. Because the firmware setup needs about 2 seconds to start the `TCTA_Cycle`, the reset was not done after every execution, to improve the performance. Also, it is likely that a test case will cause an application logic error that is not immediately resulting in a crash and only becomes active after some time. Errors like this would not be noticed, if the emulation was resetting after every execution. Another aspect is that the defined 'execution end' might be too early for some errors to come into effect.

Because of the full system emulation, the firmware will not end with an error exit code if a crash occurs. The QEMU emulator will only exit with an error if an illegal instruction was found. Therefore, we needed to implement another QEMU-helper that was added to the RJMP instruction that performs a relative jump. The helper is executed if the RJMP instruction is used at any of the addresses listed in Table 5.3. The functions in Table 5.3 are only executed if the firmware notices that an error occurred. The QEMU-helper then reports a segmentation fault to AFL that will be recorded as a crash.

Table 5.3: Crash handling functions.

Function	**Address of crash helper injection**
cpu_reset	0xd00c307e
vApplicationStackOverflowHook	0xd00c460e
exit	0xd00c4682
do_unknown_exception	0xd00c42ca

If one of the above addresses is reached, this may indicate that the firmware left its regular execution path because an error occurred. During a normal execution, without any fuzzing input, non of these functions are called.

5.2.1 Coverage Tracing

AFL needs information about the program's execution path to conclude how an input changed the program's behavior. We added a patch to the `avr32_tr_tb_start` function that is called at the beginning of every QEMU translation block. The patch calls a QEMU-helper that writes the current program counter address to AFL's shared memory map. To reduce the amount of information that is send to AFL, the helper function only writes data into the memory map during the execution of the `TCTA_Cycle`. The reporting function can also be used to output detailed information about the program flow, for example, the arguments of a function call can be written to a file. In a future update, the insertion of the fuzzing test cases could also be implemented in this function. This way, the *translate.c* file doesn't need to be changed if the fuzzing target is changed. With this technique, the code complexity is reduced, as the fuzzing functionality is fully capsuled in a separate file.

Another requirement for coverage tracing is to be able to evaluate the covered code blocks later. The first approach to do this was to simply write every code edge, which means the previous program counter address and the current program counter address, to a file. This approach was used for debugging during the implementation of the AVR32 emulation. However, this came with a tremendous performance impact, as the emulation creates thousands of edges per second. In general, input- and output-operations are relatively time-consuming. The resulting logfile also gained multiple gigabytes in size within minutes.

To circumvent that issue, we created a hash table. For every edge that is reported, it is first checked if the source address is already stored in the hash table. If that is not the case, the source address is added to the table and associated with an array. The array holds the destination addresses that are reached from the source address. The destination of the edge was then added to the array. This way, it can be quickly evaluated whether an edge was already reported or not. This structure also allows one basic block to have multiple destination addresses. For example, the ICALL instruction jumps to the address that is provided by a register. A code segment that dynamically calls the start address of tasks could have more than 2 destination addresses. If an edge was already stored in the described data structure, no further action was taken. If an edge was reported for the first time, it was written to a log file. This implementation seemed to have no considerable negative impact on the performance because every edge was only written to a file once. The separate log file can later be used to evaluate the reached code blocks.

5.3 Input Seeds

AFL needs at least one valid input for the target program. This input is used as a seed from which the test cases are derived. The test cases are then passed to the target application.

To test our fuzzing integration, we first used a simple seed that consisted of the alphabet in lower and upper case. Although the fuzzing integration was working as intended, the `MOSManager_RedirectMessage` function was not executed, even after several hours of runtime. Because the fuzzing input was not a valid telecommand, multiple checks in `TCMA_VerifyCommand` failed. Hence, the `TCMA_ExecuteCommand` function never executed a command, as the status bytes of the telecommand had the wrong values. Because of the checks that are performed by the firmware, AFL needs to calculate specific values for multiple bytes in the telecommand, to trigger the execution of a command. These bytes are at least the two status integers, the primary headers upper two bytes, the service, the service area, the operation and the priority byte. In total, AFL would need to find 10 bytes that need be have specific values at the same time. With 2^{80} possible combinations, this is likely not going to happen for a long time.

To improve the fuzzing inputs, we used the insights that were gained from the reverse engineering of `TCMA_VerifyCommand`. As a result, we came up with 18 input seeds. In each seed, the status bytes were set to the value that is expected in `TCMA_VerifyCommand`. Also, the primary header and the priority byte were set to the expected values. Additionally, we added correct values for the service area, service, operation and service area version fields that are evaluated in `MOSManager_Redirect Message`. These values were different for every input seed, so that AFL has a valid input for different telecommand handlers. With this updated input seeds, `MOSManager_Redirect Message` was called and telecommands were passed to the command handlers.

6 Evaluation

In the evaluation chapter, we first analyze the coverage that was obtained during the fuzzing and we also look at the performance of our implementation. Following that, we describe a vulnerability that was found by fuzzing the OPS-SAT firmware. We also explain how we were able to find an exploit for this vulnerability. Additionally, we test our emulator in a case study, where we verify a vulnerability that was already known.

6.1 Fuzzing coverage

This section explains how the coverage varies for different fuzzing approaches. First, we show that fuzzing without suitable input seeds is not efficient. Next, we explain the results for fuzzing with optimized input seeds. We also selected certain command handlers in particular for a second fuzzing-run to improve their coverage. The results of this approach are evaluated in the third subsection. At the end of the section, we summarize our results.

6.1.1 Initial coverage

To test the AFL integration, we first started AFL with one input seed that consisted of the alphabet in lower and uppercase. While the AFL integration worked flawlessly, AFL was not able to generate input that passed the checks in the `TCVE_GenericVerification` function, even after multiple hours of runtime. Therefore, the coverage in the relevant parts of the firmware was nonexistent.

6.1.2 Coverage with optimized seeds

As we explained in section 5.1.4 and section 5.3, the firmware checks multiple bytes in every telecommand to determine if the input is valid. After we used optimized seeds as input for AFL, QEMU started to execute the telecommand handlers in the `MOSManager_RedirectMessage` function after a short amount of time.

In total, AFL found 56377 edges after 7 days of fuzzing. The majority of these edges were discovered within the first 48 hours. Because the virtual CPU receives an interrupt from the virtual timer every 100 microseconds, the coverage contains many edges to and from the interrupt handler that are not part of the regular execution flow. If these edges are removed from the coverage log, 35852 edges remain. During a timer interrupt, the firmware executes the `vTick` function. In some cases, a context switch happens during the execution of `vTick` and QEMU reports an edge from `vTick` to a regular function. If these edges are also removed, 26325 unique edges remain in the coverage. The coverage was only recorded between the insertion of the fuzzing input and the end of the `TCTA_Cycle`. Therefore, edges that occur before and after the `TCTA_Cycle` are not considered here, except, when they are part of a task that became active during the execution of the `TCTA_Cycle`. Inside the `MOSManager_RedirectMessage` function, every telecommand handler was executed.

Table 6.1: General fuzzing coverage

ID	**Telecommand Handler**	**Covered blocks**	**Percent**
1	INVOKE_GetGPSData	15/18	78%
2	INVOKE_ReadI2CBus	9/9	100%
3	INVOKE_ReadI2CPayload	9/9	100%
4	INVOKE_WriteI2CBus	9/9	100%
5	INVOKE_WriteI2CPayload	9/9	100%
6	PUBSUB_MonitorEvent	6/6	100%
7	REQUEST_AGGR_AddDefinition	53/78	67%
8	REQUEST_AGGR_GetValue	40/58	68%
9	REQUEST_AGGR_ListDefinition	33/47	70%
10	REQUEST_AddDefinition	14/15	93%
11	REQUEST_AddParameterCheck	7/7	100%
12	REQUEST_CheckMemory	9/9	100%
13	REQUEST_GetCurrentTime	5/6	83%
14	REQUEST_GetTimeMode	7/7	100%
15	REQUEST_ListDefinition	46/60	76%
16	REQUEST_ListOperation	8/20	40%
17	REQUEST_PerformHealthCheck	19/22	86%
18	REQUEST_TestFile	25/55	45%
19	REQUEST_WriteFile	61/133	45%
20	SEND_Alive	3/3	100%
21	SEND_GoToMode	12/12	100%
22	SUBMIT_AGGR_EnableGeneration	32/34	94%
23	SUBMIT_AGGR_RemoveDefinition	27/33	81%
24	SUBMIT_AGGR_UpdateDefinition	45/62	72%
25	SUBMIT_ALERT_EnableGeneration	66/71	92%
26	SUBMIT_ApplyManualTimeShift	12/13	92%
27	SUBMIT_ClearFromTime	15/16	93%
28	SUBMIT_ClearMemory	4/5	80%
29	SUBMIT_CreateFile	33/40	82%

Table 6.1: General fuzzing coverage

ID	**Telecommand Handler**	**Covered blocks**	**Percent**
30	SUBMIT_DeletePacketStore	4/5	80%
31	SUBMIT_DownlinkPacketStoreContent	2/2	100%
32	SUBMIT_EnableCheck	84/101	83%
33	SUBMIT_EnableCriticalCommands	11/12	91%
34	SUBMIT_InsertOperation	13/17	76%
35	SUBMIT_LoadMemory	10/11	90%
36	SUBMIT_PatchFile	5/17	29%
37	SUBMIT_PowerOff_SBandRX	6/6	100%
38	SUBMIT_PowerOff_SBandTX	6/6	100%
39	SUBMIT_PowerOn_SBandRX	6/6	100%
40	SUBMIT_PowerOn_SBandTX	6/6	100%
41	SUBMIT_PowerOn_XBandTX	6/6	100%
42	SUBMIT_Powercycle	36/39	92%
43	SUBMIT_RemoveDefinition	79/95	83%
44	SUBMIT_RemoveFile	32/46	69%
45	SUBMIT_RemoveParameterCheck	79/95	83%
46	SUBMIT_ResetADCS	3/4	75%
47	SUBMIT_ResetEPSWatchdog	2/2	100%
48	SUBMIT_ResetOBSW	6/6	100%
49	SUBMIT_SetADCSMode	19/28	67%
50	SUBMIT_SetAccessMask	55/59	93%
51	SUBMIT_SetBootImage	23/25	92%
52	SUBMIT_SetManualTime	12/13	92%
53	SUBMIT_SetNavTransMode	14/16	87%
54	SUBMIT_SetPowerState	11/52	21%
55	SUBMIT_StopPacketStoreDownlink	7/7	100%
56	SUBMIT_SubmitAction	15/22	68%
57	SUBMIT_UpdateDefinition	72/89	80%
58	SUBMIT_UploadOrbitTLE	35/37	94%
59	SUBMIT_UseAutomaticTime	8/8	100%

For 32 of the 59 telecommand handlers, a high coverage above 90% was achieved. However, 5 handlers were only covered to less than 50%. Three of these handlers, `SUBMIT_PatchFile`, `REQUEST_TestFile`, and `REQUEST_WriteFile`, are working on files that are stored on the flash chip. Because the flash chip is not fully emulated, some conditional operations that depend on the fash chips content fail and some of the following branches are never taken. Also, some checks of the input data fail, as specific values that were not found by AFL are expected at certain points. The other two handlers, `REQUEST_ListOperation` and `SUBMIT_SetPowerState`, have low coverage because parts of the input are checked against specific values that were also not found by AFL.

For some of the other command handlers with coverage below 100%, we manually investigated why certain basic blocks were not executed and if they possibly contain unsafe operations. In some command handlers, multiple larger basic blocks are missed in sequence, for example in `INVOKE_GetGPSData` or `SUBMIT_EnableCheck`. In both cases, the handler tried to perform hardware interactions. The `INVOKE_GetGPSData` handler tries to read the GPS power status. Because the GPS device is not emulated, the resulting value is null and a conditional branch is not taken. The `SUBMIT_EnableCheck` handler tries to read data from the FRAM chip. Because only a very basic emulation of the chip was implemented and the actual contents are not known to us, the resulting value is also null and multiple basic blocks are not executed, including function calls.

Multiple smaller basic blocks were not executed because the fuzzing input data missed values at specific locations that are needed for conditional branches. However, these basic blocks only contain one or two instructions, for example, to set a function call argument in a register.

6.1.3 Targeted coverage

Because some telecommand handlers with a high number of basic blocks had lower coverage, we decided to redo the fuzzing for them in a more targeted approach, where only one command handler was executed in each fuzzing run. To do so, we set the content of the telecommand's status integers and the space packet's primary header to fixed values that would trigger the execution of the telecommand. Additionally, the values for the service area, service and operation were set in the secondary header, so that the targeted telecommand handler was called. The fuzzing input was also injected right before the `TCMA_ExecuteCommand` function started, as the previous results showed that only the content of the space packet header was checked in `TCMA_VerifyCommand`. This way, the performance could be slightly improved, as unneeded function calls were prevented.

The fuzzing input was injected into the telecommand array and the maximum length for the input was set to 0x12a bytes so that only telecommands with a valid length were created by AFL. We also increased the timer interrupt speed, by setting the sleep duration to 50 microseconds. This resulted in a faster execution, with 310 to 350 executions per second. As only one telecommand was targeted, we could use multiple instances of AFL that worked in parallel on the same input seed.

The first telecommand handler that was fuzzed this way, was the `SUBMIT_SetPowerState` function. After running 6 AFL instances for 120 minutes, the coverage increased to 35 of 52 blocks, or 67%. The remaining, non-executed parts, are basic blocks in a for-loop that loops over the status of peripheral devices. Because those devices are not part of the emulation, the code inside the loop likely will not be executed and the remaining basic blocks will not be covered. It was also noticed that all fuzzing instances fall to a very low execution seed after less than 60 minutes. This is possibly caused by hardware interactions that wait for a result with a timeout. Because most hardware devices are not emulated, especially the power management system, this may cause waiting times that result in a low execution speed for the `TCTA_Cycle` task. However, the specific cause of this behavior needs to be further investigated.

For the command handler `REQUEST_ListOperation` the targeted approach resulted in no new edges after 2 hours of fuzzing by 12 parallel instances with 2.78 million executions each. Here, a specific value is expected that was not found by the fuzzing tool. Because other handlers seem to be more promising, we did not continue the fuzzing of that command handler.

Using targeted fuzzing, the coverage for the `REQUEST_ListDefinition` command handler could be increased to 51 of 60 blocks or 85%. The fuzzing was done by 12 parallel instances for 11 hours. Each instance performed 13.8 million executions. Some of the non-executed basic blocks will only be executed if certain values in the memory are set. These values are loaded from the connected flash device during the firmware startup. Because this device is not emulated, the blocks will not be executed.

For the `SUBMIT_SetADCSMode` handler, a coverage of 67% percent was reached after 30 minutes with 15 parallel fuzzing instances. This shows that the originally obtained coverage can be reached in a short time by the fuzzing implementation. However, the coverage did not increase anymore. After 2 hours and 45 minutes, with about 3 million executions per instance, no new code blocks were executed. Again, a possible reason is that the command handler calls functions that read data from non-emulated hardware. Therefore, it is unlikely that the coverage can be improved without implementing further virtual devices.

Table 6.2: Targeted fuzzing coverage

Telecommand Handler	**Original Coverage**	**Targeted Coverage**
SUBMIT_SetPowerState	21%	67%
REQUEST_ListOperation	40%	40%
REQUEST_ListDefinition	76%	85%
SUBMIT_SetADCSMode	67%	67%

The results of the second fuzzing run show that the coverage of specific code segments can be improved if a they are targeted by fuzzing in particular. However, this does not apply to code that depends on non-emulated hardware devices. Code segments that are only executed if a specific input was found by the fuzzing tool could still be executed, as it is only a matter of time until the fuzzing tool will have generated the needed input. Nevertheless, this might be very time-consuming and could be achieved in a more efficient way, if further optimized input seeds are used.

6.1.4 Summary

The fuzzing implementation and the defined input seeds resulted in an average coverage of 85,37% in the telecommand handlers. 39 of the 59 telecommand handlers were covered to more than 85%, only 20 telecommand handlers were covered to less than 85%.

In some cases, command handlers depend on data from peripheral devices. In such cases, some basic blocks may never be executed, because the respective device is not emulated.

Some basic blocks were not executed because they expect specific values that AFL did not provide. However, it is possible to improve the coverage of such command handlers by fuzzing them in particular with specific input seeds.

6.2 Performance

The performance of AFL and the emulation highly depends on the timer interrupt interval. A shorter interval results in a faster execution speed. However, a very short interval results in a lower execution speed. The reason for this is that a long timer interval results in fewer timer interrupts. If no task is in the *ready* state, the firmware executes the *idle task* until another task becomes ready. Because this often depends on the amount of passed time, fewer tasks are executed. If more timer interrupts occur, waiting times pass faster, as they are measured in ticks and not in real-time. Hence, a short timer interval results in an overall faster execution. But, if the timer interval is too short a task will be interrupted just after it became active and only a short segment of the task's code will be executed. By experimenting with different timer intervals, we found out that the `TCTA_Cycle` has the highest execution speed if an interval of 50 microseconds is used. In the first fuzzing phase an interval of 100 microseconds was used. For the targeted fuzzing, the 50 microseconds interval was used.

Most of the 12 AFL instances that were used to obtain the coverage with optimized seeds performed about 112 million executions within 7 days. The average execution speed is therefore about 182 executions per second. 3 instances had a significantly slower performance, with 12, 23 and 53 million executions. One instance performed about 96 million executions. We were not able to determine the exact reason for the difference in execution speed. AFL may generate input that under certain conditions results in a lower execution seed. For example, if the firmware waits for external devices to act with a timeout, before executing the `TCTA_Cycle` again, AFL will not receive a new execution signal from QEMU. Another possible reason is that the emulation contains a bug that provokes this behavior. However, we noticed a reduced execution speed during the fuzzing of the `SUBMIT_Set PowerState` command handler and regular execution speeds for the other three command handlers that were selected for targeted fuzzing. Therefore, it is more likely that certain command handlers may be the source for this behavior.

We noticed that AFL reports a very low *stability* value for all instances. *Stability* is a measurement that shows if an input results in the same execution path for multiple runs. For the 12 instances, AFL reported stability values below 1% without exception. The reason for this is that the timer interrupts stop the regular execution flow during every interrupt. QEMU then reports a new edge to AFL, from the current position to the interrupt handler. After the interrupt, QEMU reports a new edge from the interrupt handler to the previous program position. Because the timer interrupts do not happen at the same time for every execution, over time every basic block, outside a protected code segment, will have an edge to and from the interrupt handler. If the same AFL input is used in multiple executions, QEMU will therefore report different execution paths for every execution. It is likely that AFL generates unnecessary test cases that could have an influence on the fuzzing performance. Hence, the fuzzing performance could be improved if edges to and from the interrupt handler would no longer be reported to AFL. But, even then the stability will be below 100% because the firmware sometimes continues the execution at a different address after an interrupt. For example, if a task with a higher priority becomes ready after the timing interrupt, a new execution path results. Another reason for the low stability is that QEMU sometimes optimizes the immediate code and the start or end of translation blocks.

The performance for general fuzzing could be improved by a more targeted approach. The first 0x10 bytes of a telecommand are the same for every input and therefore could be set by the fuzzing integration functions instead of being part of the fuzzing input data. The same applies to other values that are the same for every telecommand, like the priority byte or the first two bytes of the primary header.

6.2.1 Specific Fuzzing Experiment

To determine, if the fuzzing performance could be further improved we experimented with different approaches. One of these approaches was to only execute the `TCTA_Cycle` and skip the other parts of the firmware.

For this experiment, we patched the QEMU implementation. When the end of `TCTA_Cycle` is reached, the emulation should not continue the execution as planned, but jump back to the start of the `TCTA_Cycle`. This way, the `TCTA_Cycle` would be executed in an endless loop, without a long interruption. This should result in an increased execution speed, as less time passes between the test cases.

To test this approach, we used 12 AFL instances that worked on different input seeds. As expected, the execution speed was drastically increased. Each AFL instance executed about 3000 test cases per second.

This experiment shows that a specific part of the firmware can be executed much faster if other parts of the firmware are left out. However, it needs to be considered that this kind of partial execution may lead to incomplete results. As we show in the following section, some errors that are caused by the fuzzing input only can be observed later in the execution. By only executing a specific part of the firmware, such cases will be missed. Therefore, this approach should only be used if an error is expected that certainly will become active inside the tested code segment, for example, a buffer overflow that results in a crash before the end of the tested code segment is reached.

6.3 Findings

In this section, we describe a vulnerability that was found during the fuzzing. Then, we explain how we developed an exploit for said vulnerability. We also cover how the vulnerability can be mitigated. In the last subsection, we evaluate the security of the telecommand handlers in general.

6.3.1 Memcopy bug

During the fuzzing phase, we noticed that the emulation started to behave strangely after some time. AFL was not producing any new output and in some instances the emulation was stuck in one of multiple endless loops. AFL also reported several crashes that were caused by one of the functions that we defined as *crash points*. After investigating these crashes, we found out that they were initiated by an error handling routine in the `xQueueGenericReceive` function. This function was executed after the `TCTA_Cycle` completed a full cycle and before the `TCTA_Cycle` was started again. Before the error occurs, the function checks if its first parameter, a pointer to a queue, is null. If so, the execution is stopped by the error handling routine.

By analyzing the execution path and the coverage, we could determine that the queue pointer is taken from the data structure `TaskInfo` that holds information about the active FreeRTOS tasks. The pointer is loaded in the `OTSK_CyclicTaskPattern` function and later passed to `xQueueGenericReceive`. After analyzing `OTSK_CyclicTaskPattern`, we concluded that this function is used to set metadata for cyclic tasks, such as the start and stop tick count or the delay until the task should start.

`OTSK_CyclicTaskPattern` also loads a code pointer that points to the start of a tasks program code from the `CyclicTaskInfo` data structure. This structure contains information about cyclic tasks. The code pointer is then called via the ICALL instruction.

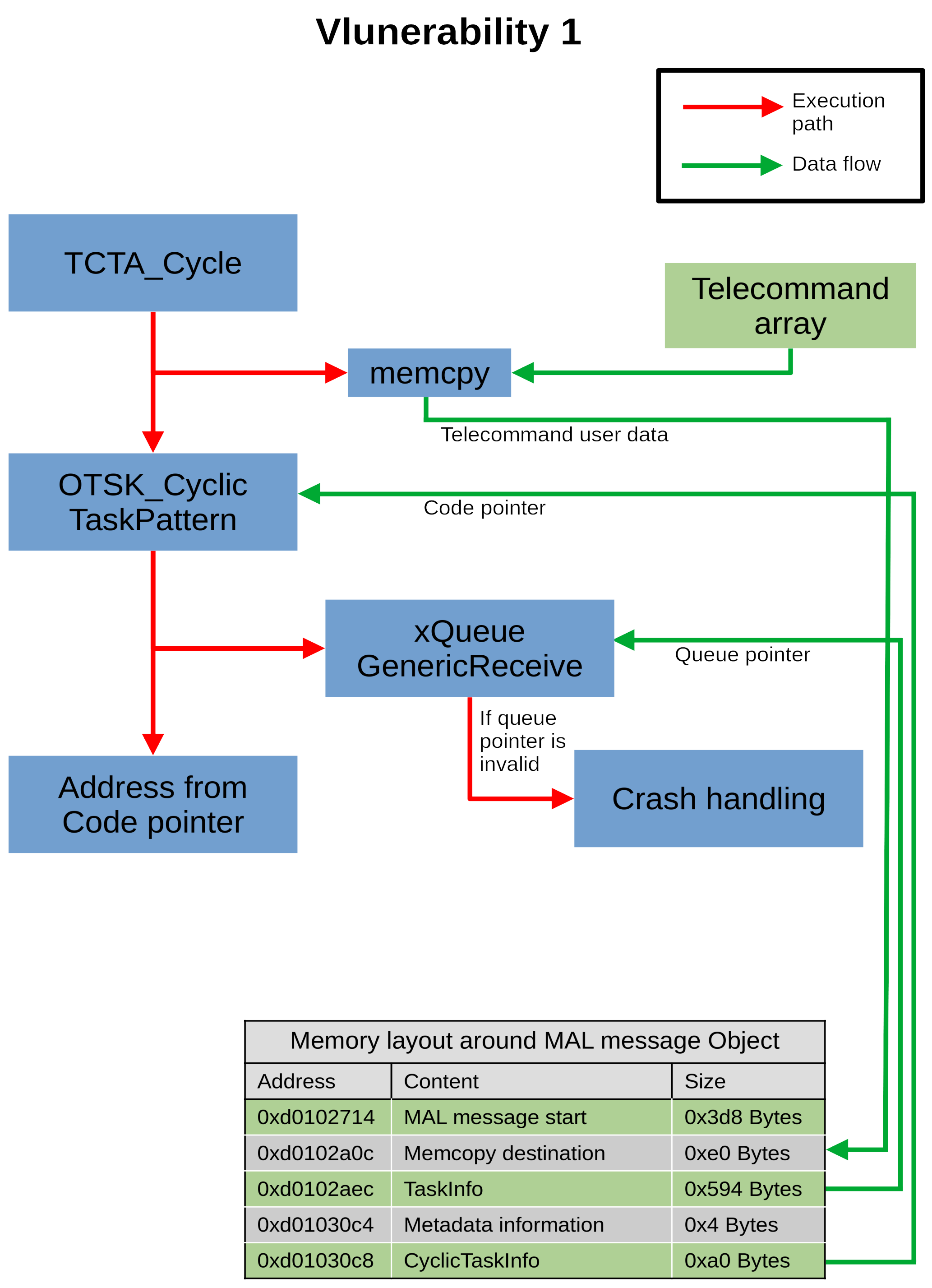
Next, we investigated the register contents during the execution of `OTSK_CyclicTask Pattern`. We found out that the queue pointer that is loaded from `TaskInfo` is null if the fuzzing input that AFL saved when the crash was reported is written into the telecommand array. If no fuzzing input is written to the telecommand array, a valid pointer is loaded and the error handling routine in `xQueueGenericReceive` is not called later on. Therefore, the invalid pointer is a result of the fuzzing input.

To find the code segment that changes the content of the queue pointer, we applied a patch to QEMU that loads the content of the pointer at the start of every basic block and writes it to a debugging file. That way, we could identify that the pointer is changed inside the `memcpy` function. This function copies the content of one memory area to another. By backtracking the execution path, we found out that the relevant function call is initiated inside the `MALSPP_ConvertSPP2MA` function. This function is used inside the `TCTA_Cycle` and builds a Message Abstraction Layer (MAL) protocol message from the space packet of a telecommand [fSDS15]. This process is illustrated in Figure 6.1.

The first argument of `memcpy` is the destination address. The relevant call uses a fixed address in the memory area that stores the MAL message. The second argument of `memcpy` is the source address. The source address that is used is the start of the user data segment in the currently processed telecommand. The third argument of `memcpy` is the size, more specifically the number of bytes that should be copied. This value is loaded as an unsigned half-word from the end of the user data segment in the currently processed telecommand.

The copy size value is directly taken from the user data segment and no checks are performed. Therefore, the user data may contain a size value that is larger than the size of the destination memory area. By implementing a memory dump in QEMU, we could confirm this assumption, as a memory area of the size that was set in the telecommand was overwritten by `memcpy`. The memory layout around this location is also shown in Table 6.3.

Because the memory area of the message object is not part of the stack of any task and the memory area is intended to be written on, no security mechanisms in FreeRTOS or the CPU can detect that values outside the MAL object are overwritten.



Memory layout around MAL message Object		
Address	Content	Size
0xd0102714	MAL message start	0x3d8 Bytes
0xd0102a0c	Memcopy destination	0xe0 Bytes
0xd0102aec	TaskInfo	0x594 Bytes
0xd01030c4	Metadata information	0x4 Bytes
0xd01030c8	CyclicTaskInfo	0xa0 Bytes

Figure 6.1: Function calls and data flow in `OTSK_CyclicTaskPattern`
To provide a better overview, the diagram is simplified and leaves out certain aspects, like function calls that are not necessary to understand the general concept. A detailed description is given in the text of section 6.3.1. The blue boxes represent functions, the green box a memory area. The execution path does not show function returns. The table shows the memory layout and content from Table 6.3.

The `TaskInfo` structure and the `CyclicTaskInfo` structure are inside the memory region that was overwritten, alongside other task-related information. Because the fuzzing input size was smaller than the size value in the user data segment, said area was overwritten with zeros for the most part. The queue pointer inside the `TaskInfo` structure was also overwritten with zeros because of this. Without valid task metadata, it is likely that the firmware continues to run with undefined behavior. This explains unexpected the behavior that we observed earlier.

Table 6.3: Memory layout of the `memcpoy` destination.

Address	**Content**	**Size**
0xd0102714	MAL message start	0x3d8 Bytes
0xd0102a0c	`Memcopy` destination	0xe0 Bytes
0xd0102aec	Metadata information	0x44 Bytes
0xd0102b30	`TaskInfo` structure	0x594 Bytes
0xd01030c4	Metadata information	0x4 Bytes
0xd01030c8	`CyclicTaskInfo` structure	0xa0 Bytes

6.3.2 Exploit prototype

With the possibility to overwrite a code pointer in the `CyclicTaskInfo` structure, the `memcopy` bug was an interesting candidate for further research. If the pointer can be overwritten with meaningful values, the execution of the firmware can be controlled. This part of the thesis explains how we developed an exploit prototype that uses a buffer overflow to overwrite said code pointer. The exploit allows us to take over the execution of the firmware in a laboratory setting. However, we were not able to prove that this exploit would work under real-world conditions, as the necessary hardware emulation was not implemented in our emulator. Further research needs to be done to investigate if the vulnerability can be exploited in practice.

We first analyzed a regular execution without fuzzing input to determine which cyclic task is executed after the `TCTA_Cycle` ends. This is again the `TCTA_Cycle`. As explained earlier, the code pointer for a task is loaded from the `CyclicTaskInfo` structure and then called with the ICALL instruction inside the `OTSK_CyclicTaskPattern` function.

Because one of the fuzzing inputs that triggered a crash overwrote the `CyclicTask Info` structure, we assume that it is possible to overwrite the code pointer with a value of our choice. To do so, we need to inject a payload that contains meaningful values for the task metadata structures into the telecommand array. Only the code pointer needs to be modified. Because a single telecommand is not long enough for such a payload, we need to overwrite multiple telecommands. The firmware does not verify if a telecommand is too long, therefore this would not trigger any error during the execution of the exploit. For this proof of concept, we also assume that any checks during the telecommand's construction are passed.

To build a payload that can overwrite the code pointer, we first use padding bytes to fill the memory area of the MAL message. Then, we append data from a memory dump that contains legitimate values of the metadata information that is stored after the MAL message's memory area. We only apply two changes to this data. First, we set the size value that is used by `memcpy` to the length of the payload. Second, we change the value of the code pointer to an invalid address. This should result in an *illegal instruction* exception in QEMU that indicates that the modified code pointer was loaded into the program counter. The payload was injected by implementing a loading function that loads input data from a file and then writes it to the telecommand array.

After executing the emulator with the above modifications and with the payload as input, the execution path logging showed that the `TCTA_Cycle` was left in a regular manner and no exception occurred in the following functions. During the next execution of the `OTSK_CyclicTaskPattern` function, the modified code pointer was loaded and called by the ICALL instruction. As a result, the emulation continued at the address that we defined in the payload. Because QEMU can not find any valid instruction at this address, it exited with an *illegal instruction* error, as was expected.

6.3.3 Exploit evaluation and mitigation

The constructed exploit was successfully used to change the execution path of the firmware. However, the exploit was directly written into the telecommand array. Therefore, we skipped any checks that would be done during the construction of the telecommand. When a telecommand is loaded into the telecommand array, one of the used functions verifies that the telecommand is not longer than 0xff bytes. Hence, our exploit would not be loaded, as it is significantly longer. However, the OPS-SAT firmware also tries to load a telecommand from the CAN interface if no command could be loaded from the I2C interface. In this case, no size check is performed during the `TCTA_Cycle`.

As can be seen in Figure 2.1, the SEPP is connected to the NanoMind board via a CAN line. Hence, it could be possible to send a manipulated space packet to the onboard computer from the SEPP. Because we did not focus on the peripheral hardware and did not emulate the CAN interface in a way that supports data transmission, this assumption needs to be verified in the future.

The `memcpy` bug led to multiple problems during the initial fuzzing approach. To circumvent further issues, we decided to patch the bug. To do so, we implemented a sidepatch in the implementation of the MCALL instruction in QEMU.

```
1   if(ctx->base.pc_next == 0xd00842b2){
2     TCGLabel* no_action = gen_new_label();
3     tcg_gen_brcondi_i32(TCG_COND_LT, cpu_r[11], 0xe0, no_action);
4     tcg_gen_movi_i32(cpu_r[11], 0xe0);
5     gen_set_label(no_action);
6   }
```

Listing 6.1: Sidepatch for the memcpy bug.

In the first line, we check if the current instruction is at address `0xd00842b2`, where the relevant call sequence of the bug starts. If this is the case, we define a new `TCGLable`. In line 3, we check if the size argument in register 11 is 0xe0 or lower. If this is the case, the memcpy-operation would not overwrite any data outside the MAL message object. Therefore, we perform a branch to the `no_action` label that is set in line 5. If the value in register 11 is greater than 0xe0, QEMU will execute the code that is generated by line 4. The frontend-operator that is used there overwrites the size argument with the value 0xe0, so that no values outside the MAL message object are overwritten.

After this patch was applied to QEMU, the fuzzing performed with much fewer issues. The number of hangs, endless loops, and unexpected behavior went down and was only observed in AFL instances with specific seeds. That way, the fuzzing tool could search for further crashes without interruption.

6.3.4 Telecommand handler security evaluation

After finding the bug described above, we statically analyzed how the telecommand handlers access the memory. We noticed that the handlers use special functions to access the MAL message object. For each data type a separate function is used, for example, there are functions to read integer values or functions to write boolean values. These functions perform size checks for their corresponding data type before doing the actual memory operation. This implementation prevents buffer overflows, like the one we identified.

After implementing the bug mitigation that we explained in the previous subsection, no new crashes were observed by the fuzzing tool. We only noticed a low fuzzing speed for certain telecommand handlers, but we assume that this is caused by timed operations or hardware dependencies that are not emulated.

6.4 Case Study - String Buffer Overflow

As mentioned in section 2.1.1, the OPS-SAT contains at least one more vulnerability. Therefore, we decided to investigate if this vulnerability can be found by the fuzzing implementation and if it can be verified by using the emulator. By doing so, we were able to verify the existence of not one but two bugs that can cause of buffer overflows. We were able to develop an exploit for one of the buffer overflows that was successfully tested with the emulator. The exploit allows us to take control of the firmware.

6.4.1 Vulnerability Verification

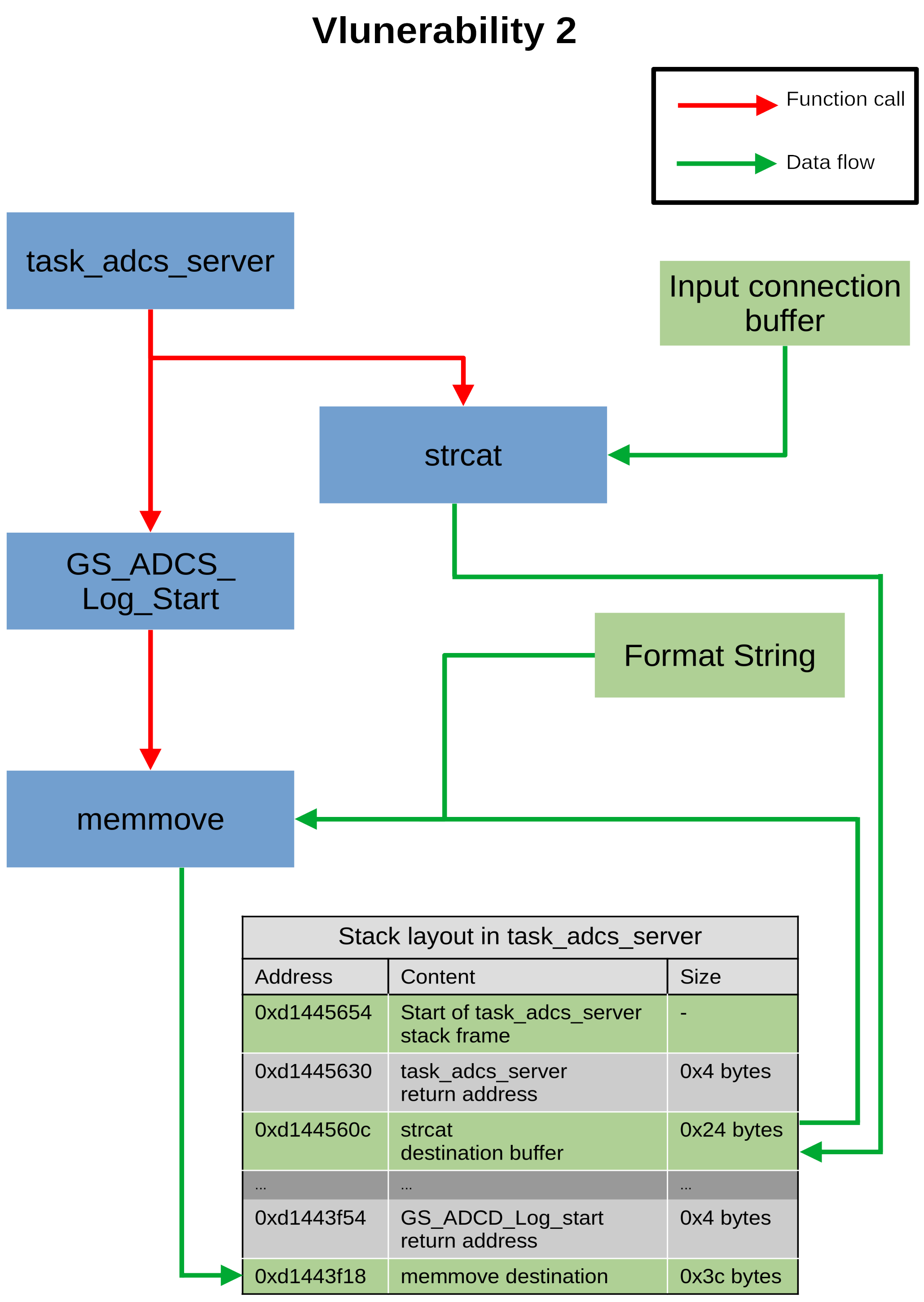
The vulnerability in question is a stack overflow in the `task_adcs_server` function. This function opens a socket for the CubeSat Space Protocol and reads a packet from an incoming connection [Chr]. Depending on the content of the packet, different actions are performed. In one case, a buffer of 0x24 bytes in length is created on the stack. We will refer to this buffer as *destination buffer*. Next, the `strcat` function is used to copy a string from the packet into the destination buffer. However, the firmware fails to do a size check before calling `strcat`. Hence, it is possible to overflow the buffer with a string that is longer than 0x24 bytes.

To verify the assumption that the destination buffer can be overflown, we injected a 0x30 byte long string into the packet. We also set other bytes in the packet, so that the execution would continue at the vulnerable code location. After emulating the firmware with this modification, the emulator crashed with an *illegal instruction* error at a memory address that is not inside the firmware's code section. This indicates that a code pointer was overwritten with an invalid address.

By exporting the memory region around the destination buffer, we could determine that the stack overflow was successful, as the memory area above the destination buffer was overwritten with the padding value. After further investigating the stack layout, we noticed that the return address of `task_adcs_server` is directly above the destination buffer and hence was overwritten by the buffer overflow. However, the task is a cyclic task and has no return statement. FreeRTOS sets a dummy return address for such tasks. Therefore, the bug in question can be used to overwrite values on the stack, but it is not the reason for the observed crash, as the return address is never used.

After further investigating the execution path, we found out that the last function that was executed before the crash is `GS_ADCS_Log_Start`. This function is called after `strcat` returns. The function initiates a sequence of other function calls, including multiple calls of `memmove`. The `memmove` function moves the content of one memory area to another area. These operations are illustrated in Figure 6.2.

The first `memmove` operation copies the input string from the destination buffer into a new buffer that is created just below the return address of `GS_ADCS_Log_Start` on the stack. Then, additional data is appended to the string in the new buffer. The



Stack layout in task_adcs_server

Address	Content	Size
0xd1445654	Start of task_adcs_server stack frame	-
0xd1445630	task_adcs_server return address	0x4 bytes
0xd144560c	strcat destination buffer	0x24 bytes
...	...	...
0xd1443f54	GS_ADCD_Log_start return address	0x4 bytes
0xd1443f18	memmove destination	0x3c bytes

Figure 6.2: Function calls and data flow in `task_adcs_server`
The blue boxes represent functions, the green boxes memory areas. The table shows the stack layout. To provide a better overview, the diagram is simplified and leaves out certain aspects.

relevant memory layout is also shown in Table 6.4. Depending on the length of the input string, the new buffer is overflown and one of the `memmove` operations overwrites the return address of `GS_ADCS_Log_Start`. If the input string is longer than 0x23 bytes, the return address is overwritten by fixed values from a format string. If the input string is longer than 0x3c bytes, the return address is overwritten with values from the input string.

Table 6.4: Memory layout in `GS_ADCS_Log_Start`

Address	**Description**	**Size in bytes**
0xd1445654	Start of `task_adcs_server` stack frame	-
0xd1445630	`task_adcs_server` return address	0x4
0xd144560c	Destination buffer for `strcat`	0x24
0xd1443f54	`GS_ADCS_Log_Start` return address	0x4
0xd1443f18	Destination buffer for `memmove` operations	0x3c

To test the above assumption, we injected a string into the CubeSat packet that consisted of 0x3c padding bytes followed by an address that was randomly chosen. The test resulted in a jump to said address, after `GS_ADCS_Log_Start` executed its return instruction. Hence, it is verified that the OPS-SAT firmware contains another vulnerability that can be exploited. It needs to be considered that we wrote the input data directly into the space packet buffer. Therefore, any functions that interact with the hardware were not part of the test and it needs to be investigated if the exploit would work in a real-world setting.

As mentions in section 2.2, FreeRTOS provides two mechanisms that can detect a stack overflow. However, the vulnerability only affects values inside the stack-frame of the task if no extensively large input string is used. Therefore, the security mechanisms of FreeRTOS cannot detect that a buffer was overflown. Hence, the implementer of a task needs to ensure that buffer sizes are checked before data is copied to the buffer.

6.4.2 Fuzzing Test

To evaluate if the fuzzing integration can identify the `memmove` vulnerability, we modified the fuzzing integration to inject the fuzzing input into the CubeSat space protocol packet. We reported the execution end to AFL just before the task finished one execution cycle and is executed again. The defined *crash points* were not changed.

The fuzzing integration worked as expected, but the execution speed that was reported by AFL was dreadful. AFL reported about one execution every 2 to 3 seconds. The reason for this is that firstly, the task uses functions with timeouts, like `cps_read`, which reads in the CubeSat packet. And secondly, the task maybe has a lower priority than the `TCTA_Cycle` and therefore is executed less often. To circumvent this issues, we used an optimized input seed. As AFL just needs to generate any test case that is at least 0x23 bytes long, a crash was found after a few executions. This result shows that the vulnerability at hand would also be found by using fuzzing.

In a real-world setting we would not have been aware that a vulnerability is present in `task_adcs_server` or how to trigger it. To meaningful perform fuzzing under this assumption, a modification of the firmware would be necessary. The priority of the task must be increased and operations with timeouts would need to be patched to remove the waiting time. This does not only apply to this function, but to all functions that use timeouts or have a low priority.

6.5 Testing framework evaluation

The test cases can be build and executed automatically by the *avr32test.py* script. The framework was used to define 470 test cases. For most instructions, multiple test cases were needed. This is, because we tried to test every edge case for every instruction. For example, for the ADD instruction, not only the results of additions with different numbers, like positive and negative numbers, were tested, but also the corresponding contents of the status register.

Approximately 35% of the instructions that were implemented before we started to use the testing framework contained errors that were found with the testing framework. The found errors were all implementation errors that resulted in a faulty application logic in the emulator. After we started using the testing framework, such errors were noticed during the implementation of new instructions. Hence, they did not stay unnoticed for long periods and did not influence the emulation. Some notable finds are:

- In the ASR instruction that performs an arithmetic shift to the right, the negative-flag of the status register was not set correctly. The ASR operation moves bit 31 of the result to the negative-flag bit, however, the result value was shifted 32 bit to the right, instead of 31.
- In the reverse subtraction instruction, the carry-flag of the status register was set, instead of the negative-flag.
- In the ANDL instruction, the upper half-word was not cleared, when the clear-flag was set.
- In the store byte instruction, the pointer address was not increased after the store operation was done.

All of the errors were not noticeable during the emulation of the OPSSAT firmware, as the emulation in general was running with expected behavior. The only reason to question the correct implementation of the emulation appeared when a branch instruction did not behave in the way it was expected and the emulation ended in a endless loop or crashed.

Building and executing all test cases takes about 15 seconds, therefore the framework seems to have a reasonable performance.

In conclusion, it is advised to use a test-driven approach, when a new architecture is added to QEMU. This seems even more useful when the implementation is done by a small number of people, without the resources for a quality testing team.

7 Discussion

In this section, we discuss the results of our research. We first look at the fuzzing of satellite firmware in general. Next, we discuss the security of the OPS-SAT, followed by a discussion of the hurdle and issues that we came across during our work. Finally, we name 3 lessons that we would like to have known before we started work to on this thesis.

7.1 Satellite firmware fuzzing

By using our AVR32 emulator, we were able to perform fuzzing on the OPS-SAT firmware and identify a vulnerability. Additionally, we verified a second vulnerability and showed that it can also be found with fuzzing tools. However, in the second case a modification of the firmware is necessary, because the fuzzing speed would be to slow to identify an unknown vulnerability in a reasonable time.

This results show that fuzzing can be used effectively to evaluate the security of satellite firmware, although some limitations need to be considered. We will further discuss this hurdles in section 7.3.2.

In general, the implementation of the emulator and the fuzzing connection was the most time consuming task during this thesis. The actual fuzzing needed significantly less time than the implementation and worked as we expected, besides some issues that are discussed later. The fuzzing of satellite firmware is comparable to the fuzzing of other embedded devices, as similar aspects need to be considered.

7.2 OPS-SAT firmware security

The firmware of the OPS-SAT contains at least two vulnerabilities that were confirmed in this thesis. Both vulnerabilities are the result of insecure memory operations, or more precisely missing input data validation. We showed that the vulnerabilities can be used to change the execution path of the firmware under laboratory conditions. Therefore, the OPS-SAT may be vulnerable to attacks with manipulated input, although it remains questionable how much afford is needed to exploit this in practice, as sending signals to a satellite is not a trivial task. Also, we did not determine how the receiver hardware on the satellite reacts to manipulated data, as we only focused on the firmware. Additionally, the firmware has another buffer that can be overflown. However, it is not possible to exploit this bug as no code pointer can be overwritten.

Missing input validation is one of the most common types of security vulnerabilities [MIT]. Buffer overflows are also a well-known and widespread security issue [LC03]. Modern operating systems provide various security features that defend applications against this type of vulnerability, for example, by using stack canaries. It was shown years ago that such defense mechanisms can also be applied to embedded systems [PW08]. However, newer research shows that buffer overflows are still one of the most relevant security issues in embedded systems [TM18]. Our results show that this is also true for CubeSats, which can be seen as a spacial kind of embedded device.

Firmware can be secure, even without advanced memory protection mechanisms if it is programmed correctly. In the case of the bugs in the OPS-SAT, a simple if-statement that checks the input length against the buffer size would be enough to prevent a buffer overflow. Nevertheless, the firmware developers failed to implement such a check for the three buffers that can be overflown. A possible reason for this is that software developers often don't focus on security when they implement software [BFCB21]. Past research also discovered that software developers often expect other parties to ensure the security of a software project as a whole and therefore do not consider security issues during their work [XLC11]. Further, developers will not perform additional steps to make a product secure if there is no regulatory need for it, as their primary objective is that the software works as intended from a functional point of view [XLC11]. Combined with the missing regulations on the security of space systems and the assumption that safety concerns are seen as more important than security, this may explain why the vulnerabilities in the OPS-SAT were not detected during development [Fal18].

Interestingly, we did not identify any vulnerability in the telecommand handlers themself. Instead, we noticed that they use special functions to access data in MAL message objects. These functions perform size checks before the memory operation and therefore are more resistant against manipulated input. The developers of these functions seem to be aware of the risks of non-validated input data and hence applied the necessary validation functions. Because we do not have any insight into the development process of the OPS-SAT firmware, we can not determine if these are the same developers that implemented the vulnerable functions. However, if the same level of quality and security assurance would be used during the whole development process, the security of the final product would be improved.

Considering the above aspects, the development of secure CubeSat firmware should be improved on different layers.

First, the technical aspects of CubeSat security should be considered more strongly. For example, modern concepts for memory protection can be used in embedded systems and therefore could be used in CubeSats. Additionally, technical security should get a higher focus during the development phase of a CubeSat. Besides the safety and functionality tests, there could be tests that aim to verify the security of CubeSat firmware. For example, there are concepts for the detection of buffer overflows by performing static analysis of the program's source code [LE01].

Second, apart from technical aspects, the security of CubeSats could be improved if certain conceptual aspects in the development process are optimized. One example is the security awareness of the developers that needs to be enhanced [TTCL18].

Finally, the development of secure CubeSat firmware should be improved by coordinating all security-relevant tasks centrally. The project management should include a role that is responsible to ensure that security aspects are taken into consideration by all project members and during any development phase. As was shown for the OPS-SAT, security aspects were considered in some cases, but not in others. A central coordination of related tasks could reduce the risk that developers expect other project members to consider the security and vice versa [Fal18].

7.3 Hurdles and Issues

During our work, we encountered various issues and hurdles. Some of them are related to the QEMU implementation, while others are caused by the characteristics of the OPS-ST firmware. In this section, we will discuss the 5 most important hurdles that we came across.

7.3.1 Documentation

The QEMU project is a huge open source project that many people contributed to. But, even with the wiki on the project website, there is a significant lack of documentation and manuals for new developers. Also, there is not much discussion on public forums about errors or issues that developers come across. Even frequent errors that appeared during the implementation were not discussed between developers online. Searching for issues often did end without any meaningful results. Even the QEMU project itself states that there is no design overview and that descriptions are often outdated [QEMc]. To gain an understanding of QEMU, developers are expected to read the code of QEMU by themself. Although the code is decently commented on for most parts, going through such a large code base without a general overview is a time consuming task.

The lack of documentation was a huge hurdle for us as new QEMU developers. To understand the inner workings of QEMU, we often needed to look at the implementations of other architectures. That sometimes led to more hurdles, as different architectures used different ways to implement the same functionality and not all approaches were compatible with our existing implementation. We also needed to read QEMUs source code which sometimes is not easy to understand.

In general, it would be advantageous for the QEMU project to provide documentation that is more helpful for new developers and that explains how the inner workings are connected and how a new architecture can be implemented. Therefore, the descriptions in chapter 4 may be beneficial for new developers.

7.3.2 Missing hardware input or output

The OPS-SAT firmware uses many functions that interact with the satellites hardware components. If those components are not emulated, QEMU provides the value *null* as a result if the firmware tries to read data from them. Any output send to the components is ignored and creates no reaction by the emulator.

For the most part, the value *null* is accepted as a valid result, although it often indicates that a hardware device is not responding or is not ready. In this case, the firmware sometimes reports an error via the UART output. As the error was expected, the emulation continues, even if some code areas will not be executed. Thus, many functions of the firmware were not covered by the fuzzing.

An issue also occurs if the firmware waits for a certain input. For example, there is a while loop at address *0xd00c1064* that reads the Cycle Counter register and only breaks, if a specific cycle count was reached.

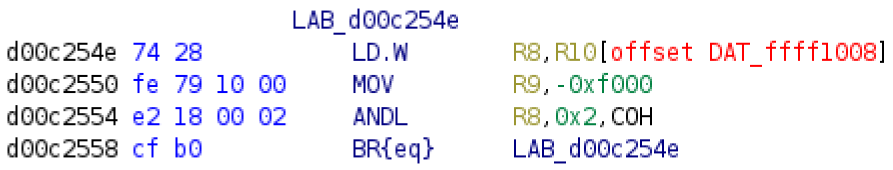
Because the cycle counter register is not updated in our implementation, the loop will never exit and the emulation did not run any further. To circumvent this behavior, we implemented a workaround that simply skips the branch instruction at this specific address:

```
1  if(ctx->base.pc_next == 0xd00c106a) {
2    ctx->base.pc_next += 2;
3    ctx->base.is_jmp = DISAS_CHAIN;
4    return true;
5  }
```

Listing 7.1: A workaround in the branch instruciton.

The workaround is placed before the actual translation code of the BR instruction. In line 1, we check if the current program counter address is `0xd00x106a`, the address of the branch instruction at the end of the loop. If this is the case, we tell QEMU to increase the program counter by 2 in line 2, as the BR instruction is 16 bits long. In line 3 we tell the TCG that the basic block has ended and that the next basic block should be appended without a jump. In line 4, the workaround returns true to skip the actual code of the instruction and prevent a branch back to the start of the loop. With this workaround, the emulation will always jump to the next instruction and the value in the Cycle Counter register is ignored. In total, there were 4 loops that waited for a certain Cycle Counter value that needed this workaround. Because a timing device was added to the implementation later, this workaround could be removed in the future if the Cycle Counter register is updated by the timer.

Another example of a hardware-caused endless loop is found at address `0xd00c254e`. In this do-while loop, the firmware reads input from the watchdog timer until the value `2` is received. As there is no watchdog timer implemented, this will be never the case and the emulation will not leave the loop. Again, the above workaround was used to skip the BR instruction at address `0xd00c2558`.



```
                             LAB_d00c254e
d00c254e 74 28                  LD.W        R8,R10[offset DAT_ffff1008]
d00c2550 fe 79 10 00            MOV         R9,-0xf000
d00c2554 e2 18 00 02            ANDL        R8,0x2,COH
d00c2558 cf b0                  BR{eq}      LAB_d00c254e
```

Figure 7.1: A do-while loop at address `0xd00c254e` that reads data from the watchdog timer.

An alternative solution would be the implementation of a dummy watchdog timer device. But, as the workaround is done much quicker and has less potential for implementation errors, the workaround was preferred.

For some hardware components, a virtual device was added to the emulator. For example, we implemented a dummy FL512S Flash device and a UART interface. This was done to better understand what inputs and outputs the firmware is receiving or sending. The virtual UART device also provides us with the UART output that could be used to further understand the program flow of the firmware.

For example, some functions that print data to a log file do not work, because the file descriptor is not created by the firmware, as the virtual FL512S device does not contain a file system. If the print functions were called, they checked the file descriptor and noticed that it was not initiated. The resulting call to an error handling function lea to a CPU reset. At one point, this prevented the firmware from finishing the setup functions and starting the tasks. Fortunately, the error handling functions print out an error message to the UART interface. This way, the source of the error could be observed in real-time. The error message also contained the location of the error indicating-function in the source code of FreeRTOS. With this information, it was easy to identify the MCALL instruction that needs to be skipped to prevent the faulty code segment.

7.3.3 Identifying vulnerabilities

When fuzzing an application with AFL, AFL tries to generate input that results in a crash of the application. If a crash occurs, AFL receives the exit code of the application and reports that a crash was found. However, this is not the case for our QEMU implementation, as we emulate the whole operating system. If an error occurs during the firmware emulation, an exception may be raised *inside* the emulated firmware. But, this exception does not result in a crash of QEMU that AFL could recognize. Instead, the emulated firmware performs a CPU reset and restarts. To circumvent this, we added patches to QEMU that report an exit code to AFL without stopping the emulation if one of the emulated error-handling functions is executed.

As we noticed during the investigation of the memcopy-bug, not every error results in an instant exception inside the firmware. We observed that the firmware was not running properly, but no crash was reported to AFL and no error-handling routines were executed inside the firmware. It is possible that the memcopy-bug results in undefined behavior of the firmware that does not trigger any noticeable crash. Because of this, *classic* crashes are not enough to determine if an input triggered a vulnerability. Instead, the fuzzing needs to be monitored manually for unexpected behavior, such as very low execution speed or unintended endless loops. But, even then a bug may stay unnoticed if it does not provoke behavior that can be noticed with AFLs output.

Another issue is that a vulnerability may provoke an error that only effects the firmware after some time. If our fuzzing loop would have restarted the emulation after the `TCTA_Cycle` was completed, the memcopy-bug would not have been noticed.

With these considerations, we conclude that fuzzing satellite firmware with full system emulation may not be able to detect issues that could be detected when fuzzing a regular application. The chances to detect firmware bugs could be increased if checks are added to the emulator that try to detect unexpected memory manipulation. This approach was shown to be effective by other research in the past [MSK$^+$18b]. An example for such a functionality is to observe memory areas that were used by a PUSHM instruction. PUSHM writes multiple registers to the stack and is often used at the very beginning of functions to save the caller register values. Usually, these values are restored by a POPM instruction at the end of a function just before the function returns. If the contents of such memory areas are modified at any other point, this may indicates that a stack buffer overflow occurred.

7.3.4 Large basic blocks

QEMU can only work on basic blocks up to a certain length. The OPSSAT firmware has at least one function that exceeds this limit. The *CKSM_Init* function performs 256 load/store operations that consist of 2 instructions. As there is no branch between these instructions, the resulting basic block is too long for QEMU and the emulation ends with an *icount_enabled* exception that indicated that the maximum number of instructions was reached.

To circumvent this issue, we injected a patch into the STB (store byte) instruction that tells QEMU to perform a branch after every store operation in *CKSM_Init*. Because not every part of the firmware was executed so far, it is possible that there is another function where this issue occurs.

While this specific issue was easy to fix, it illustrates that workarounds are needed for various situations when fuzzing satellite firmware.

7.3.5 Slow fuzzing speed

During the fuzzing phase, we frequently encountered AFL instances with a low performance. In some cases, we were able to determine the reason for the slow speed or at least identify the responsible code segment. However, in multiple instances we were not able to do so. Because the impact on our results was not significant, we did not further address this issue. Nevertheless, if another area of the firmware is selected for fuzzing, meaningful fuzzing is probably not possible. For example, fuzzing of the `task_adcs_server` function resulted in one execution every 2 or 3 seconds. This performance prevents any fuzzing results in a reasonable time.

To address this issue, multiple approaches can be used. For example, the targeted code segment could be executed on its own, without a full emulation of the firmware. Our experiment in section 6.2.1 showed that a high performance can be achieved this way. However, this approach is only effective if the targeted code segment contains a bug that results in an instant crash inside the emulated area. Usually, it is unknown if this is the case beforehand. Hence, it will be more effective to modify the functions that are responsible for the slow execution speed. These are mainly function calls with timeouts. Such functions can be skipped by applying side patches to the emulator or by modifying the firmware image.

7.4 Lessons learned

In the previous section, we explained the different hurdles and issues that we came across during this work. We also made different smaller experiences that will be helpful for future fuzzing-based security assessments of satellite firmware. We will use this knowledge to name the 3 most important lessons that would have improved our work if we would have known them beforehand.

1. Test-driven development

One of the most important lessons that we learned during this work is the proper implementation approach when adding a new architecture to QEMU. Because QEMU's translation functions add another layer of complexity, implementation errors are likely to occur, as we explained before. We learned that a test-driven development approach helps to prevent unnoticed errors and therefore advise other researchers to use such an approach from the start, when they plan to implement a new architecture into QEMU. The use of the testing framework that was developed for our work is an example for such a test driven approach.

2. Advanced crash detection

As explained earlier, we noticed unexpected behavior of the firmware during the fuzzing phase. In some cases, the reason for this was a buffer overflow that overwrote information that is needed for the firmware to operate. Because FreeRTOS does not provide security mechanisms that can detect this situation, error handling routines were only executed if another more serious error occurred as a consequence. A possible solution for this issue that uses advanced crash detection mechanisms is discussed in section 7.3.3. Our fuzzing would have been significantly more effective if such mechanisms were used during our work.

3. Useful hardware emulation

We implemented multiple hardware devices during this thesis and planned to emulate the full functionality of some of them. However, we noticed that this was a complex task and in some instances we were missing necessary information. For example, we do not have an image of the FRAM content. Hence, we only used most of the virtual devices to observe the input that the firmware sends to them and to respond with a value that lets the firmware continue it's execution. When fuzzing satellite firmware, it will be helpful to have access to all relevant memory images to provide a better emulation. Otherwise, it is more effective to only implement a basic emulation and leave out parts that will not be used because of missing images or high complexity. This way, no time is lost with the implementation of functionality that will not be useful for the security assessment. It will be more time efficient to modify the firmware or the emulator to skip code segments that stop the execution until a certain value is received from peripheral hardware.

On the other hand, the emulation of some hardware interfaces was helpful to understand the functionality of the firmware. For example, the UART interface that we implemented allowed us to read logging output from the firmware. That output helped us multiple times to understand why the firmware we behaving in a certain way, for example, when a crash was reported. Hence, we recommend that hardware interfaces that are used to print or store log information should be emulated early.

8 Related Work

8.1 Related work

Gregory Falco analyzed why the security of space systems is not researched as well as the security of regular computer systems [Fal18]. He found out that space systems are often ignored due to multiple reasons when the security of infrastructures is discussed. These results were part of the motivation for this thesis. A related assessment was done by Pavur and Martinovic [PM20]. They analyzed various satellite security incidents and concluded that criminals are likely able to perform command injection in satellites. These insights were useful, when we decided which parts of the firmware should be in focus during the fuzzing.

Falco, Viswanathan and Santangelo also did an analysis to develop an Attack Tree for CubeSats, in which they determined different approaches to attack CubeSats [FVS21]. This attack tree provides a detailed overview of potential attack vectors and demonstrates the broad range of possible malicious activities that motivated us to do this thesis. Willis et. al. evaluated what attackers could do with a satellite, once they gained access to it [WMMG17]. For example, the readings of positioning sensors could be spoofed to force the satellite to perform unwanted course corrections. The considerations from this research could be tested with the emulator and the vulnerabilities that we discovered.

Muench et. al. researched the fuzzing of embedded devices [MSK$^+$18b]. One important aspect of their research is that embedded systems may not produce noticeable errors if a vulnerability was triggered by fuzzing. This was an issue that we also came across during our research. We were able to identify fuzzing indicated error without advanced detection techniques, however our fuzzing implementation could be improved, if such techniques are applied.

9 Conclusion

In the conclusion, we summarize the results of this thesis. We first describe our work and explain how we achieved our research goals. Then we discuss further research questions that can be based on our work.

9.1 Summary

The goal of this thesis was to provide the following three contributions:

- Implementation of an AVR32 emulator
- Fuzzing-based security assessment of the OPS-SAT firmware
- Collection of hurdles and issues, that come up when fuzzing satellite firmware

We were able to achieve all three of these goals.

The first part of our work was the implementation of an AVR32 emulator that is based on the QEMU project. The emulator allows us to execute the OPS-SAT firmware, which is using the AVR32 architecture, on a CPU of another architecture. This process is called *rehosting* and increases the execution speed of the firmware, as CPUs in desktop computers are significantly more powerful than the CPU of the OPS-SAT. *Rehosting* will therefore improve the performance of the fuzzing process.

To add a new architecture to QEMU, we needed to develop translation functions that reproduce the operation of each relevant AVR32 instruction. The translation functions create an Intermediate Representation (IR) of each instruction. The IR contains code that performs operations on emulated variables, like the emulated CPU registers, and the emulated memory. The IR is later translated into instructions of the host architecture by QEMU.

QEMU needs to be able to determine which instruction should be executed when a sequence of bytes is loaded from the firmware image. Hence, we needed to define *patterns* that represent the different instructions of the AVR32 architecture. Patterns consist of fixed bits that identify the instruction and variable bits that represent *fields*. Fields contain values that are variable, for example, the number of a register that is used by an operation. The fields are set during the compilation of a program and are passed to the translation functions.

The implementation of the translation functions is a process that is prone to errors because the IR adds another layer of complexity. The values of the virtual variables are not known during the translation of an instruction, as QEMU first translates a block of instructions and then executes it. Therefore, values can not be accessed for debugging and implementation errors are easy to miss. To circumvent this issue, we developed a testing framework that allows us to verify the emulation of the AVR32 instructions. We defined 470 test cases that test the emulated instructions and compare the actual results of each instruction with the expected results. By using this framework, we identified and corrected various implementation errors that were not detected beforehand.

The second part of this thesis was a fuzzing-based security assessment of the OPS-SAT firmware. The first task at hand was to identify a suitable function that can be tested in the fuzzing loop. We evaluated different functions and selected one of them for the assessment. The selected function is responsible to handle *telecommands* that are used to control the satellite. Next, we implemented a connection between our emulator and the AFL fuzzing tool. The connection allows the fuzzing tool to send input data into the emulated memory of the firmware and receive coverage and status information from QEMU. We also created multiple input seeds for the fuzzing tool that should trigger the execution of all available telecommands.

We performed multiple fuzzing runs that resulted in the execution of all telecommand handlers and a total coverage of 85%. The fuzzing tool reported multiple crashes and we also noticed that the firmware execution behaved in unexpected ways for some fuzzing instances. We also tested a targeted approach that focuses on a single command handler and were able to improve the initial coverage for some of the handlers.

After investigating the crashes and the unexpected behavior, we identified an insecure memory operation that caused a buffer overflow. The buffer overflow can be used to overwrite a code pointer in the firmware's memory. We were able to develop an exploit that allows us to overwrite this pointer with a value of our choice and therefore take control of the firmware. However, this exploit was only tested in a laboratory setting and might not work on the real OPS-SAT. Because we did not implement a full emulation of peripheral devices, related functions of the firmware were skipped during the fuzzing and it is possible that the exploit does not work in a real-world scenario.

We were aware of another vulnerability in the OPS-SAT firmware that was identified before this thesis. We decided to test if our fuzzing implementation can detect said vulnerability and therefore modified our fuzzing integration accordingly. That test was successful, as the fuzzing tool triggered a crash of the firmware. However, the vulnerability in question was not the cause of the crash. While the vulnerability causes a stack overflow, which overwrites values on the stack, it does not overwrite a code pointer. Instead, another insecure memory operation also caused a stack overflow. We developed an exploit for this second vulnerability and used our emulator to test if the exploit can be used to gain control of the firmware. That test was also successful, as we were able to overwrite a code pointer that was loaded into the program counter. Again, this exploit was only tested under laboratory conditions and needs to be verified with full hardware emulation.

We showed that the OPS-SAT contains a least two vulnerabilities that can be exploited in a laboratory setting. We also showed that fuzzing can be used to identify vulnerabilities like these in satellite firmware.

The third contribution of this thesis is a collection of hurdles and issues that researchers need to handle when fuzzing satellite firmware. To this end, we summarized multiple difficulties that we came across during our work.

The first hurdle is the lack of documentation and tutorials for developers that want to add a new architecture to QEMU. We needed to read the code of other architecture implementations and the QEMU core system, to gain an understanding how to implement certain aspects of the emulation.

The second hurdle is the dependence on hardware devices. The firmware has many interactions with peripheral hardware that change how the firmware behaves. As we did not emulate most hardware interfaces, we implemented various workarounds that skip parts of the firmware that interact with hardware.

Another issue was the detection of errors that should be counted as crashes in the fuzzing tool. Because we emulated a full operating system, an error did not result in an exit code. Instead, the firmware executes it's internal error handling routines and resets the virtual CPU. We needed to manually define points in the firmware that would trigger the report of an error to AFL, if they were executed. However, in some cases, the found vulnerability did not trigger a reset of the CPU but other errors that were not automatically detected.

A further hurdle is the low execution speed of certain parts of the firmware. Because some functions perform timed operations, the fuzzing speed in some code segments was significantly slower than in other parts.

We learned multiple lessons from this experience. For example, we will use a test-driven development approach in the future when new instructions are added to QEMU. We also would use advanced crash detection mechanisms during the fuzzing. Finally, we would reduce the complexity of hardware emulation by focusing on workarounds that skip parts of the firmware that depend on hardware interactions. The only exception are hardware interfaces that are used to print log information, like UART. Such interfaces should be emulated early on, as they can be used to better understand the behavior of the firmware.

9.2 Future work

This thesis showed how fuzzing can be used to test the security of satellite firmware. Besides the exploited vulnerabilities, we noticed that the firmware starts to behave unexpectedly if specific input seeds are used for the fuzzing. This behavior should be further investigated to determine if it is the result of another vulnerability. To investigate this potential vulnerability in an effective manner, advanced fault detection techniques, like Heap Object Tracking, should be implemented into the emulator. Such techniques showed a significantly improvement in the detection of vulnerabilities in embedded systems [MSK+18b].

To further assess the security of the OPS-SAT more hardware devices could be implemented into the emulator. This way, many functions that depend on hardware-related values could be tested more effectively. It would be also useful to integrate a copy of the file system that is stored on the flash memory, because there are functions that write data to or read data from this memory.

The developed AVR32 extension of QEMU can be optimized to make it capable to emulate the full AVR32 instruction set. The virtual QEMU CPU could also be updated to support AVR32b CPUs. The resulting emulator can be used as a base for other research projects that aim to assess the security of AVR32 based satellite firmware or AVR32 devices in general. A first version of the QEMU extension with full support for the AVR32 instruction set will be provided to the QEMU project after this thesis.

Finally, the results of the above improvements can be used to do further fuzzing-based security assessments of the OPS-SAT. The other functions that were listed in section 5.1 could be assessed for vulnerabilities. Our approach could also be used to evaluate the security of other satellites.

A Acronyms

CAN	Controller Area Network
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
ELF	Executable and Linkable Format
ESA	European Space Agency
FRAM	Ferroelectric Random Access Memory
GPS	Global Positioning System
IR	Intermediate Representation
LEO	Low Earth Orbit
LR	Link Register
MAL	Message Abstraction Layer
PC	Program Counter Register
RTOS	Real Time Operating System

SEPP Satellite Experimental Processing Platform

SOC System-On-Chip

SDRAM Synchronous Dynamic Random Access Memory

SPI Serial Peripheral Interface

SRAM Static random-access memory

TCG Tiny Code Generator

UART Universal Asynchronous Receiver Transmitter

List of Figures

List of Tables

List of Listings

Bibliography

[Atma] Atmel Corporation. Acr32uc technicall reference manual.

[Atmb] Atmel Corporation. Avr32 architecture document.

[BFCB21] Larissa Braz, Enrico Fregnan, Guel Calikli, and Alberto Bacchelli. Why don't developers detect improper input validation? '; drop table papers; –. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 499–511, 2021.

[Bun22] Bundesamt für Sicherheit in der Informationstechnik. Die lageder i t-sicherheit in deutschland 2022, 2022.

[Chr] Johan De Claville Christiansen. The cubesat space protocol. `https://github.com/libcsp/libcsp`, 31.10.2022.

[CMT20] Giacomo Curzi, Dario Modenini, and Paolo Tortora. Large constellations of small satellites: A survey of near future challenges and missions. *Aerospace*, 7(9), 2020.

[Eura] European Space Agency. Ops-sat. `https://www.esa.int/Enabling_Support/Operations/OPS-SAT`, 31.10.2022.

[Eurb] European Space Agency. Ops-sat: your flying laboratory. `https://www.esa.int/Enabling_Support/Operations/OPS-SAT_your_flying_laboratory`, 13.11.2022.

[Eva16] David Evans. Ops-sat: Operational concept for esa's first mission dedicated to operational technology. 05 2016.

[Fal18] Gregory Falco. The vacuum of space cybersecurity. *2018 AIAA SPACE and Astronautics Forum and Exposition*, 2018.

[FMEH20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++ : Combining incremental steps of fuzzing research. In *WOOT @ USENIX Security Symposium*, 2020.

[Frea] FreeRTOS.org. Context switching. `https://www.freertos.org/implementation/a00006.html`, 09.11.2022.

[Freb] FreeRTOS.org. The freertos kernel. `https://www.freertos.org/RTOS.html`, 09.11.2022.

[Frec] FreeRTOS.org. Multitasking. `https://www.freertos.org/implementation/a00004.html`, 09.11.2022.

[Fred] FreeRTOS.org. Real time scheduling real time scheduling. `https://www.freertos.org/implementation/a00008.html`, 09.11.2022.

[Free] FreeRTOS.org. Scheduling. `https://www.freertos.org/implementation/a00005.html`, 09.11.2022.

[Fref] FreeRTOS.org. Stack usage and stack overflow checking. `https://www.freertos.org/Stacks-and-stack-overflow-checking.html`, 09.11.2022.

[Fri13] Jason Fritz. Satellite hacking: A guide for the perplexed. 2013.

[fSDS15] Consultative Committee for Space Data System. Mission operations—mal space packet transport binding and binary encoding, 2015. `https://public.ccsds.org/Pubs/524x1b1.pdf`, Accessed at 21.11.2021.

[fSDS20] Consultative Committee for Space Data System. Space packet protocol, 2020. `https://public.ccsds.org/Pubs/133x0b2e1.pdf`, Accessed at 21.11.2021.

[FVS21] Gregory Falco, Arun Viswanathan, and Andrew Santangelo. Cubesat security attack tree analysis. 2021.

[Gom21] GomSpace. A3200 datasheet, 2021. `https://gomspace.com/shop/subsystems/command-and-data-handling/platform-software.aspx`, Accessed at 21.11.2021.

[Jon18] Harry W. Jones. The recent large reduction in space launch cost. *International Conference on Environmental Systems*, 48, 2018.

[Kul20] Erik Kulu. Nanosatellite launch forecasts - track record and latest prediction. *35th Annual Small Satellite Conference*, 2020.

[Kul21] Erik Kulu. Satellite constellations - 2021 industry survey and trends. *35th Annual Small Satellite Conference*, 2021.

[LC03] Kyung-Suk Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exper.*, 33(5):423–460, apr 2003.

[LE01] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, USA, 2001. USENIX Association.

[MH] Reinhard Zeif Maximilian Henkel, Patrick Romano. Ops-sat phase b2/c/d/e1, experimenter icd.

[MIT] MITRE Corporation. Cwe top 25 most dangerous software errors. `https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html`, 2022. Viewed at 25.10.2022.

[MSK+18a] Marius Muench, Jan Stijohann, Frank Kargl, Aurelien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. 01 2018.

[MSK+18b] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[NE19] Cristóbal Nieto and Reza Emami. Cubesat mission: From design to operation. *Applied Sciences*, 9:3110, 08 2019.

[Nib21] Nibedita Mohanta. How many satellites are orbiting the earth in 2021?, 28 2021. `https://news.viasat.com/blog/corporate/ka-sat-network-cyber-attack-overview`, 31.10.2022.

[Oeh05] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security Privacy*, 3(2):58–62, 2005.

[PM20] James Pavur and Ivan Martinovic. Sok: Building a launchpad for impactful satellite cyber-security research, 2020.

[pro] AFLplusplus project. Aflplusplus. `https://github.com/AFLplusplus/AFLplusplus`, 31.10.2022.

[PW08] Sri Parameswaran and Tilman Wolf. Embedded systems security—an overview. *Design Autom. for Emb. Sys.*, 12:173–183, 09 2008.

[QEMa] QEMU Project. Decodetree specification. `https://www.qemu.org/docs/master/devel/decodetree.html`, 20.09.2022.

[QEMb] QEMU Project. Documentation/gettingstarteddevelopers. `https://www.qemu.org/docs/master/about/index.html`, 31.10.2022.

[QEMc] QEMU Project. Documentation/gettingstarteddevelopers. `https://wiki.qemu.org/Documentation/GettingStartedDevelopers`, 20.09.2022.

[QEMd] QEMU Project. Documentation/tcg/frontend-ops. `https://wiki.qemu.org/Documentation/TCG/frontend-ops`, 20.09.2022.

[QEMe] QEMU Project. Qemu tcg readme. `https://www.qemu.org/docs/master/devel/decodetree.html`, 20.09.2022.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (state of) the art of war: Offensive techniques in binary analysis. *2016 IEEE Symposium on Security and Privacy*, 2016.

[TM18] Nektarios Georgios Tsoutsos and Michail Maniatakos. Anatomy of memory corruption attacks and mitigations in embedded systems. *IEEE Embedded Systems Letters*, 10(3):95–98, 2018.

[TTCL18] Tyler W. Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. Security during application development: An application security expert perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, page 1–12, New York, NY, USA, 2018. Association for Computing Machinery.

[tur] turbosree. Qemu-avr32. `https://github.com/turbosree/QEMU-AVR32`, 31.10.2022.

[Via22] Viasat. Ka-sat network cyber attack overview, 03 2022. `https://news.viasat.com/blog/corporate/ka-sat-network-cyber-attack-overview`, 31.10.2022.

[WMMG17] John M. Willis, Robert F. Mills, Logan O. Mailloux, and Scott R. Graham. Considerations for secure and resilient satellite architectures. In *2017 International Conference on Cyber Conflict (CyCon U.S.)*, pages 16–22, 2017.

[XLC11] Jing Xie, Heather Richter Lipford, and Bill Chu. Why do programmers make security errors? In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 161–164, 2011.

[ZDY+19] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1099–1114, 2019.