Bachelor Thesis
Computing Science

Midnight Blue & Radboud University Nijmegen

Security analysis of a TETRA Base Station air interface implementation

Company Supervisors:
Carlo Meijer - Midnight Blue
Wouter Bokslag - Midnight Blue

Author:
Jonathan Jagt

University Supervisor:
Erik Poll

Second reader:
David Rupprecht

January 19, 2023

Abstract

Terrestrial Trunked Radio (TETRA) is a radio system powering the communication for emergency services, public safety networks, train radios, and other critical infrastructure. It is expected that this system ensures confidentiality, integrity and availability. As a public researcher, analyzing the security of TETRA Base Stations is not straightforward since the firmware binaries are closed source and obtaining hardware is difficult. In this thesis, we test the security of the Air Interface of a TETRA Base Station by fuzzing the packet parsing code and performing a brief security analysis by manually reverse engineering the firmware binaries.

The fuzzing is done using custom tooling to modify the running firmware to make a memory dump at the moment of parsing a packet. This memory dump contains all the code of the firmware and operating system, all data used by the firmware and operating system and the contents of the CPU registers. We wrote a program to load the memory dump into the Unicorn emulator so we can emulate the execution of the packet parsing. Using this technique, we can connect a fuzzer to the Unicorn emulator to fuzz the packet parsing code of the Base Station. We have not found any issues with regards to memory corruption in the packet parsing code we fuzzed. We have concluded that it is unlikely that this code contains any issues related to memory corruption, since the packet parsing code appears to be automatically generated code based on the TETRA specification.

During manual reverse engineering the firmware, we have found that with physical access to the Base Station it is possible to authenticate with a backdoor password to obtain high privileged shell access. Moreover, by entering the `.crashdumptest` command, it is possible to crash the firmware and enter an interactive debugger session. With the interactive debugger session, it is possible to extract encryption keys from memory. This is against the design of the Base Station: it should not be possible to extract encryption keys from the Base Station this easily, because it compromises the confidentiality of the TETRA network.

Contents

Glossary

Air Interface The wireless communication link between two devices.

Authentication Centre A separate entity in the TETRA network which manages the unique pre-shared Authentication Key between every Mobile Station and the Authentication Centre. The Base Station is trusted by the Authentication Centre and may carry out the encryption and authentication of a Mobile Station.

BR Base Radio: The Base Radio is a firmware binary of the Base Station which is responsible for the low level functionalities of the TETRA specification.

BS Base Station.

CC Colour Code.

CC Call Control.

CCK Common Cipher Key: Cipher key that is generated by the infrastructure to protect group addressed signalling and traffic [9].

CK Cipher Key: Value that is used to determine the transformation of plain text to cipher text in a cryptographic algorithm [9].

CLI Command-line Interface.

CMCE Circuit Mode Control Entity.

CN Carrier Number.

Cydia Substrate A code modification platform used to replace and/or modify the behavior of existing functions in the BS firmware (see Section 2.3).

DCK Derived Cipher Key: DCK is generated during authentication for use in protection of individually addressed signalling and traffic [9].

Downlink The radio path from the Base Station to the Mobile Station (sometimes called the outbound path) [7].

ECK Encryption Cipher Key: Cipher key that is used as input to the encryption algorithm. The Encryption Cipher Key is further explained in Section 2.1.1.

Fuzzing Fuzzing is an automated software testing technique used by security researchers to find bugs in all kinds of software. Fuzzing is further explained in Section 2.5.

GCK Group Cipher Key: Cipher key known by the infrastructure and MS to protect group addressed signalling and traffic. Not used directly at the air interface but modified by CCK or SCK to give a Modified Group Cipher Key (MGCK) [9].

IPC Inter-Process Communication.

IV Initialization Value: Sequence of symbols that initializes the KSG inside the encryption unit [9].

K Authentication Key: Primary secret, the knowledge of which has to be demonstrated for authentication [9]. The Authentication Key is a pre-shared secret between the Mobile Station and the Authentication Centre. For example, this secret can be pre-shared by a SIM card or this secret is embedded in the Mobile Station.

KEK Key Encryption Key: A Motorola specific key used to encrypt secrets for long-time storage purposes on the Base Station.

Ki Infrastructure Key: The encryption key used to encrypt the Key Encryption Key. The Infrastructure Key is also used for different purposes related to infrastructure management.

KSG Key Stream Generator: Cryptographic algorithm which produces a stream of binary digits which can be used for encipherment and decipherment. The initial state of the KSG is determined by the initialization value [9].

KSO Session Key for OTAR: Derived from a user's authentication key and a random seed for OTAR. KSO is used to protect the transfer of the Static Cipher Key [9].

KSS Key Stream Segment: Key stream of arbitrary length [9].

KVL Key Variable Loader: A hardware device used to update the Infrastructure Key stored in the MBTS Base Station.

LA Location Area id: Unique identifier within the infrastructure of a location area [9].

MBTS The name of the model of the Base Station [12] used in this research.

MGCK Modified Group Cipher Key: See Group Cipher Key.

MLE Mobile Link Entity.

MM Mobility Management.

MS Mobile Station: The device that connects to other Mobile Stations through the Base Station. For example, this can be a device such as a walkie-talkie that uses TETRA as the communication channel.

MTS2 The name of the model of the Base Station [15] used in the research of Müller et al. at the TU Darmstadt [16].

OTAR Over The Air Re-keying: Method by which the Authentication Centre and Base Station can transfer secret keys securely to Mobile Stations.

RSO Random Seed for Over The Air Re-keying.

RTOS Real-Time Operating System.

SC Site Controller: The Site Controller is a firmware binary of the Base Station which is responsible for the high level functionalities of the TETRA specification.

SCCK Sealed Common Cipher Key: Encrypted Common Cipher Key.

SCK Static Cipher Key: Predetermined cipher key that may be used to provide confidentiality in class 2 networks with a corresponding algorithm [9].

SCK-VN Static Cipher Key Version Number.

SDL Specification and Description Language.

SDS Short Data Service.

SKEK Sealed Key Encryption Key: encrypted Key Encryption Key.

SNDCP Subnetwork Dependent Convergence Protocol.

SSCK Sealed Static Cipher Key: Encrypted Static Cipher Key.

TETRA Terrestrial Trunked Radio.

Uplink The radio path from the Mobile Station to the Base Station (sometimes called the inbound path) [7].

When this PDF is viewed digitally, it is possible to click in the text on words described in the glossary to jump to the explanation in the glossary.

1 Introduction

First published in 1995, TETRA is a multi-function mobile radio standard used for critical infrastructure in more than 120 countries [27]. Its purpose is to provide secure and always available mobile-to-mobile voice and data communication possibilities [7]. The network can be used with a Mobile Station (MS) which can serve as a phone to call and send text messages to/from. Since the TETRA network is completely independent from other mobile networks, such as GSM, a cellular outage of a service provider will not affect TETRA. This makes the technology suitable for critical infrastructure and everyday services. Therefore, it is also important to verify that TETRA is indeed secure and always available.

In this thesis, we perform two security analyses:

1. We fuzz the code responsible for parsing incoming packets from the Air Interface.
2. We perform a manual security analysis by reverse engineering the Site Controller firmware.

For the former security analysis, we test the code responsible for parsing incoming packets from the Air Interface of one of the TETRA Base Stations. This Base Station runs on a PowerPC architecture, which is commonly used for Real-Time Operating Systems (RTOSs). We are going to test the air interface by fuzzing the packet parsing code in the Site Controller firmware. Since the Base Station runs on a PowerPC architecture and is powered by the pSOSystem RTOS, we cannot execute the Site Controller firmware on an external device without emulation. Moreover, we want to fuzz the packet parsing code with coverage guidance, so the fuzzer can find issues related to memory corruption more effectively. Therefore, we use emulation to run the packet parsing code on an external device by performing a memory dump at the moment of packet parsing in the Site Controller firmware. Then we connect a fuzzer to the emulator to test the packet parsing code (see also Section 4.1.2).

In short, the steps of the fuzzing research are:

1. We find the code in the firmware responsible for parsing the packets from the Air Interface (Section 5.1).
2. We write a custom memory dump module which can be loaded by our instrumentation (Section 5.2).
3. We hijack the control flow of the packet parsing in the event of an incoming packet and dump all the memory of the Site Controller firmware (Section 5.3).
4. We load the memory dump in an emulator to run at the moment of packet parsing (Section 5.4).
5. We connect a fuzzer to the emulator so we can fuzz the parser (Section 5.5).

For the latter security analysis, we reverse engineer parts of the Site Controller firmware to assess the security of one of the TETRA Base Stations when an adversary has physical access to the serial port of the Base Station, but does not modify the Site Controller firmware and does not perform any hardware attacks. We first test the security of the authentication mechanism when the adversary connects to the Site Controller firmware. We identified an undocumented backdoor password that allows for high privileged shell access. The Base Station is designed to keep the encryption keys secret, but due to flaws in the design, it is possible to extract the encryption keys using the backdoor password. With the encryption keys, the adversary can decrypt over-the-air traffic and thus compromise the confidentiality and integrity of the TETRA network.

In Section 2.1 we discuss some background information about how the TETRA protocol works. In Section 2.2 we discuss the Base Station setup. Our instrumentation, constituting of Tinker, Fiddle and Cydia Substrate, are explained in Section 2.3. Furthermore, in Section 2.4 we explain some theory about watchdogs and explain how they impact our research. We also explain some theory about fuzzing and emulation methods in Section 2.6. The impact of compiler optimization on compiled code, and hence the decompilation output when reverse engineering the compiled code is discussed in Section 2.7. In Section 3 we explore the research that has already been done on TETRA Base Station security, and describe the differences with the research of Section 5. Section 4 will explain the reasoning behind some of the decisions made prior to the research of Section 5. Section 5 will go in depth about how we performed our fuzzing research and what decisions we have made during our research. Section 6 will go in depth about how we have performed the manual firmware analysis of the Site Controller firmware and what we have found. What can be improved in this research, is discussed in the future work in Section 7. Finally, Section 8 will discuss all the findings and conclusions, and will reflect on decisions we made during our research.

This research was done as an internship at Midnight Blue. Midnight Blue provided the tools (see Section 2.3), hardware and firmware required to analyze our MBTS Base Station.

2 Background

In this chapter, we discuss the background information required for understanding the research done in this thesis. In Section 2.1, we discuss the most important parts of the TETRA protocol for our research. The setup of our testing environment of the BS is discussed in Section 2.2. The purpose and functionalities of our instrumentation Tinker, Fiddle and Cydia Substrate are explained in Section 2.3. Some background information about watchdogs is provided in Section 2.4. We provide some information about fuzzing and coverage guidance in Section 2.5. Section 2.6 explains why we have chosen for our emulation and fuzzing setup. We finish this chapter by discussing the concept of compiler optimization and the results on the decompiled output in Section 2.7.

2.1 The TETRA protocol

The TETRA protocol is a highly complex protocol. It has many functionalities and the TETRA standard consists of thousands of pages [8]. For the fuzzing research in this thesis, we are only interested in the data that is parsed by some packet parsing code. Therefore, we limit ourself to only a small part of the standard that is relevant for the packet parsing.

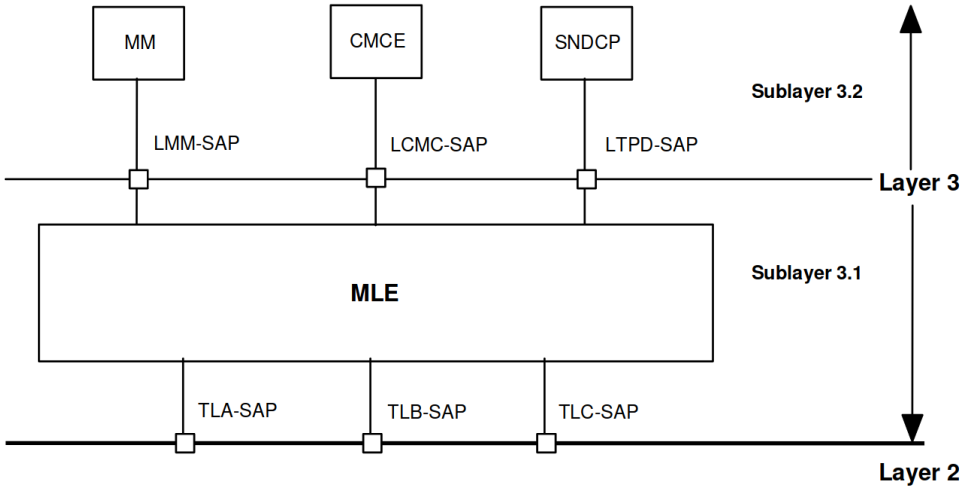

Figure 1: Overview of sublayer 3.1 and sublayer 3.2 of the TETRA protocol [8].

The TETRA protocol consists of multiple layers, each with sublayers. Layer 1 and layer 2 are related to the low level aspects of the protocol, like the (de)modulation, radio transmission and reception, channel coding/multiplexing and more related to the radio link. Layer 3 is related to the higher level of the protocol, such as packet interpretation, routing and compression of the packet data. Layer 4 and further are out of scope for this research.

The reference architecture of the start of layer 3 is shown in Figure 1. The communication between layer 2 and layer 3 is defined as TLA-SAP, TLB-SAP and TLC-SAP:

- TLA-SAP: responsible for signalling messages
- TLB-SAP: responsible for broadcasting system information
- TLC-SAP: responsible for layer management, status and configuration via data base access

TLA-SAP, TLB-SAP and TLC-SAP are the communication channels between layer 2 and layer 3 and are used by the Mobile Link Entity (MLE). The MLE is used as an abstraction by the Mobility Management (MM), Circuit Mode Control Entity (CMCE) and Subnetwork Dependent Convergence Protocol (SNDCP). The MM is responsible for several Mobile Station (MS) management functionalities, such as registration and de-registration to the network. The CMCE is a network layer and does packet parsing for the services built on top of CMCE, such as Call Control (CC) and Short Data Service (SDS).

Each sublayer of the layer 3 protocol (as shown in Figure 1) reads and interprets its own packet header. Therefore, it is likely that the packet parsing is done in several parts of the protocol, such as the MLE, MM and CMCE. This is important to remember when reverse engineering the firmwares of the Base Station (Section 5.1), because finding references to TLA-SAP, TLB-SAP, TLC-SAP, MLE, MM and CMCE may indicate that we are close to packet parsing code.

2.1.1 Use of cryptography in the TETRA protocol

The TETRA protocol may use encryption to protect data that is transported between Mobile Stations (MSs), Base Stations (BSs) and the Authentication Centre. The authentication of MSs and the exchange of encryption keys is managed by the BS and the Authentication Centre. The BS is assumed to be trusted by the infrastructure and has access to the keys used to encrypt and decrypt the transported data [9]. An overview of the keys used in the TETRA protocol, the MBTS BS and their purposes is given in Table 1 on page 11. An overview of the available keys in the MS, BS and the Authentication Centre is given in Figure 2.

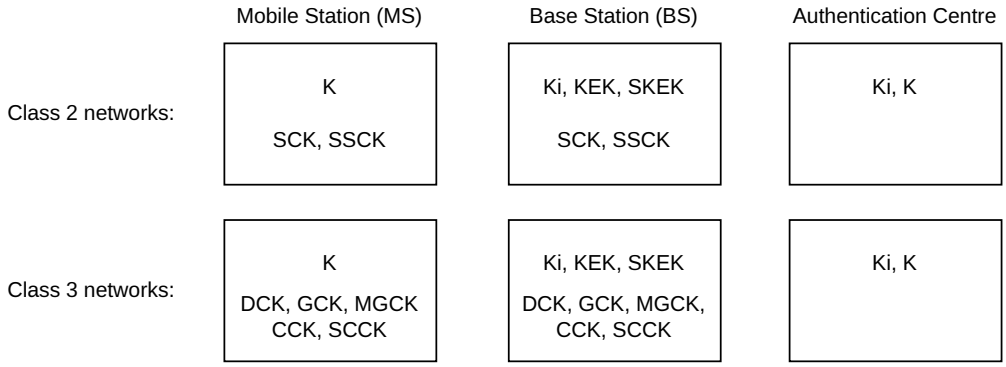

Figure 2: Overview of the availability of the keys listed in Table 1 for the Mobile Station, Base Station and the Authentication Centre in class 2 networks and class 3 networks.

Since the TETRA standard is highly complex and therefore difficult to understand, we first examine how the encryption is performed and then see which data is required to perform the encryption. In the end, this will result in an overview of which encryption keys are required to perform encryption and decryption.

In Figure 3, the final key derivation is shown to obtain a Key Stream Segment (KSS) from the Key Stream Generator (KSG). The KSS is used to XOR the data to perform encryption or decryption. The KSS is created from an Initialization Value (IV) and an Encryption Cipher Key (ECK). The IV is derived from broadcast data from the BS. The ECK is derived by running the proprietary TB5 algorithm over the Cipher Key (CK), Carrier Number (CN), Location Area id (LA) and Colour Code (CC). The CN,

Name	Category	Purpose
Authentication Key (K)	Authentication	Primary secret, the knowledge of which has to be demonstrated for authentication [9]. The Authentication Key is a pre-shared secret between the Mobile Station and the Authentication Centre. For example, this secret can be pre-shared by a SIM card or this secret is embedded in the Mobile Station.
Static Cipher Key (SCK)	Encryption	Encryption key used in class 2 networks to protect voice, data, and signalling sequences between the infrastructure and an individual Mobile Station.
Sealed Static Cipher Key (SSCK)	Encryption	The encrypted Static Cipher Key.
Common Cipher Key (CCK)	Encryption	Encryption key used in class 3 networks to encrypt communication within groups, such as voice data for a group call.
Sealed Common Cipher Key (SCCK)	Encryption	The encrypted Common Cipher Key.
Derived Cipher Key (DCK)	Encryption	Encryption key used in class 3 networks to encrypt one-to-one communication between devices. This key is a session key and is created during authentication of the Mobile Station to the Base Station.
Group Cipher Key (GCK)	Encryption	(In class 3 networks a configurable) encryption key for group communication in class 2 networks and class 3 networks used to create the Modified Group Cipher Key.
Modified Group Cipher Key (MGCK)	Encryption	If a Group Cipher Key is configured in class 3 networks, the Common Cipher Key is used to derive the Modified Group Cipher Key. The Modified Group Cipher Key is used for encryption of the transported data within the group.
Infrastructure Key (Ki)	Storage	The encryption key used to encrypt the Key Encryption Key. The Infrastructure Key is also used for different purposes related to infrastructure management.
Key Encryption Key (KEK)	Storage	Storage key used to encrypt encryption keys to store the keys in memory of the MBTS BS.
Sealed Key Encryption Key (SKEK)	Storage	The Key Encryption Key encrypted with the Infrastructure Key.

Table 1: Overview of the keys used in the TETRA protocol and the MBTS BS. The storage keys are used for encryption designed by Motorola and are not defined in the TETRA standard.

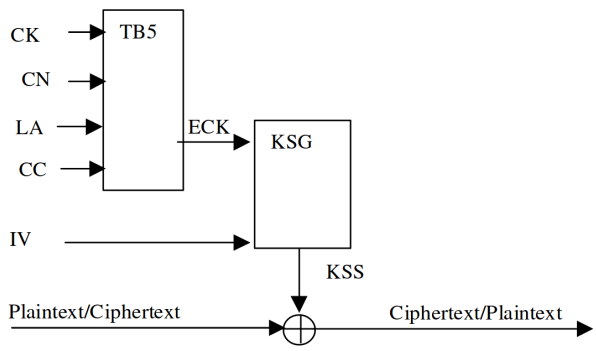

Figure 3: The final key derivation to create a Key Stream Segment (KSS) for encryption and decryption [9].

LA and CC are not secret values since they are known by the network. Which CK is used, depends on the class and method used by the TETRA network.

The class of the TETRA protocol depends on the use-case of the devices:

1. Class 1 networks have no encryption
2. Class 2 networks support encryption with the Static Cipher Key (SCK)
3. Class 3 networks support encryption with the Derived Cipher Key (DCK) or Common Cipher Key (CCK)

Downlink traffic may be sent to a group instead of an individual MS. In such a case, the SCK (class 2 networks) or CCK (class 3 networks) is used for encryption. A Group Cipher Key (GCK) may be configured for a specific group, which enables derivation of an Modified Group Cipher Key (MGCK) to be used instead, in order to achieve confidentiality between groups. Since the addressing mode does generally not affect parsing, group addressed traffic is outside of the scope of this research.

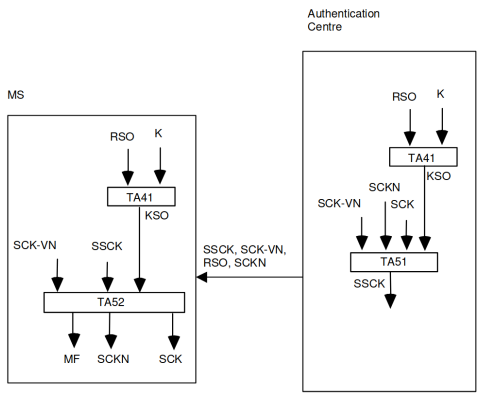

Figure 4: Distribution of SCK to a Mobile Station by an Authentication Centre [9].

The SCK used in class 2 networks is used to protect voice, data, and signalling sequences between the infrastructure and an individual MS. The SCK is a fixed value and should be known to the infrastructure and every MS [9]. In order to update the SCK, the SCK will be *sealed* (encrypted) by the proprietary TA51 algorithm using the Static Cipher Key Version Number (SCK-VN) and a Session Key for OTAR (KSO). The KSO is

derived from the K and a RSO, using the proprietary TA41 algorithm. To decrypt the SCK, you need to run the proprietary TA52 algorithm with the SCK-VN, the Sealed Static Cipher Key (SSCK) and the KSO. The distribution of the SCK is visualized in Figure 4.

The CCK is encrypted and updated in a similar way as the SCK. However, the CCK uses the proprietary TA31 algorithm to seal (encrypt) the CCK and the proprietary TA32 algorithm to unseal (decrypt) the CCK. Moreover, the KSO is not used as an input to the TA31 and TA32 algorithms, but instead the DCK is used.

The DCK is derived from the result of the authentication protocol. Therefore, the DCK is different for every session between the MS and the BS. Since the authentication protocol is long, complex and not relevant for this research, we will not go into detail of this.

From this, we can conclude that the encryption of the TETRA protocol for one-to-one data transportation mainly depends on:

- The KSG and TB5 algorithms, and the SCK for class 2 networks, using the KSO, the TA51, TA52 and TA41 proprietary algorithms.
- The KSG and TB5 algorithms, and the DCK for class 3 networks, using the TA31 and TA32 proprietary algorithms.

The encryption of the TETRA protocol for group data transportation mainly depends on:

- The KSG and TB5 algorithms, the SCK and an optional GCK for class 2 networks, using the KSO, the TA51, TA52 and TA41 proprietary algorithms.
- The KSG and TB5 algorithms, the CCK and an optional GCK for class 3 networks, using the TA31 and TA32 proprietary algorithms.

2.1.2 Use of cryptography in the storage of the MBTS Base Station

In Section 2.1.1 we have discussed the use of cryptography in the TETRA protocol as described in the TETRA specification. However, for our MBTS Base Station (BS) there are Motorola specific implementations for key storage. Key storage refers to the long-term storage of encryption keys in the BS. The key storage is not defined in the TETRA specification and therefore is vendor specific.

The Infrastructure Key (Ki) is an encryption key used to encrypt the Key Encryption Key (KEK). The Ki is also used for encrypting traffic related to infrastructure management. Therefore, the Ki can be seen as a master key for the BS. The Ki can be updated by using a Key Variable Loader (KVL) hardware device. The KVL uses an undocumented protocol to update the Ki over the serial port of the MBTS BS.

The MBTS BS uses encryption for long-term storage of encryption keys such as the Static Cipher Key (SCK) and Common Cipher Key (CCK) (see Section 2.1.1). These encryption keys are encrypted using the KEK. The KEK is unique per BS.

When the MBTS BS has booted and needs access to the encryption keys, the following is done:

1. The Ki is read.
2. The Sealed Key Encryption Key (SKEK) is decrypted using the Ki to obtain the KEK.

3. The KEK is used to decrypt the stored SCK to obtain the SCK.

The storage method used for encryption keys is relevant when we perform the manual firmware analysis to try to extract encryption keys from the BS as described in Section 6.

2.2 Setup of our MBTS Base Station

In this thesis, the Motorola MBTS BS [12] is used as the target used for testing and developing the fuzzer and for performing the manual firmware analysis. The MBTS BS can be seen in Figure 6 and Figure 7. Our MBTS BS runs on a PowerPC architecture and is powered by the *pSOSystem* Real-Time Operating System (RTOS). There are two (primary) firmwares used by our MBTS BS, each running on their own hardware component:

1. The Base Radio (BR) firmware. The BR can be seen in Figure 8.
2. The Site Controller (SC) firmware. The SC can be seen in Figure 9.

A visualization of the setup of our MBTS BS with the RTOS, firmwares and the data flow can be found in Figure 5.

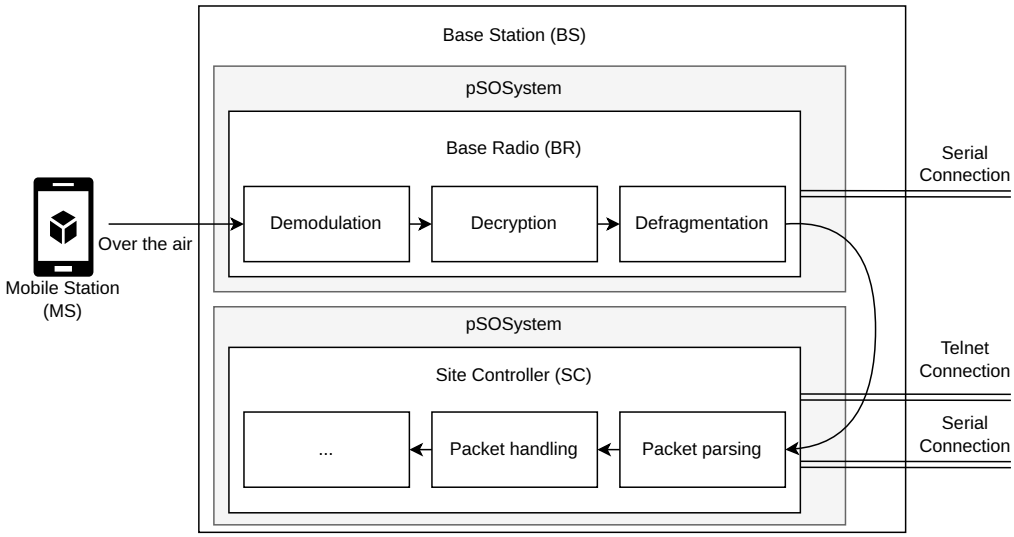

Figure 5: Setup of the Base Station, including the RTOS, the Base Radio and Site Controller firmwares, and the data flow.

The BR firmware is the first firmware that is handling incoming traffic on the Air Interface. The BR can be accessed through a serial connection for maintenance and configuration purposes. The BR is responsible for demodulation of the signal, decryption of the traffic and defragmentation of the packets. When the BR is done with the traffic, it will send the traffic to the SC over an internal network. The SC is responsible for the higher-level handling of the traffic. The SC can be accessed through a telnet connection or the serial port for maintenance and configuration purposes. The SC will parse the packet, interpret the packet and take actions accordingly. The SC will also create a response and send it back to the BR, and the BR will send it over the air.

The BR firmware and the SC firmware of our MBTS BS run on the pSOSystem (pSOS) RTOS. pSOS is a closed source RTOS and commonly used around the year 1980 until

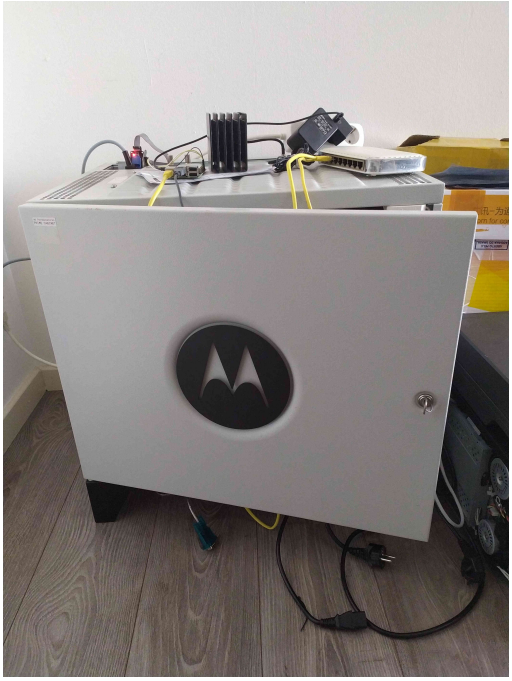

Figure 6: The front of our MBTS BS.

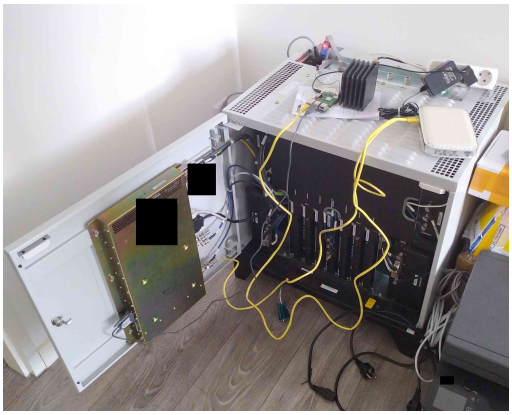

Figure 7: Overview of the inside of our MBTS BS.

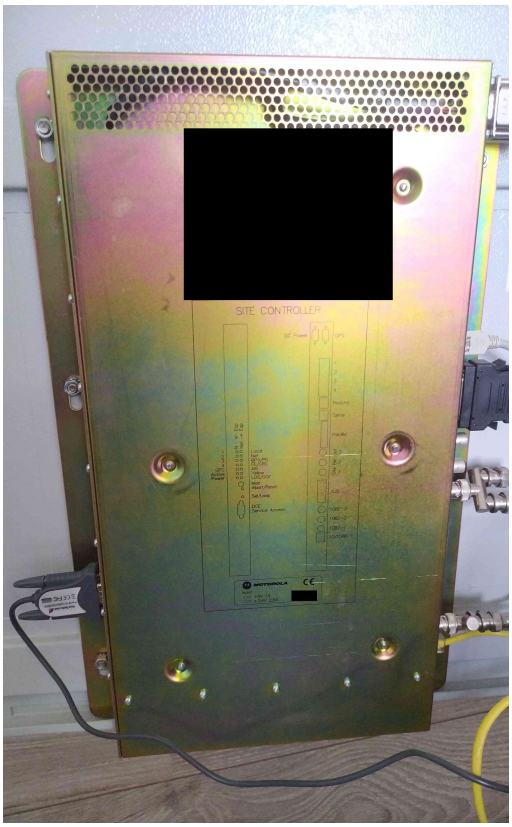

Figure 8: The Site Controller of our MBTS BS.

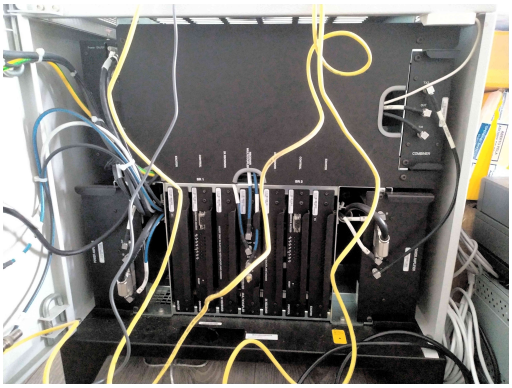

Figure 9: The Base Radio of our MBTS BS.

2000. Like many other RTOSs, pSOS supports multi-tasking, priority levels, predefined latencies for execution and much more. pSOS uses 'tasks'. 'tasks' can be seen as a process on a regular operating system: for each job that needs to be done, there exists a task which handles this job. A task receives input through Inter-Process Communication (IPC), which are called 'messages'. This IPC between tasks is handled by pSOS through message queues. These queues can be filled by other tasks and are consumed by the task corresponding to the queue. For example, the packet parsing is one task, while the handling of the packet is another task. The packet parsing task communicates the parsed packets through IPC to the packet handling task.

2.3 Tinker, Fiddle and Cydia Substrate

For the fuzzing research, we want to be able to execute arbitrary code in the Base Station (BS) to perform a memory dump so we can research the packet parsing code. In order to accomplish this, we are using the following:

1. Tinker: a firmware modification to the Site Controller (SC) firmware to provide read/write/execute/malloc/free primitives through a CLI command, which is accessible over the telnet connection.
2. Fiddle: a module loading framework that makes it possible to load an ELF binary into the firmware of the BS.
3. Cydia Substrate: a (function) hooking mechanism to redirect control flow to another function.

Tinker, Fiddle and a modified version of Cydia Substrate were all created and provided by Midnight Blue, the company where this research was carried out at. These tools are required to be able to research the BS and will eventually be open sourced.

The BR firmware and the SC firmware of the BS are modified to include Tinker. The SC firmware can be modified since no signature checks are done over the executed binaries. A function that is accessible over the telnet connection is overwritten with Tinker code. Therefore, Tinker is accessible as a shell command. Tinker provides arbitrary read/write/execute/malloc/free primitives as a shell command. Tinker is used by Fiddle to load ELF binaries into the firmware without requiring static firmware modifications.

Fiddle is a module loading framework that makes it possible to load an ELF binary into the firmware of the BS. Fiddle communicates over the telnet connection with Tinker. Fiddle uses Tinker's malloc to allocate sufficient memory in order to write the binary to this allocated memory. The malloc implementation of Tinker will return the base address of the allocated memory. Fiddle uses this base address to re-locate the ELF binary. This means that the addresses within the ELF binary are modified to use the address given by malloc as the base address of the binary. Therefore, the ELF binary will execute correctly when written and executed on this allocated memory. After this ELF binary modification, Fiddle will write the new ELF binary to the allocated memory. Fiddle will then execute the initialization function of the ELF binary to start code execution. It is also possible for Fiddle to unload the module by executing the de-initialization function and then free the allocated memory.

The ELF binary that is loaded into the memory of the BS includes a modified version of Cydia Substrate. Cydia Substrate is a code modification platform used in the jailbreaking scenes about a decade ago [20]. Using Cydia Substrate, it is possible to inject arbitrary code into other processes to change the behavior of the process by executing the injected code. We are using a modified version of Cydia Substrate which supports

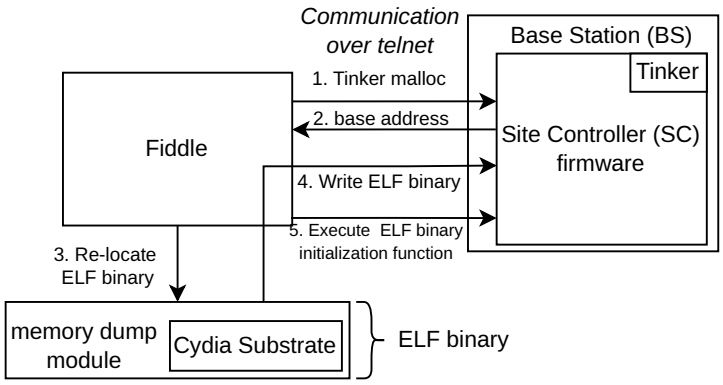

Figure 10: The steps of loading the ELF binary into the SC using Tinker, Fiddle and Cydia Substrate.

the PowerPC architecture, so it can be used on our MBTS BS for research purposes. The modified version of Cydia Substrate is used to hook functions within the firmware to redirect the control flow. This redirection of control flow is useful for creating the memory dump module, since we want to execute the memory dump code instead of the functions responsible for parsing the packet.

The steps of loading the ELF binary into the SC are visualized in Figure 10.

2.4 Ensuring system availability through watchdogs

As explained in Section 2.3, we are able to execute arbitrary code on our MBTS BS. However, we are not completely unrestricted in what we can execute since the MBTS BS is protected by a *watchdog*. Because of the watchdog, we are limited by an execution time of around 30 seconds of our arbitrary code (see Section 5.2.3). In order to remove this restriction provided by the watchdog, we first have to better understand what a watchdog exactly is.

Watchdogs are a common method in Real-Time Operating Systems to ensure availability of the devices [31]. The purpose of a watchdog is to check if the device or software is still running and not caught in an infinite loop. A watchdog can be implemented in hardware, in software or in both. There are different methods to accomplish this monitoring, but for this thesis we focus on hardware watchdogs since that is also used our MBTS BS.

A hardware watchdog has two components: the hardware monitoring, and the software watchdog 'feeding'. Build into the hardware of our MBTS BS, there is a watchdog component which checks the software running on the chips. It does this by checking the incoming I/O communication on two ports and keeping track of a timer. The watchdog expects that the software component will alternately send a specific constant to the two ports. As long as the watchdog will receive the constant, it will reset the timer.

However, if the software is caught in an infinite loop and does not send the constant in time to the watchdog, the timer of the watchdog expires. If the timer is expired (the length of the timer is dependent on the configuration in the hardware), the watchdog will reset the device. Some hardware watchdogs first signal to the Real-Time Operating System (RTOS) that a reset is about to happen. This gives the RTOS the opportunity to log debugging information before it is being reset, in order to help the developers discover the root cause of the infinite loop. It is also possible that the hardware watchdog skips this step and only performs the last step: resetting the device. The hardware watchdog

will trigger a full reboot of the device. This should temporarily resolve the issues caused by the infinite loop and make the BS available again.

The hardware watchdog is accompanied with a software implementation. The programs and RTOS that are running on the device should communicate to the watchdog that they are not caught in an infinite loop. However, the programs and RTOS should also do other jobs. This is why a RTOS is commonly implemented using tasks (Section 2.2). The watchdog timer task is a lower priority task. This task is responsible for alternately communicating the constant value to the port of the hardware watchdog. This is called *feeding* the watchdog. As long as the value is send to the hardware watchdog, the device will not be reset. If the device is idling, this task will be executed. However, if other tasks are being executed, this task will not block the other tasks since it is a low priority task.

Depending on the RTOS, it is possible to have deadlines for a task [2]. If the RTOS supports deadlines, the watchdog feeding implementation can be aware that the task should be executed at least once in a predefined interval. Without this, if the RTOS is busy with other tasks for a long time, it may be possible that the watchdog task will not be executed for too long. This would cause a reset of the device, even though it was not caught in an infinite loop. With deadlines, the RTOS makes sure that the watchdog feeding will happen at least once in a predefined interval which will prevent the reset of the device. But it may not interrupt any other tasks, so the watchdog feeding task will not interrupt a task that is caught in an infinite loop.

2.5 Fuzzing & coverage guidance

Fuzzing is an automated software testing technique used by security researchers to find bugs in all kinds of software. A fuzzer generates 'random' inputs for a program based on given valid inputs, known as a *seed*. If the program crashes with one of the 'random' inputs, for example by a failing assertion or memory corruption, a bug is found in the program. A bug found by fuzzing does not always have to be exploitable and thus a security issue.

The inputs are 'random' because this is not the only input generation method a fuzzer uses. When a fuzzer is started, it may indeed start generating random inputs based on the seed. However, *coverage guidance* is also a very important aspect for effectively finding bugs by fuzzing.

Coverage guidance is used by a fuzzer to know if an input creates a new execution path. Each input can be seen as an execution *path* through the code. When fuzzing, it is important to know if a new input creates a new path, because a new path could contain a bug. Moreover, different paths may result in different states of the program in which new bugs can occur. Therefore, maximizing code coverage is desirable during fuzzing for increasing the probability of discovering an execution path with a bug.

In this thesis, we use AFL++ as the fuzzer. The reason for this, is that it a very commonly used fuzzer used by security researchers with lots of features and support, and it is actively maintained. In Section 2.6 we also show that we can achieve coverage guidance by using AFL++ and Unicorn as the emulation framework. Using coverage guidance, AFL++ can determine the effectiveness of a generated input. If a randomly generated input results in a new path, then this path is marked as interesting for AFL++. AFL++ will then use this interesting input as a seed for generating new test cases. Using the coverage guidance methodology, AFL++ is able to discover new bugs very effectively.

2.6 Choosing the emulation framework

In this thesis, the goal is to fuzz the packet parsing code of our MBTS BS. We perform fuzzing through emulating the hardware and the pSOSystem kernel of our MBTS BS, so we can run the packet parsing code on the emulator. With this emulation, we can attach a fuzzer to the executed code and we can start fuzzing the packet parsing code effectively with coverage guidance. In order to make this happen, we have to choose which emulator and fuzzer we are going to use.

We have the following requirements for the emulator:

1. The emulator supports the PowerPC architecture.
2. The emulator can be used with a fuzzer for coverage guidance.
3. The emulator can load the memory dumps into the emulator (not just a cold boot from an `.elf` file).
4. The emulator can be extended to create better support for the target we are going to emulate such as through manual implementation of system calls.

One of the most well-known open source emulation software used in the Linux environment, is QEMU [17]. QEMU is a hardware emulator[1]: it can emulate processors of different architectures, such as the PowerPC architecture. However, it is not an obvious candidate for our fuzzing research because it is not trivial to load arbitrary memory dumps in QEMU. It is also not possible to (easily) interact with the emulated CPU inside QEMU, and modify register values for example. This is one of the reasons Unicorn has been created.

Unicorn is a lightweight multi-platform, multi-architecture CPU emulation framework [19] based on QEMU. With Unicorn, it is easy to modify the state of the memory and CPU registers due to the features build on top of their fork of QEMU. Moreover, hooking is fully supported in Unicorn, which means that you can easily change the behavior or implementation of a function in the firmware. Since July 7th 2022, Unicorn 2.0 has been released which introduced support for the PowerPC architecture. This makes it possible for Unicorn to be used in this research as an emulation framework. Furthermore, Unicorn combined with a memory dump has proven to be useful for devices that are difficult to fuzz, such as the packet radio receiver FSK_Messaging_Service [29, 30]. Unicorn also supports attaching AFL++ as a fuzzer to the emulated code using UnicornAFL [24]. One downside of Unicorn is that Unicorn does not support system calls (syscalls). Syscalls are not supported, since Unicorn does not emulate the kernel, unlike QEMU. If the emulated code invokes a syscall, then it will not do anything unless a custom implementation is provided using the hooking mechanism.

Another emulation framework is Qiling [18]. Qiling is an advanced binary emulation framework extending Unicorn. It promises all the benefits of Unicorn, including support for the PowerPC architecture. Futhermore, it adds support for various syscalls, debugging capabilities and fine-grained hooking levels. The API of Qiling is exposed through Python bindings. Qiling itself is also implemented in Python. In our experience, this results in slow execution because of the overhead Python introduces [22]. Moreover, there are issues with Qiling's behavior when debugging is enabled, differences in the expected and actual behavior of specific syscalls, and the lack of documentation [14]. For these reasons, we have decided to choose Unicorn as the emulator we are going to use.

[1] QEMU can also be used for virtualization, but that is not relevant in this thesis.

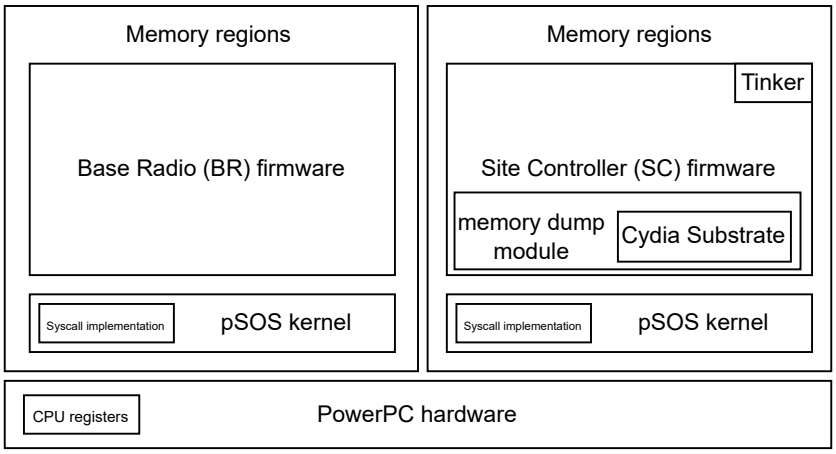

Figure 11: Setup of firmwares of the BS, including the modifications of the SC firmware.

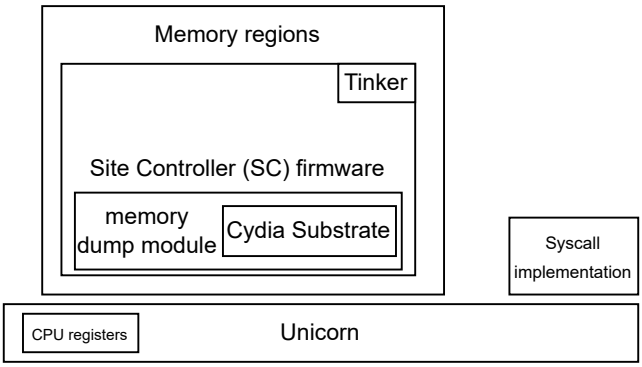

Figure 12: Setup of the SC firmware for emulation with Unicorn.

Unicorn exposes an API for different languages, such as Python, Java, Rust and C. Since we would like a high-performance emulation and fuzzing framework, we decided to go with the Rust bindings. We have chosen Rust since it is a high performance and low-level language and unlike C, it provides strict compile-time checks to ensure reliability. To improve the debugging capabilities of Unicorn, we will attach udbserver to the Unicorn engine. Udbserver, the Unicorn Emulator Debug Server, exposes the GDB Remote Serial Protocol for Unicorn, which allows one to connect to the emulator with GDB [3]. This can be useful to inspect the memory regions and registers of the firmware during execution, and to triage any bugs found during fuzzing.

An overview of what the setup is of the firmwares executed on the BS, can be found in Figure 11. An overview of what is emulated by Unicorn and what is executed on top of it, can be found in Figure 12.

2.6.1 Rust harness with Unicorn for fuzzing with AFL++

With the Rust bindings of Unicorn, we can write Rust code to load the memory dumps into Unicorn and set the CPU registers for Unicorn. Moreover, Unicorn supports fuzzing with AFL++. After we have performed all the initialization of Unicorn in the Rust code and we are ready to start the emulation, we can execute the function `afl_fuzz` to make the program compatible with the command line tools of AFL++. The program is then

called the *harness* for AFL++. AFL++ is able to run the harness and perform IPC with the harness to efficiently fuzz inside Unicorn.

2.7 Compiling and decompiling

In this section we discuss the effects of compiler optimization settings on the resulting compiled code, and hence the decompilation output when reverse engineering the compiled code. This is important to know during reverse engineering when comparing code in different firmwares, since code that is being compared may be equal but compiled using a different optimization setting.

Compiling is the process of converting source code into assembly instructions. Decompiling is the reversed process: converting assembly instructions back into source code. For compiling source code, a compiler such as `gcc` can be used. `gcc` is capable of converting source code written in C (and other languages) into assembly instructions.

When compiling source code using `gcc` without specifying any optimization options, `gcc`'s goal is to reduce the cost of compilation and to make debugging produce the expected results [25]. If an optimization option is specified, depending on which option is specified, `gcc` will change the resulting assembly instructions to improve performance and/or code size, with a penalty on compilation time. There are numerous optimization options available in `gcc`, each with their own goal. All these optimization options are bundled into four levels: `-O0` (no optimization), `-O1` (optimize), `-O2` (optimize even more), `-O3` (optimize yet more). Depending on which level is chosen during compiling the source code, the decompiled code may change drastically. This is best explained by an example program written in C:

```
#include <stdio.h>

int generate_number()
{
    for (size_t i = 0; i < 2; i++)
    {
        printf("Computing...\n");
    }

    return 42;
}

int main()
{
    for (size_t i = 0; i < 99999999; i++)
    {
        printf("Random number: %d", generate_number());
    }

    return 0;
}
```

For this example, we use Ghidra [1] as the decompiler to convert the assembly instructions back into source code.

When compiled with `gcc` without any optimization specified, the decompilation is:

```
undefined8 generate_number(void)
```

```
{
    ulong local_10;

    for (local_10 = 0; local_10 < 2; local_10 = local_10 + 1)
       {
        puts("Computing...");
    }
    return 0x2a;
}

undefined8 main(void)
{
    undefined8 uVar1;
    ulong local_10;

    for (local_10 = 0; local_10 < 99999999; local_10 = local_
       10 + 1) {
        uVar1 = generate_number();
        printf("Random number: %d",(int)uVar1);
    }
    return 0;
}
```

When compiled with `gcc` with the `-O3` optimization specified, the decompilation will be:

```
undefined8 main(void)
{
    long lVar1;

    lVar1 = 99999999;
    do {
        puts("Computing...");
        puts("Computing...");
        printf("Random number: %d",0x2a);
        lVar1 = lVar1 + -1;
    } while (lVar1 != 0);
    return 0;
}
```

As seen in de the above decompilation outputs, the decompiled code differs significantly when using the `-O3` optimization compared to the decompiled code without any optimizations specified. In the decompilation of the optimized binary, we see no function call anymore to the `generate_number` function. Instead, the whole function is inlined in the `main` function and the for-loop of printing "Computing..." is optimized away. Also, the for-loop in the `main` function is replaced with a do-while loop. Still both the non-optimized binary and the optimized binary gives equal output.

From this example it is important to remember that even if the decompiled code may be significantly different, it could be the case that they originate from the same source code but compiled with different optimization settings. This becomes relevant when comparing firmware of our MBTS BS with the firmware of the MTS2 BS, as described in Section 3.2, since the firmwares are compiled using different optimization settings.

3 Earlier research for the TETRA Base Station

Security research on a TETRA Base Station (BS) is not completely new. At the TU Darmstadt, there has also been research carried out on the TETRA BS by Müller et al. [16]. In this chapter, we review the research by Müller et al. (Section 3.1). To highlight that this research is not a mere repetition of their work, we make the differences of firmware and fuzzing between the research of Müller et al. and this research explicit in Section 3.2 and Section 3.3. In Section 8.3 in the Conclusion, we reflect on the usefulness of the research of Müller et al. for our own research.

3.1 PowerPC Binary Patching for Base Station Analysis

The research done by Müller et al. is focussed on providing tools for analyzing the firmware of a Base Station (BS). These tools include a memory dumper, function tracer, flexible patching capabilities at runtime, and a custom fuzzer. The flexible patching capabilities are made in a generic way so they can be reused for research on other systems running the PowerPC architecture. Besides the patching tools, their research is focused on the Base Radio (BR) firmware and the communication stack between the BR firmware and the Site Controller (SC) firmware.

Prefix	Purpose
`__`	`zlib`, symbol names match library [11]
`__`	`libc`, symbol names match library [5]
`efs_`	High-level file system functionality
`ipcom_`	IP communication
`iplite_`	IP communication
`iptcp_`	Transmission Control Protocol (TCP)
`iptftp_`	Trivial File Transfer Protocol (TFTP)
`tftp_`	Trivial File Transfer Protocol (TFTP)
`snmp_`	Probably Net-SNMP library.
`scomm_`	Site communication with UDP socket abstraction.
`pthread_`	OSE POSIX-compliant thread wrapper.
`ose_`	Generic OSE functions.
`afm_`	OSE Atomic File Manager (AFM).
`fam_`	OSE Flash Access Manager (FAM).
`shell_`	OSE Command Line Shell.
`cmd_`	Shell commands like ls or cat.
`rtc_`	OSE Real Time Clock (RTC).
`pmd_`	OSE Post Mortem Dump (PMD).
`bs_`	Probably basic system process and timer management.
`core_`	Core functionality.
`sysconf_`	Configuration access.
`zz`	Functions that force the syscall interface.
`xx`	Kernel-side implementation of functions like `xxmutex_lock`

Table 2: Symbol names in the firmware of the BR of the MTS2 BS and their purpose, from the research by Müller et al. [16].

Since the research of Müller et al. focused on the firmware of the BR, they have also published a detailed table of symbol prefixes and their purposes. This table of symbols can be found in Table 2. Symbols are added by the compiler during compilation to make

it possible to dynamically link shared libraries. Symbols can also be used to identify what happened where, when there is a crash. For example, the symbols in the firmware include function names of every function in the firmware. This is useful information to better understand the structure of the binaries, and where interesting code may be.

The research of Müller et al. concludes with *Hyphuzz*, a fuzzer that has been created using the tools they developed in order to fuzz the MTS2 BS (see Section 3.3).

3.2 Firmware differences

The MTS2 Base Station (BS) hardware and firmwares used in the research of Müller et al. is not the same as the hardware and firmwares used in our research. This has impact on what is useful for this thesis from the research of Müller et al.

The hardware that was used in the research of Müller et al., is a DIMETRA Express MTS2 BS [15]. This is different from the BS used in our research: the DIMETRA Express MBTS BS. The MTS2 BS is the successor of the MBTS BS. The MTS2 BS is powered by the *Enea* Operating System Embedded (OSE) [6] while our MBTS BS is powered by *pSOSystem* (pSOS).

The firmwares used in the research of Müller et al. include symbols. These symbols are also used by decompilation software to make the decompiled code more readable and easier to understand. The function names also reveal a lot about the use and purpose of the code in the function. This can be helpful in reverse engineering the Site Controller (SC) firmware. The firmwares of our MBTS BS do not have symbols.

The firmwares of our MBTS BS also appears to be older than the firmwares of the MTS2 BS. We concluded this, since we noticed that the firmwares of the MTS2 BS is larger in size than the firmwares of our MBTS BS. Parts of the SC firmware of the MTS2 BS is also written in C++ while the SC firmware of our MBTS BS does not include any C++ code. Moreover, strings containing a copyright date indicate that the MBTS BS firmware has been created before the MTS2 BS firmware.

While reverse engineering and comparing the SC firmware of the MTS2 BS and our MBTS BS, we noticed that the SC firmware of their MTS2 BS contains symbols related to C++ functions. For example, there are symbols named `AiProtocolConsumer::handleUnpack`, which uses C++ namespaces. This reveals that besides the use of C, some code is also implemented in C++ which is called by the C code. Following the same code paths in the SC firmware of our MBTS BS shows us different assembly instructions and decompilation which are related to C code and not C++ code. This indicates that some parts of the C code are re-written/re-implemented in C++ in the SC firmware of their MTS2 BS.

Another difference is that the firmwares of our MBTS BS seems to be compiled using a higher optimization setting compared to the firmwares of the MTS2 BS. This is indicated by the high amount of inline functions in the MBTS BS firmware. As explained in Section 2.7, this results in different decompilation output which makes it in some cases more difficult to compare the two firmwares.

3.3 Fuzzing differences

The research of Müller et al. focusses primarily on understanding and modifying the firmware of their own Base Station (BS). They created their own custom PowerPC binary patcher which makes it also possible to hook functions in the firmware at runtime. So they created a platform similar to Fiddle and Cydia Substrate (see Section 2.3), but

with their own implementation and extra functionalities. While their tool suite has more functionalities, such as call traces, and is more integrated in the underlying operating system, we are mostly interested in the fuzzing part.

For the fuzzing, they have chosen to use the original hardware to run the modified firmware on. This is because they concluded that it is not feasible to run the binary in an emulator such as Unicorn, because of hardware dependencies of the firmware. Therefore, their *Hyphuzz* fuzzer has to run on another device and interacts with the BS through a UDP connection for executing the test cases. This means that they are only able to observe crashes from the outside, and do not have detailed information about why it crashes. 45% of their crashes contained a kernel error message, which is helpful for triage and root cause analysis. But this also means that 55% of the crashes cannot be triaged. Moreover, the state of the BS is not reset between test cases which can be a source of instability for their test cases. Most importantly, they fuzz the `scomm` stack, which is used for trusted backbone services. Being able to communicate with the `scomm` stack already implies that the adversary has crossed an important trust boundary since the adversary is able to reconfigure the BS.

4 Research Decisions and Preparation

4.1 Decisions

In this section we discuss some initial decisions we made when starting our fuzzing research (Section 5). In Section 4.1.1 we discuss how we are going to use the MBTS BS and MTS2 BS firmwares in our advantage. Section 4.1.2 clarifies the decision of making a memory dump, and the benefits of it during fuzzing.

4.1.1 Comparing the firmware of our MBTS Base Station and the MTS2 Base Station

As discussed in Section 3.2, we have access to two firmware binaries:

1. The firmware binary of our MBTS BS (the device used in this research), which does not include symbols.
2. The firmware binary of the MTS2 BS (the device used in the research of Müller et al.), which does include symbols.

While we have to keep in mind that there are differences between the two firmwares, we can also make use of the firmware of the MTS2 BS to better understand our MBTS BS firmware. Because of the symbols in the MTS2 BS firmware, the MTS2 BS firmware is easier to understand. Furthermore, since a large part of the code is shared between the two firmwares, we think it is faster to first search for the packet parsing code in the MTS2 BS firmware. Afterwards we can check our MBTS BS firmware to see if the packet parsing code is also implemented there. Moreover, using pattern matching on the decompiled code and looking for static strings within the code, we can identify the functions in the MTS2 BS firmware in our MBTS BS firmware (see also Section 5.1.1).

4.1.2 Emulation & Fuzzing

As discussed in Section 3.3, Müller et al. makes use of the actual hardware during the fuzzing of the Base Station (BS). In our research, we do not make use of the hardware during the fuzzing phase. Instead we choose to run it in an emulator. The rationale behind this choice is threefold. First, emulation allows for effective coverage guidance, which will greatly increase the relevance of generated fuzzing cases. Second, emulation helps verifying the execution path and debugging any issues found in the code because we can freely inspect the memory regions and registers of the emulator at the moment of crashing. Without emulation, it is difficult to inspect the internal state of our MBTS BS. Lastly, emulation will increase execution speed since the hardware of our MBTS BS is old and slow.

However, it is not possible to load the firmware of the BS directly into an emulator (simulating a cold boot). This is because during boot, the BS firmware does hardware initialization which cannot be easily simulated in an emulator due to hardware dependencies. Instead, we create a memory dump of all memory regions and registers of the BS at the moment of packet parsing on the actual hardware. This memory dump will contain all the code of the firmware and operating system, and all data related to it. Using this memory dump for the internal state of the emulator, we completely skip all the hardware initialization which makes the emulation less hardware dependent.

4.2 Feasibility study of fuzzing through Unicorn with AFL++

Before deciding to use Unicorn as an emulation framework, we want to verify the following:

1. Unicorn supports loading in the memory dumps and CPU registers.
2. Unicorn supports running the PowerPC architecture of the memory dumps.
3. AFL++ properly supports Unicorn used in the fuzzing harness and the bindings are stable.
4. The speed of fuzzing with AFL++ and a Rust-based Unicorn harness is sufficiently high.
5. Debugging the memory dump through Unicorn with GDB is properly supported and stable.

Therefore, we created a small demo using *fuzzgoat* to verify the decision of using Unicorn as our emulation method.

4.2.1 Fuzzgoat

Fuzzgoat is a C program that has been deliberately backdoored with several memory corruption bugs to test the efficacy of fuzzers and other analysis tools [21]. Fuzzgoat already provides a setup for installing and using AFL++. However, we are not interested in the usual AFL++ setup where we just execute the binary. Since we are going to emulate a memory dump in Unicorn for the Base Station (BS), we will do the same for fuzzgoat.

For the fuzzgoat demo, we created a setup to use Unicorn with Rust bindings with AFL++ as the fuzzer. This setup includes setting the CPU registers, loading the memory dump and attaching AFL++ to Unicorn. This setup can be used as a boilerplate for the emulation, since not a lot of code has to be changed in order to port it to a different target.

Since the BS has a PowerPC architecture, we have compiled fuzzgoat to the PowerPC architecture too. In order to make a memory dump, we run the PowerPC binary of fuzzgoat with QEMU user emulation and attach a GDB debugger. Through GDB, we break just before the parsing and we dump all available memory regions to a file and we save the CPU registers as shown by GDB. Now the memory dump and CPU registers can be loaded into Unicorn through the Rust code.

With this setup, we have fuzzed fuzzgoat and we were successfully able to find the intended vulnerabilities in fuzzgoat. Moreover, AFL++ reported that the execution speed was around 1000 executions per second. From this we can conclude that it is a good approach for fuzzing the BS, since the method does discover vulnerabilities and has good execution speeds.

In order to improve debugging capabilities of Unicorn, we have connected udbserver to Unicorn [3]. Since we experienced segmentation faults with udbserver, we have created a fork with a patch to fix the crashes [4].

5 Fuzzing the packet parsing code of the Base Station

In order to fuzz the packet parsing code in the Site Controller (SC) firmware of the BS, we first have to go through the following steps:

1. We have to find the functions that implement the packet parsing (Section 5.1).
2. We have to write a memory dump module to dump all memory when invoked (Section 5.2).
3. We have to execute the memory dump module in the SC firmware (Section 5.3).
4. We have to load the memory dump into the Unicorn emulator (Section 5.4).
5. We have to extend Unicorn to deal with syscalls of the pSOS kernel (Section 5.4).

After we have done all the above steps, we can start fuzzing the packet parsing (Section 5.5). We have to do all the above steps since we want to fuzz the packet parsing through Unicorn to have coverage guidance available (see also Section 4.1.2).

In Figure 13, the flow of creating a memory dump and running it in the Unicorn emulator is summarized. Figure 13 visualizes the hooking in the BS, which redirects the execution flow to the memory dump module, which is step (1), (2) and (3) together. It will produce a memory dump which contains the CPU registers and the memory regions of the SC firmware. This memory dump will be used for step (4) and (5) to fuzz the packet parsing code on an external device using AFL++.

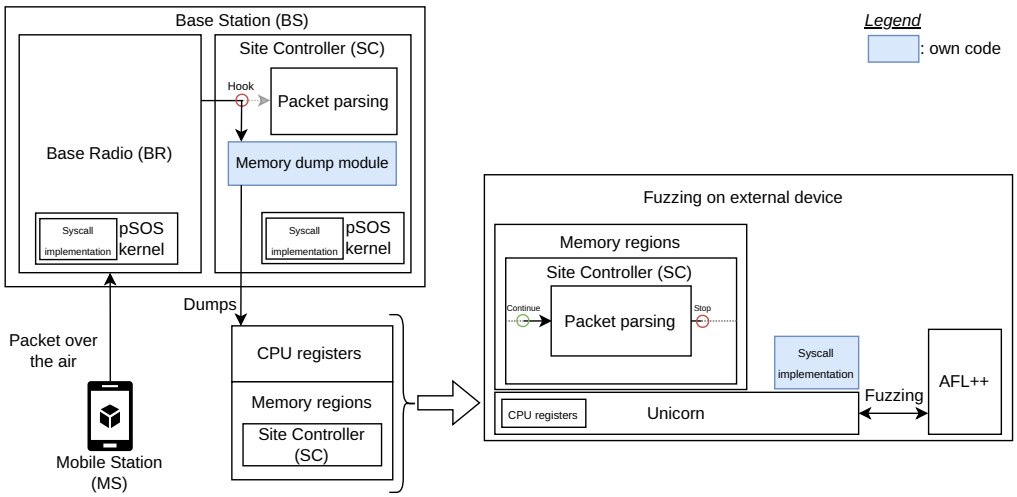

Figure 13: The flow of triggering and creating the memory dump on the BS, emulating the packet parsing in Unicorn, and fuzzing the packet parsing with AFL++.

5.1 Finding the target through reverse engineering

In this section, we describe the method we used to find the 'target function'. The target function is the function implementing the packet parsing. We want to fuzz the logic in this function, but first we have to locate the the function in the assembly of the firmware.

5.1.1 Procedure for finding functions

As explained in Section 4.1.1, we have our MBTS BS firmware without any symbols and the MTS2 BS firmware with symbols. We use the following procedure to find functions in our MBTS BS while using the symbols of the MTS2 BS firmware:

1. We first determine in the MTS2 BS firmware which function we want to find in our MBTS BS. We call this function f.
2. We search for static strings in the decompiled code referenced by f, or the parent/child functions of f.
3. If we have found a static string in the decompiled code, we search for this exact same string in our MBTS BS firmware.
4. If we have found such a string in our MBTS BS firmware, we identify references to this string. Such a reference comes either from f, or from a function closely related to f.
5. If the found string was in a parent/child function of f, we have to try to follow the reversed code path in the decompiled code of how we found that string. For example, if the found string was in a child function of f, we have to traverse all the parent functions of the function where we found the string. If the decompiled code of one of the parent functions mostly matches with f, we have found f in our MBTS BS firmware.

5.1.2 Analyzing the Base Radio (BR) through reverse engineering

Since we assumed that the Base Radio (BR) firmware would contain the logic for packet parsing, we started with reverse engineering the BR firmware[2]. The BR firmware of our MBTS BS is difficult to understand due to the lack of symbols, so we started with the firmware of the MTS2 BS.

Some symbol prefixes are already documented in Table 2 as a result of the research done by Müller et al., but there are more symbol prefixes in the firmware. Therefore, we use some educated guesses to give other relevant prefixes (related to the Air Interface) a useful name. The result is listed in the Table 3.

Prefix	Purpose
`dlai_`	DownLink Air Interface
`ulai_`	UpLink Air Interface
`aiei_`	Air Interface Encryption Interface
`ai`	Air Interface

Table 3: Description of the Air Interface-related symbols in the BR firmware.

Futhermore, we noticed that `_ld_` would mean "Late Decryption" if the function name contains that string. We think this because of the strings in the `aie_ld_task` function.

According to the TETRA standard, the *Uplink* is the inbound data path, i.e. from the Mobile Station to the BS [7]. This is also the path we want to research, since packet parsing is performed on the incoming data.

Following the cross references from functions starting with `ulai_`, brings us to the `ula_main` function. This is the Uplink's main function, and it probably contains all the

[2]We later conclude that this assumption is wrong and we continued with the SC (see Section 5.1.3).

BR-related code for the Uplink. After inspecting all the assembly and called functions in `ula_main`, we concluded that almost no packet parsing is done in the BR firmware. The BR firmware is mostly responsible for encrypting/decrypting the packets and passing it to the Site Controller (SC) firmware. Therefore, the target for this research shifted from the BR to the SC.

5.1.3 Analyzing the Site Controller through reverse engineering

For the Site Controller (SC), we used the same method as the Base Radio (BR): we first reverse engineer the SC firmware of the MTS2 BS since it is easier to understand with symbols. After we find something interesting, we compare it with the SC firmware of our MBTS BS to see if the implementation is similar or different.

For the SC there was no symbol interpretation done by Müller et al. [16], since they focused on the BR firmware. Therefore, we have to start from scratch and try to understand the structure of the SC firmware. There are some function names reused from the BR in the SC, however they are not relevant to the packet parsing logic. After glancing over many strings in the SC and some educational guesses, we concluded that functions containing the following names may have the following purposes:

Partial name	Purpose
`tsc`	Tetra Site Controller
`Ai`	Air Interface
`uplink`	Related to Uplink
`aie`	Air Interface Encryption
`Shm`	Site Health Monitoring
`ss_`	System Service
`rrd_`	Rom Data Handler
`fm_`	Fault Management (error handling)
`SMTask`	State Manager task
`alm_`	Alarm Management
`ic_`	Info Cluster
`CCMT`	Call Control Manager Type
`SCFG`	Save Configuration
`br_`	Base Radio
`aie`	Air Interface Encryption

Table 4: Description of the symbols in the SC firmware.

Using the symbols from Table 4, it is a bit easier to understand the SC firmware. Next, we need to find the target function which is responsible for packet parsing.

The strategy of finding the target function is mostly searching for symbol names containing the strings 'ai', 'parse', 'packet', 'pack', 'unpack', 'tetra', 'uplink', 'handle' and 'process'. These strings may indicate that the function is (nearby code) related to packet parsing code.

A candidate in the SC firmware was the `ai_unpack` function. Following the call graph upward from this `ai_unpack` function brings us to the `CallProcessing` function, which indicates that it has something to do with incoming data. We also crossed the function `cr_handle_msg_tla_tlc`, which is related to TLA and TLC data (see Section 2.1). Looking into the implementation of the `ai_unpack` function, shows us that it is a starting point of calling C++ related logic for `AiProtocolConsumer::handleUnpack`. However, we

first want to verify that this function actually exists in our MBTS BS firmware since all this reverse engineering is done in the MTS2 BS firmware.

By finding strings defined in the decompiled code in nearby functions of `ai_unpack`, we managed to find one of the parent functions `z0001002H1J_handle_tla_ex_data_ind` in the MBTS BS SC firmware. However, it appears that the function `ai_unpack` does not exist in our MBTS BS, since the disassembly is different in our MBTS BS. This is not very surprising since we did not find any C++ related code in our MBTS BS, so the code path to the `ai_unpack` function also does not exist in our MBTS BS. But the `ai_unpack` function has given us some new direction for where to look into to find packet parsing logic.

One of the parent functions of `z0001002H1J_handle_tla_ex_data_ind` is `yPAD_z0001002H_DataTransferEntityType`. The disassembly of `yPAD_z0001002H_DataTransferEntityType` in the MTS2 BS shows us that some big memory object is allocated at the start, and afterwards it will go into a huge switch-statement with about 90 cases. The case in the switch-statement is determined by the first two bytes of the memory object. Every case in the switch-statement will call a function, which will do something with the allocated object and then change the first two bytes of the memory object to another case-number in the switch-statement. This implementation feels very similar to a state machine, where the memory object is the state and the switch-statement implements the machine. Using this knowledge, we focused on finding more packet parsing related code within this state machine.

Continuing the search for symbol names, we found the `z0001002H1K_unpack_sdu` function which appears to be responsible for parsing various types of packets using different parsers. This includes logic for parsing Circuit Mode Control Entity (CMCE) (`z0001002H1L_handle_CMCE`), Mobile Link Entity (MLE) (`z0001002H1M_handle_MLE`) and Mobility Management (MM) (`z0001002H1N_handle_CAM`). After discussing this finding internally, we decided to focus on the packet parsing for the MM since these parsing routines are nontrivial and can be reliably be triggered over-the-air using our TETRA setup and our MS. We first try to find `z0001002H1K_unpack_sdu` in the MBTS BS since `z0001002H1K_unpack_sdu` only has been found in the MTS2 BS. For this, we use the method as described in Section 5.1.1. By using static strings in the decompiled code of the child functions of `z0001002H1K_unpack_sdu`, we managed to find several packet parsing functions in our MBTS BS. The packet parsing functions are listed in Table 5.

In our MBTS BS, all of the above functions are a direct child function of `yPAD_z0001002H_DataTransferEntityType`. This can be explained by the higher compiler optimization used for our MBTS BS as explained in Section 2.7.

All of the functions mentioned in Table 5 may be good targets to fuzz, since they are directly related to packet parsing of incoming data from the Air Interface. However, we focus on three specific target functions since we can reliably trigger the corresponding parsing routines of these three targets (further explained in Section 5.5.2). Using these functions as the target, we can continue creating the memory dump of our MBTS BS to start executing these functions in the emulator.

5.2 Writing the memory dump module

In order to dump the memory regions and registers of our MBTS BS, we need to have a memory dump module compatible with Fiddle and the modified version of Cydia Substrate (see Section 2.3). This memory dump module needs to do the following:

Symbol name of function
z0001002H1Q_handle_U_INFO
z0001002H1P_handle_U_SDS_DATA
z0001002H1O_handle_U_SDS_STATUS
z0001002H1U_handle_U_DEMAND
z0001002H20_handle_U_LOCATION_UPDATE_DEMAND
z0001002H1X_handle_U_ITSI_DETACH
z0001002H1W_handle_U_DISABLE_STATUS
z0001002H1S_handle_U_DISCONNECT
z0001002H1V_handle_U_PREPARE
z0001002H1R_handle_U_CONNECT
z0001002H4L_u_authentication_response_unpack
z0001002H4O_u_authentication_demand_unpack
z0001002H4M_u_authentication_result_unpack
z0001002H4N_u_authentication_reject_unpack

Table 5: Interesting target functions in the MM, CMCE and MLE found in the SC firmware of our MBTS BS.

1. Dump the CPU registers (Section 5.2.1)
2. Dump all the memory regions (Section 5.2.2)

Because we use the modified version of Cydia Substrate, we can easily hook specific addresses and let it execute another function instead. In this case, we let it execute the `dump_memory` function, which we implement in the memory dump module.

5.2.1 Dumping the CPU registers

The first thing we need to dump before dumping anything else is the CPU registers. This is because the CPU registers quickly change between function calls. We do not want to lose any information stored in the CPU registers, since then emulation will not be accurate and therefore may fail.

We have chosen to dump the registers by writing C-code and not writing assembly. While it is possible to write this in pure assembly (and having full control over the executed code), we think it is easier to use the C-code equivalent since it is easier to understand and use within the memory dump module. The GNU Compiler Collection (gcc) supports accessing registers in C-code with the following syntax: `register int * foo asm ("r12");` [26]. This syntax supports dumping all the general purpose registers 0 up to (and including) 31, `sp` and `ctr`. There are some registers which are unsupported, such as `cr`, `lr` and `xer`. For these registers, we use the `mfcr 4`, `mflr 4` and `mfxer 4` instructions to move the contents of these registers to the general purpose register 4, which we can dump via the gcc-method.

Next, we need to print the contents of the registers to the telnet session so we can extract them. Because we use Fiddle and the modified version of Cydia Substrate, this is straightforward: the telnet connection object is passed to the module during initialization. This telnet connection object is required as the first argument in the `mmi_flow_control_printf` function. The second argument of the `mmi_flow_control_printf` is a format string, and their parameters, just like the arguments of a normal `printf` function in C. So we can write the content of the registers as hex values in a string using the `%08x` format-specifier. This string is automatically sent over the telnet connection,

which we can listen to.

5.2.2 Dumping the memory regions

We also need to have a memory dump of all the memory regions in our MBTS BS. Somewhere in this memory dump will be the packet that is being parsed, as well as the assembly code parsing the packet. Dumping the memory is straightforward using Fiddle and the modified version of Cydia Substrate and similar to dumping the CPU registers: we iterate over all the memory addresses in the existing memory regions, and we dump the content of the memory address to the telnet session using `mmi_flow_control_printf` and the `%02X` format specifier. This will print the memory contents as hex values to the telnet session, which we can later decode and load into Unicorn.

We need to know the exact starting addresses of the memory regions and the sizes of the memory regions in order to dump the memory. The memory regions are initialized at boot time of our MBTS BS. During initialization, some special registers called 'BAT registers' are set which maintain address permissions for the memory regions [13]. During development of Tinker (see Section 2.3), the BAT register setup in our MBTS BS has been reverse engineered and we know that the following memory regions exists with the following permissions:

1. Start: 0x0, end: 0x2000000, read-only
2. Start: 0x2000000, end: 0x4000000, read/write
3. Start: 0x4200000, end: 0x5200000, read/write
4. Start: 0xfe000000, end: 0xffffffff, read/write (used for I/O)

We can use this information to configure the memory dump module to dump regions (1), (2) and (3). Since (4) is used for I/O, it is not necessary to dump it because we do not have any actual hardware connected in the Unicorn emulator.

We have chosen to dump the memory contents as hex and not as raw bytes, since dumping the raw bytes over telnet did not work. When doing so, only a few kilobytes will be dumped. After that, we get the message that the telnet connection is terminated by the remote and the memory dump stops. We do not know exactly why this happens, but it may be related to the telnet escape sequence. This issue is blocking the memory dump. Since dumping hex-characters only is working fine, we dump the memory using hex and convert it afterwards to raw bytes.

The telnet connection is not very fast. And since we dump the memory in hex-format, every byte will be two bytes for dumping (since hex doubles the output bytes). This results in a full memory dump taking about 5 hours. Fortunately, since the first memory region is read-only, we only have to dump the first memory region once. This saves about 1 hour for other memory dumps.

5.2.3 Defeating the watchdog

As stated in Section 2.4, our MBTS BS is protected by a watchdog. Since the memory module takes more than 30 seconds to run, the watchdog will kick in and will reset our MBTS BS, thus killing the memory dump. We have two options for solving this problem:

1. Patching the Site Controller (SC) firmware to never start the watchdog task.
2. Feeding the watchdog during the memory dump procedure so it will not reset our MBTS BS.

In order to implement method (1), we have to perform a firmware modification of the SC firmware. This can have side effects since we change the initialization of the SC firmware. We also thought that method (2) would be easier since it does not require additional firmware modifications, and can be implemented in the memory dump module itself. Therefore, we have chosen for method (2).

In order to replicate the behavior of feeding the watchdog, we need to first reverse engineer the firmware to find the code which is responsible for feeding the watchdog. Finding this code was easy, since it is part of the initialization method after boot and the function starting the watchdog task includes the string `AcLaunchWdogTask`. Analyzing the watchdog task code reveals that there are four important functions:

1. The watchdog initializer
2. The watchdog trigger
3. The first watchdog feeder
4. The second watchdog feeder

The watchdog initializer initializes a new watchdog object which can be used in the watchdog trigger function to trigger the watchdog check. There are also two feeder functions, which are called alternately with the constant value `0x1000` as the first parameter. With this knowledge, we also initialize a watchdog object in the memory dump module just after dumping the CPU registers. While dumping the memory regions, we feed the watchdog by alternately calling the feeder functions and then calling the trigger function. We do this after every 100 bytes we dump to the telnet connection. This will make sure that the watchdog will always be triggered within 30 seconds, which prevents our MBTS BS from being restarted.

5.3 Creating the memory dump

Compiling the memory dump module gives an ELF binary, which can be loaded into the Base Station (BS). This loading is done using Tinker, Fiddle and Cydia Substrate (see Section 2.3). The result is that the firmware is automatically patched on runtime to execute the dumping code when the hooked functions are executed. The moment of hooking the packet parsing is visualized in Figure 14. Now we can listen to the telnet output and redirect that to a file to store the memory dump on our external device.

Since we were unsure which function would be triggered when a Mobile Station (MS) is connected to the BS, we decided to hook all target functions described in Table 5 to greatly increase our chance of triggering at least one target function. Activating the MS and letting it start connecting to the BS almost immediately triggered the memory dumping. Upon inspection of the CPU registers, the hook placed on `z0001002H20_handle_U_LOCATION_UPDATE_DEMAND` has issued the dump since the return address of the hooked function points to an instruction after an instruction calling the `z0001002H20_handle_U_LOCATION_UPDATE_DEMAND` function. We decided that `z0001002H20_handle_U_LOCATION_UPDATE_DEMAND` is also a good fuzzing case since the packet of U-LOCATION UPDATE DEMAND is a packet with a considerable number of data fields, as can be seen in Figure 15.

5.4 Emulating the memory dump

Now the goal is to emulate the packet parsing execution of the Site Controller (SC) firmware in Unicorn using the captured memory dumps (in short 'emulating the memory

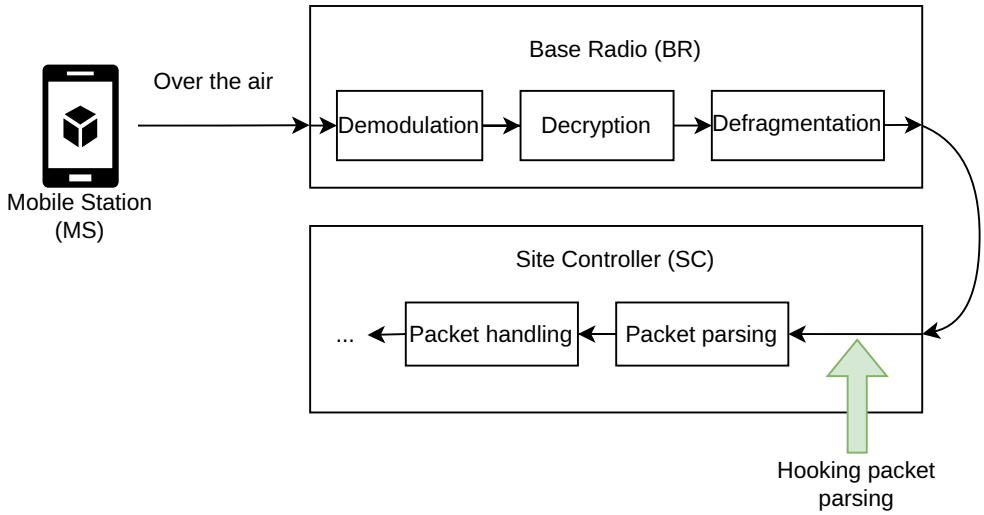

Figure 14: Visualization of the moment of hooking the packet parsing.

16.9.3.4 U-LOCATION UPDATE DEMAND

- Message: U-LOCATION UPDATE DEMAND
- Response to: -/D-LOCATION UPDATE COMMAND
- Response expected: D-LOCATION UPDATE ACCEPT/D-LOCATION UPDATE REJECT
- Short description: The MS sends this message to the infrastructure to request update of its location registration.

Table 16.18: U-LOCATION UPDATE DEMAND PDU contents

Information element	Length	Type	C/O/M	Remark
PDU type	4	1	M	U-LOCATION UPDATE DEMAND
Location update type	3	1	M	
Request to append LA	1	1	M	
Cipher control	1	1	M	
Ciphering parameters	10		C	See note 1
Class of MS	24	2	O	See note 2
Energy saving mode	3	2	O	
LA information		2	O	
SSI	24	2	O	ISSI of the MS
Address extension	24	2	O	MNI of the MS
Group identity location demand		3	O	
Group report response		3	O	
Authentication uplink		3	O	See ETSI EN 300 392-7 [8]
Extended capabilities		3	O	See note 2
Proprietary		3	O	

NOTE 1: Information element "Ciphering parameters" is not present if "Cipher control" is set to "0", "ciphering off". Information element "ciphering parameters" is present if "Cipher control" is set to "1", "ciphering on".

NOTE 2: If the "class of MS" or the "extended capabilities" element is not included in this PDU and the SwMI needs to receive either of those information elements, the SwMI may accept the registration request and then send the MS a D-LOCATION UPDATE COMMAND PDU. (This could happen if, e.g. the SwMI has no record of a previous successful registration by the MS or if the SwMI has any uncertainty about the type of cell to which the MS is registering.)

Figure 15: U-LOCATION UPDATE DEMAND packet content fields as defined on page 270 in the TETRA protocol standard [8].

dumps'). Before we can load the memory dump files into Unicorn for emulating it, we have to parse the memory dump and do the following:

1. We split the CPU register dump and memory region dumps into multiple files to easily load the memory regions at different memory offsets in Unicorn.
2. We hex-decode the memory region dumps.

A small Python script does the above two tasks.

Given the memory dump files, we can now start emulating the memory dump in Unicorn. As a basis for our emulation, we use the code written for the fuzzgoat emulation and fuzzing. This makes it trivial to load the CPU registers and memory region dumps into Unicorn.

Starting the emulation with this configuration will fail because of the following issues:

1. Registers `r0` (value of the LR register, a PowerPC specific register) and `r1` (stack pointer) are incorrect due to the function-prologue of the memory dump module.
2. The program counter is incorrect, since it points somewhere in the memory dump module.
3. The emulator will execute the memory dump module again since the start of the `z0001002H20_handle_U_LOCATION_UPDATE_DEMAND` function is still patched to execute the memory dump module instead.
4. The emulated code will have code paths which trigger a syscall.

In Section 5.4.1 we discuss how to resolve issue (1) and (2). In Section 5.4.2 we discuss how to resolve issue (3) and (4).

5.4.1 Patching the registers

Since we have decided to write the memory dump module in C and not in assembly, we had to use a compiler to convert it to assembly. For a function call, the compiler will put some 'prologue' assembly before actually executing the assembly inside the function. This prologue assembly will initialize the stack frame for the function and it will remember the return address. This changes some of the register values *before* we have dumped the CPU registers.

The stack pointer in register `r1` can be corrected by reverting the subtraction done by the prologue assembly. By reverse engineering our own ELF binary of the memory dump module, we can see that the memory dump function starts with the following instruction: `stwu r1,-208(r1)`. To undo this instruction, we can add 208 to the value of the `r1`. Now the stack is properly setup for the `z0001002H20_handle_U_LOCATION_UPDATE_DEMAND` function.

Register `r0` is lost in the process of dumping the CPU registers. The prologue assembly contains the following instruction before storing the value of `r0` on the stack: `mflr r0`. This instruction will move the value in the Link Register to the `r0` register. This will overwrite the value in the `r0` register, and is therefore lost. However, `r0` is a scratch-register, so it is used to store temporary values. Therefore, it should not be an issue since `r0` cannot contain any parameter values for the function. Furthermore, the value of `r0` in the CPU register dump will actually be the value of the LR-register, so we can use that value instead. The program counter can just manually be set to the start of the `z0001002H20_handle_U_LOCATION_UPDATE_DEMAND` function.

5.4.2 Patching the memory

Since we have dumped the memory regions with our patch applied to the code, in the emulator the `z0001002H20_handle_U_LOCATION_UPDATE_DEMAND` is also patched to execute the memory dump module instead of the actual function. Of course, this is not what we want to fuzz. So we have to revert the patch. This can easily be done by reading from the original firmware the content of the addresses that are patched. The read bytes from the original firmware will correspond to the original assembly before it was patched by the modified version of Cydia Substrate. During the loading of the memory regions in Unicorn, we can overwrite the bytes in the emulator with the original bytes. This will revert the patched assembly in the emulator.

When we now start the emulator, we will hit a code path that will actually only run a small part of the `z0001002H20_handle_U_LOCATION_UPDATE_DEMAND` function. This is because the executed code path contains other functions, which execute syscalls. Syscalls are not supported in Unicorn, unless a custom implementation is provided. Therefore, the execution of functions depending on syscalls will probably fail. For example, the `xPrdCall` function will allocate some memory using a syscall. However, the syscall does nothing and therefore the function will fail. This failing is detected in the `xPrdCall` itself and will result in a code path terminating the execution. We can solve this problem in two ways:

1. We emulate the syscalls in Unicorn by implementing the required syscalls.
2. We skip the functions/code that require syscalls and hope that the important code still works.

Our approach is to prefer method (1). It is, however, a time consuming approach since the syscall numbers are not documented anywhere. However, this will make the emulation as close as possible to the real execution of the code. Furthermore, there are not many syscalls that will be executed during the fuzzing and that will reduce the amount of work for reimplementing the syscalls in Unicorn. If we are 100% sure that a function is not useful during emulation and fuzzing, then we may choose method (2) if reimplementing the syscall is prohibitively hard. For method (2), it is straightforward in Unicorn to hook a specific function and set the program counter to the end of that function, thus skipping the function completely.

After implementing the syscalls, we verified that the firmware runs the packet parsing without any issues within Unicorn.

5.5 Fuzzing in the Unicorn emulator

5.5.1 Preparing the fuzzing

At this point, we have set up almost everything to start fuzzing the packet parsing code of the target `z0001002H20_handle_U_LOCATION_UPDATE_DEMAND`. We still need to perform two steps:

1. Find the packet in memory and dump it to a file.
2. Write code to overwrite the packet with the fuzzing case.

(1) is done by inspecting the decompiled code of the packet parsing and determine where the buffer is read. If we have found a location in the code where the reading of the buffer is done, we set a breakpoint in Unicorn just before the reading of the buffer. After executing Unicorn with this breakpoint, we inspect the pointer that is used to locate the start of the buffer. By reverse engineering, we discovered that the

data structure of the buffer includes a size, a cursor offset and a 'bits left to read' value. The packet that we are going to fuzz will then become the bits from the cursor offset until the cursor offset plus the 'bits left to be read'. We call this range the target buffer. We then dump the target buffer to a file so it can be used as the initial seed for AFL++.

For (2), we write some code to write the bytes of the fuzz case over the target buffer. Moreover, since the buffer can never hold more than 200 bytes, we implement the code to write the bytes of the fuzz case in such a way that it allows the fuzz case to reach the maximum packet length of 200 bytes. We do this by computing the new value for 'bits left to read' when the packet is bigger or smaller than the original fuzz case. This updated 'bits left to read' value is then written to the correct address in memory, so the packet parsing logic will use it correctly.

We fuzz with only a single persistent round. This means that after one fuzz case, Unicorn will fork the process again to run a new fuzz case with. This is necessary since the CPU registers and the memory are changed after one round of fuzzing. If we execute a new fuzz case with these changes, the emulation will be wrong. Therefore, the state is reset to the state before the fuzz case was executed by forking the process again.

As the last step before running the fuzzer, we wanted to verify the crash-detection of AFL++ in order to prevent an implementation mistake. We verified our fuzzer by triggering an `abort()` in the Unicorn harness when executing an unknown/not implemented syscall. Some execution flows pertaining to exceptions did have a unique syscall and AFL++ was able to detect this syscall as a 'crash', thus we have verified that AFL++ is able to properly detect crashes with our setup.

5.5.2 The three fuzzing targets

We have decided to fuzz the following three targets:

1. U-LOCATION-UPDATE-DEMAND (symbol `z0001002H20_handle_U_LOCATION_UPDATE_DEMAND`)
2. U-SDS-DATA (symbol `z0001002H1P_handle_U_SDS_DATA`)
3. U-SDS-STATUS (symbol `z0001002H1O_handle_U_SDS_STATUS`)

We have chosen for these three targets since they are all straightforward to trigger in our Base Station (BS) setup. All these three targets also parse big data frames, as can be seen in Figure 15, Figure 16, and Figure 17. Moreover, U-LOCATION-UPDATE-DEMAND and U-SDS-STATUS both are complex messages. This has been concluded based on the large size of the decompiled code of U-LOCATION-UPDATE-DEMAND and U-SDS-STATUS handlers: the decompiled code of the parsing function of U-LOCATION-UPDATE-DEMAND is 343 lines (excluding lines of code of other parsing functions within U-LOCATION-UPDATE-DEMAND) and the decompiled code of the parsing function of U-SDS-STATUS is 382 lines. U-SDS-DATA has been chosen because it can contain a lot of user defined data, since an SDS is similar to an SMS message which can be send to other Mobile Stations (MSs). Because of this user defined data, it may increase the probability of finding a bug in the code. The U-SDS-STATUS packet can be seen in Figure 17, which is documented as 'U-STATUS' in the TETRA protocol standard.

All three fuzzing targets have been fuzzed in parallel with five fuzzing instances using AFL++ support for parallel fuzzing. There is one *parent* and four *children*. This together will utilize five CPU-cores for fuzzing. As described in the AFL++ documentation [23], the parent will still perform deterministic checks while the childs are limited

14.7.2.8 U-SDS-DATA

- Message: U-SDS-DATA
- Response to: -
- Response expected: -
- Short description: This PDU shall be for sending user defined SDS data.

Table 14.28: U-SDS-DATA PDU contents

Information element	Length	Type	Owner	C/O/M	Remark
PDU Type	5	1	SDS	M	U-SDS-DATA
Area selection	4	1	SS	M	See note 1
Called party type identifier	2	1	SDS	M	Short/SSI/TSI
Called party short number address	8		SDS	C	See note 2
Called party SSI	24		SDS	C	See note 2
Called party extension	24		SDS	C	See note 2
Short data type identifier	2	1	SDS	M	See note 4
User defined data-1	16		SDS	C	See notes 3 and 4
User defined data-2	32		SDS	C	See notes 3 and 4
User defined data-3	64		SDS	C	See notes 3 and 4
Length indicator	11		SDS	C	See note 3
User defined data-4			SDS	C	See notes 3 and 5
External subscriber number		3	SDS	O	
DM-MS address		3	CC	O	

NOTE 1: This information element is used by SS-AS, refer to ETSI EN 300 392-12-8 [14].
NOTE 2: Shall be conditional on the value of Called Party Type Identifier (CPTI):
- CPTI = 0; Called Party SNA; refer to ETSI ETS 300 392-12-7 [13];
- CPTI = 1; Called Party SSI;
- CPTI = 2; Called Party SSI + Called Party Extension.

NOTE 3: Shall be conditional on the value of Short Data Type Identifier (SDTI):
- SDTI = 0; User Defined Data-1;
- SDTI = 1; User Defined Data-2;
- SDTI = 2; User Defined Data-3;
- SDTI = 3; Length indicator + User Defined Data-4.

NOTE 4: Any combination of address and user defined data type is allowed. However, the intention is to fit TNSDS-UNITDATA request into one subslot when possible. It is recommended that always the shortest appropriate user defined data type is used. One subslot signalling is possible on a π/4-DQPSK or D8PSK channel by using one of the following combinations:
- Short Number Address and User Defined Data 1 or 2;
- Short Subscriber Identity and User Defined Data 1.

NOTE 5: The length of user defined data 4 is between 0 and 2 047 bits. However, if the basic link is to be used, then the longest recommended length of the user defined data 4 is 1 017 bits while using Short Subscriber Identity and FCS on a π/4-DQPSK channel (see clause 23.4.2.1, see note 2).
Clause 23.4.2.1 also indicates the longest recommended size of TM-SDU for fragmentation for other modulations and channel bandwidths; the longest recommended length of the user defined data 4 using the basic link while using Short Subscriber Identity and FCS may be obtained by subtracting 89 bits from the longest recommended size of TM-SDU (subject to the maximum length of 2 047 bits for user defined data 4). In order to avoid needing to know the modulation level to be used to send the SDS data on a D8PSK or QAM channel, the longest recommended length of the user defined data 4 using the basic link on a D8PSK channel may be regarded as being the value for π/4-DQPSK modulation, and the longest recommended length of the user defined data 4 using the basic link on a QAM channel may be regarded as being the value for 4-QAM rate = ½ for the appropriate channel bandwidth.

Figure 16: U-SDS-DATA packet content fields as defined on page 193 in the TETRA protocol standard [8].

to random tweaks. The parent will also synchronize the results of all instances, so a case discovered by one instance can be used in other instances in subsequent fuzzing efforts.

5.5.3 Fuzzing U-LOCATION-UPDATE-DEMAND

The fuzzer ran for a total of 428.8 million executions in one day and around three hours of fuzzing. The fuzzer found a total of zero crashes and zero hangs (executions that took longer than one second). The AFL++ output can be found in Appendix A.

14.7.2.7 U-STATUS

Message: U-STATUS

Response to: -

Response expected: -

Short description: This PDU shall be used for sending a pre-coded status message.

Table 83: U-STATUS PDU contents

Information element	Length	Type	Owner	C/O/M	Remark
PDU Type	5	1	SDS	M	
Area selection	4	1	SS	M	note 1
Called party type identifier	2	1	SDS	M	Short/SSI/TSI
Called party short number address	8		SDS	C	note 2
Called party SSI	24		SDS	C	note 2
Called party extension	24		SDS	C	note 2
Pre-coded status	16	1	SDS	M	
External subscriber number		3	SDS	O	
DM-MS address		3	CC	O	
NOTE 1: This information element is used by SS-AS, refer to EN 300 392-12-8 [33]. NOTE 2: Shall be conditional on the value of Called Party Type Identifier (CPTI): CPTI = 0; Called Party SNA; refer to ETS 300 392-10-7 [41] CPTI = 1; Called Party SSI; CPTI = 2; Called Party SSI + Called Party Extension.					

Figure 17: SDS-STATUS packet content fields as defined on page 192 in the TETRA protocol standard [8].

In order for the fuzzer to work properly, we had to implement syscall number 17 in the Unicorn harness. Syscall 17 is a syscall to allocate more memory. It is used by functions to pass parameters to other functions. Moreover, we have ignored syscall numbers 2 and 18. Syscall 2 is a syscall used to free memory. However, the allocation mechanism implemented for syscall number 17 in the Unicorn harness does not try to reduce fragmentation of memory. Therefore, it is also not necessary to actually free the allocated memory, since we never allocate memory twice on the same address. Syscall 18 is used for translating a 'task name' to a 'task id'. Upon inspecting the code invoking this syscall, it appeared that ignoring the syscall will result in the correct code path being taken. So no exception is triggered when nothing happens when syscall 18 is executed.

5.5.4 Fuzzing U-SDS-DATA

For fuzzing U-SDS-DATA, we performed the following steps:

1. We modify the memory dump module created in Section 5.2 to hook the U-SDS-DATA handler instead of the U-LOCATION-UPDATE-DEMAND handler.
2. We trigger the memory dump by sending an SDS message.
3. We load the new memory dump in Unicorn by writing another Unicorn harness to parse the new memory dump.
4. We dump the U-SDS-DATA packet in memory so it can be used as the seed for AFL++.
5. We run the fuzzer with the new Unicorn harness to fuzz U-SDS-DATA.

Two syscalls have to be implemented in order to support U-SDS-DATA. Syscall 96 is used in error paths to log strings to a log file. This logging is implemented in the kernel

and therefore it uses a syscall. Syscall 96 does not expect any output from the kernel and therefore we can safely ignore the syscall. Syscall -1 is also used in some error paths. It is not clear what the exact purpose is for this syscall, but it may be related to resetting the Base Station (BS) or reporting errors to the kernel. Ignoring syscall -1 appears to work perfectly fine, and therefore we can continue fuzzing.

The packet parsing code for U-SDS-DATA is quite short. Therefore we stopped fuzzing after AFL++ did not discover any new execution paths after more than two hours, as shown in the 'last new find' timer in the AFL++ output. The fuzzer ran for a total of 39.7 million executions in around four hours and twenty minutes. The fuzzer found a total of zero crashes and zero hangs. The AFL++ output can be found in Appendix A.

5.5.5 Fuzzing U-SDS-STATUS

For fuzzing U-SDS-STATUS, we performed the following steps:

1. We modify the memory dump module created in Section 5.2 to hook U-SDS-STATUS instead of U-SDS-DATA.
2. We trigger the memory dump by sending an SDS status message.
3. We load the new memory dump in Unicorn by writing another Unicorn harness to parse the new memory dump.
4. We dump the U-SDS-STATUS packet in memory so it can be used as the seed for AFL++.
5. We run the fuzzer with the new Unicorn harness to fuzz U-SDS-STATUS.

We did not have to implement any new syscalls for U-SDS-STATUS since all required syscalls have already been implemented in the fuzzing steps for U-LOCATION-UPDATE-DEMAND (Section 5.5.3) and U-SDS-DATA (Section 5.5.4). The fuzzer ran for a total of 353.0 million executions in one day and around thirteen hours and twenty minutes. The fuzzer found a total of zero crashes and zero hangs. The AFL++ output can be found in Appendix A.

6 Manual Site Controller firmware analysis through reverse engineering

Besides fuzzing the packet parsing of the MBTS Base Station (BS) as described in Section 5, we have performed a brief manual security analysis of the firmware binary of the Site Controller (SC) in the remaining time available for this research. The goal of the manual firmware analysis is to research if it is possible to extract encryption keys from the MBTS BS. With the original firmware running on the MBTS BS, it should not be possible to easily extract any encryption keys since that would compromise the confidentiality of the communication on the TETRA network. In the research of Section 5, we assumed that the adversary does not have physical access to the BS. However, in order to interact with the SC firmware through the serial connection, the adversary has to have physical access to the BS. Therefore, in this section we assume that the adversary does have physical access to the serial port of the BS, but the adversary cannot perform hardware attacks to extract the encryption keys.

The goal of the manual firmware analysis is to extract any of the following encryption keys:

1. Static Cipher Key (SCK)
2. Common Cipher Key (CCK) or Derived Cipher Key (DCK)

As described in Section 2.1.1, access to the SCK would compromise the confidentiality of class 2 networks and access to the CCK or DCK would compromise the confidentiality of class 3 networks if the adversary has access to the proprietary algorithms used for key derivation.

6.1 Backdoor password for engineer access

It is possible to connect with the Site Controller (SC) firmware by the following methods:

1. Telnet connection
2. Serial port

Both connection methods will prompt the user for a username and password. No account registration is provided by the firmware of the SC.

We first reverse engineer the code in the SC firmware that implements the authentication mechanism. By using strings that are printed by the authentication prompt, the code for the authentication mechanism is discovered. We found that there are two privilege levels implemented:

1. Field
2. Engineer

The privilege level is determined by the password the user provides. When a custom password is set, it is set through a syscall. Only one password can be set per privilege level.

The authentication check is done by retrieving the field and engineer passwords through a syscall. Afterwards, it does a string comparison with the entered password and the obtained password from the syscall to check if the entered password is correct. If the password is correct, it will continue with the privilege level for which the password was correct. However, the authentication check also matches the entered password with two backdoor passwords: one backdoor password for field privileges and one backdoor

password for engineer privileges. Therefore, it is possible to authenticate using these backdoor passwords to obtain field or engineer privileges. Due to the nature of the findings, we have chosen to not publish the exact backdoor passwords in this thesis. It is not possible to override these backdoor passwords by setting a different password, since the user-set password as well as the backdoor password is accepted for authentication.

6.2 Extracting encryption keys using the debugger

In Section 6.2.1 we show that it is possible to dump memory by accessing the pROBE+ debugger. In Section 6.2.2 we discover the fixed addresses in memory where the encryption keys are stored. Section 6.2.3 describes the chain of the backdoor password, debugger access and fixed addresses in memory to dump the encryption keys of the Base Station (BS).

6.2.1 Accessing the pROBE+ debugger

With the backdoor passwords found in Section 6.1, we can authenticate with engineer access to the Site Controller (SC). We use engineer privileges and not field privileges, since engineer privileges are a superset of the field privileges.

With an authenticated CLI-session over the serial port with engineer privileges, we can list all the available commands by typing `.help`. One command is particularly interesting: `.crashdumptest` with description "Cause the TSC to crash and enter pROBE+". pROBE+ is the integrated debugger of the pSOSystem Real-Time Operating System. The `.crashduptest` will trigger an invalid operation, for example a read or write to unmapped memory, which will cause the SC firmware to crash. If the SC firmware crashes, it will switch to pROBE+ and will display a lot of debugging output in order to help the user troubleshoot the problem. At the end of the debugging output, a pROBE+ prompt is shown where the user can interact with the debugger. The time of interaction is limited since the watchdog will reset the Base Station (BS) within 30 seconds, as described in Section 2.4.

The pROBE+ prompt provides full access to the memory of the SC firmware and the memory of pSOSystem. Therefore, it should be possible to dump the memory range where the encryption keys are stored (in encrypted form). Dumping memory can be performed by entering the command `dm.l 00000000..ffffffff` where `00000000` and `ffffffff` is the chosen memory range to dump. However, dumping all the memory available as with the memory dump module described in Section 5.3 is not possible due to the time limit of the watchdog. We need to investigate where the encryption keys are stored within the SC firmware so we only have to dump a small part of the memory of the SC firmware.

6.2.2 Finding the encryption keys in memory

With engineer access to the Site Controller (SC) firmware, it is possible to display the configuration of the SC firmware using the `display config` command. The output of this command contains the 'KEKz Update 1' which displays the Random Seed for Over The Air Re-keying (RSO) and the Sealed Key Encryption Key (SKEK). As described in Section 2.1.2, the Key Encryption Key (KEK) is a Motorola specific key used to encrypt and decrypt stored encryption keys in the MBTS BS. Moreover, the output contains the 'SCK Update 1' which displays the stored Static Cipher Key (SCK) (encrypted with the KEK). However, we want to have the unencrypted SCK or Common Cipher Key (CCK).

We assume that the KEK and SCK are stored somewhere in memory close to the SKEK and stored SCK. We can use the values of the SKEK and the stored SCK as displayed by the `display config` command to search for the addresses storing these values in memory. To perform this search, we use the memory dumps created in Section 5.3. By searching for the SKEK and stored SCK values in the memory dump, we find various places where the exact value of the SKEK and stored SCK is stored. Because we are able to dynamically inspect the memory dump through GDB by using udbserver as described in Section 2.6, it is trivial to see that most of the results are values located on the stack. Since the stack is dependent on the state of the SC firmware, these addresses can change and are therefore not static. A different Base Station (BS) could have different addresses containing the SKEK and stored SCK values on the stack. Therefore, these addresses are not reliable and thus we ignore the values on the stack.

Only one result remains where both the SKEK and the stored SCK are stored close to each other in memory that is not located on the stack: the memory range from 0x20f6600 to 0x20f67ff. We can find the code that uses these addresses by finding references in the SC firmware to addresses within this memory range. The code that uses the information stored within this memory range is already partially reverse engineered due to other research done by Midnight Blue. Therefore, we can easily identify that the memory in this memory range stores a global struct which contains all the information related to the encryption keys. This struct includes the RSO, SKEK and stored SCK (as already identified in the memory dump), and the (unsealed) KEK, stored CCK, (unsealed) CCK and the Static Cipher Key Version Number (SCK-VN).

The (unsealed) SCK is missing. After discussing this internally, we concluded that this is because air traffic decryption is actually done by the Base Radio (BR) firmware and not the SC firmware. Therefore, the stored SCK is never decrypted to the SCK using the KEK. However, the stored SCK can be decrypted by applying the TA32 algorithm over the stored SCK with the KEK and SCK-VN, which yields the SCK.

6.2.3 Extracting the encryption keys from memory

With the backdoor password as found in Section 6.1, and the access to the pROBE+ debugger as described in Section 6.2.1, the adversary can dump the encryption keys from the memory of the Site Controller (SC) firmware using the memory range found in Section 6.2.2. The adversary can extract the Key Encryption Key (KEK), Static Cipher Key (SCK) and Common Cipher Key (CCK) from the Site Controller (SC) firmware by performing the following six steps:

1. Connect to the serial port of the SC by having physical access to the Base Station (BS).
2. Authenticate as an engineer to the SC firmware by entering a username (for example `engineer`) and the associated backdoor password.
3. Enter the following command to trigger an invalid read, therefore crash and enter pROBE+: `.crashdumptest -read`
4. Wait for the debug output to finish and enter the following command in the pROBE+ prompt to dump the correct memory region: `dm.l 210bd00..210bfff`
5. The output of step (4) will expose the Random Seed for Over The Air Re-keying (RSO), Sealed Key Encryption Key (SKEK), Key Encryption Key (KEK), stored Static Cipher Key (SCK), stored Common Cipher Key (CCK), Static Cipher Key Version Number (SCK-VN) and CCK.

6. Apply the TA32 algorithm over the stored SCK with the required parameters to obtain the SCK.

Now the adversary has access to the SCK and CCK by having physical access to the BS and not having to modify the firmware of the SC.

7 Future Work

In this chapter, we write down all research ideas for future work on the security analysis of the Base Station (BS). The future work ideas are ordered from most to least priority in our opinion: fuzzing the Base Radio (BR) firmware has the highest priority (Section 7.1) for future research while fuzzing more target functions (Section 7.5) has the least priority for future research.

7.1 Fuzzing the Base Radio firmware

In Section 5, we have focussed on fuzzing the packet parsing code. The packet parsing code we focussed on was implemented in the Site Controller (SC) firmware. We have only very briefly reverse engineered the Base Radio (BR) firmware (see Section 5.1.2). However, the BR firmware could also contain (memory corruption) bugs. Therefore, in future research it could be possible to change the research scope from the SC firmware to the BR firmware.

More specifically, the BR firmware implements a queue for incoming packets. These packets will then be defragmentated and decrypted before it is passed to the SC firmware. The defragmentation and decryption looks like it has been manually implemented (and thus it is not automatically generated code) since it does not contain any of the code-patterns we discuss in Section 8.1.1. This is an interesting target to fuzz since it is also related to the Air Interface of the Base Station (BS), and the Air Interface has the highest impact when vulnerabilities are found since the adversary does not need physical access to the BS.

7.2 Reverse engineering the Key Variable Loader protocol

As described in Section 2.1.2, an Infrastructure Key (Ki) is used to encrypt the Key Encryption Key (KEK) and is used for encrypting traffic related to infrastructure management. The Ki can be seen as a master key for the Base Station (BS). The Ki can be updated using a Key Variable Loader (KVL) hardware device. A KVL uses an undocumented protocol to update the Ki in the MBTS BS over the serial port. It is expected that the Ki can only be updated by using a KVL. Moreover, it should not be possible to extract the Ki from the MBTS BS.

For future work on the security of the encryption keys used in the TETRA protocol and in the BS, research can be done for this undocumented protocol to update the Ki in the MBTS BS. This would allow a user to update the Ki without the need for a KVL which is difficult to obtain and expensive to buy. Besides changing the Ki through an update without using a KVL, this research could also include investigating whether the Ki can also be extracted from the MBTS BS. Extracting the Ki would allow an adversary to decrypt network management traffic, since the Ki is used for encrypting traffic related to infrastructure management.

7.3 Improving emulation of pSOSystem

We have only fuzzed the packet parsing code in Section 5.5. We had to select a specific target function from Table 5 on page 33 and we fuzzed the packet parsing code in the target function from the beginning until the end, or we have stopped fuzzing earlier due to interaction with the pSOSystem RTOS. The fuzzing can be improved by implementing the syscall routine in the Unicorn harness. The syscall routine is triggered when a syscall is executed by the firmware. The syscall routine changes the execution to kernel-land and executes the corresponding syscall function, based on the syscall number. After the

syscall handler is done, the syscall routine will change back the execution to user-land and will continue the execution of the firmware. With the syscall routine, it is possible to fuzz the packet parsing state machine from the beginning until the end, including the IPC done by pSOSystem. This will result in much more accurate results since the execution is more realistic, and it will result in much more code that is fuzzed since the whole state machine can then be fuzzed.

This syscall routine can for example be implemented by making it possible to change to kernel-land when a syscall is executed. The CPU registers should be saved before switching to kernel-land and be restored after the code located in kernel-space is done. Furthermore, the user-land stack should be swapped to a kernel-land stack and the user-land stack should be restored after swapping back to user-land. The syscall implementation should also be resolved by using the interrupt vector table of the pSOSystem kernel. The interrupt vector table is a table which maps the syscall number to the corresponding syscall implementation. Since the kernel and firmware are all in one memory region, the kernel and thus the interrupt vector table is already in the memory dump. By reverse engineering, the interrupt vector table can be found in the memory dump and can be connected to the syscall routine. Implementing the syscall routine in the Unicorn harness will also reduce the implementation time of the syscalls in the Unicorn harness, since it is not necessary anymore to implement each syscall individually in the Unicorn harness.

7.4 Discovering more targets in the Site Controller firmware

Table 5 on page 33 contains all the target functions we have found for the packet parsing code during reverse engineering. This table could be extended by reverse engineering different functionalities within the Site Controller (SC) firmware, which are also related to the Air Interface of the Base Station. For example, we encountered similar state machines in the SC firmware as the packet parsing state machine we found. The other state machines can also be reverse engineered and fuzzed. However, it is important to keep in mind that since these state machines are similar to the state machine we found, these other state machines are automatically generated code too (see Section 8.1.1). Therefore, it is very unlikely to uncover any issues related to memory corruption in the packet parsing code.

7.5 Fuzzing more target functions in the Site Controller firmware

In this thesis, we have only fuzzed three target functions as described in Section 5.5.2. The research could be extended by fuzzing more target functions as shown in Table 5 on page 33. Even less interesting targets can be fuzzed, such as targets that where only a few bits of data can be fuzzed. Moreover, if another Mobile Station (MS) would be used (which was not available in this research), it is possible to setup a call between the two MSs. The packet parsing code for the packets related to calling can then all be fuzzed. However, the probability of finding issues related to memory corruption is very low, since we conclude in Section 8.1.1 that the packet parsing code is automatically generated.

8 Conclusions

In this section we reflect on the decisions, method and results of this thesis. This includes summarizing and reflecting on various decisions and assumptions throughout our research. In Section 8.1 we discuss the results of the fuzzing done in this thesis. We also conclude that there are no vulnerabilities related to memory corruption found in the packet parsing code. This is due to the packet parsing code being automatically generated, as explained in Section 8.1.1. The limitations of the fuzzing done in this thesis are discussed in Section 8.1.2. In Section 8.2, we discuss the insecure design of the Site Controller (SC) firmware as researched in the manual firmware analysis as described in Section 6. Moreover, in Section 8.3 we reflect on the usefulness of earlier research done for the TETRA Base Station (BS). We also reflect in Section 8.4 on the decisions made in Section 4.1.2 related to the emulation method with Unicorn. Furthermore, we shortly discuss in Section 8.5 the effectiveness of our fuzzing efforts in this thesis.

8.1 The fuzzing results

In this thesis, we have fuzzed the following packet parsing code in the Site Controller firmware:

1. U-LOCATION-UPDATE-DEMAND (symbol `z0001002H20_handle_U_LOCATION_UPDATE_DEMAND`)
2. U-SDS-DATA (symbol `z0001002H1P_handle_U_SDS_DATA`)
3. U-SDS-STATUS (symbol `z0001002H1O_handle_U_SDS_STATUS`)

(1) is responsible for parsing the 'U_LOCATION_UPDATE_DEMAND' packet. The U_LOCATION_UPDATE_DEMAND packet is for example send when the Mobile Station connects to the Base Station after a (re)boot of the Mobile Station. (2) is responsible for parsing the data of an SDS message. An SDS message can be seen as a small text message, similar to SMS. (3) is responsible for parsing the SDS status packet. The SDS status packet is used to communicate a pre-coded status message. The packet parsing code of (1) and (3) are quite long and complex: the decompiled code of the parsing function of U-LOCATION-UPDATE-DEMAND is 343 lines (excluding lines of code of other parsing functions within U-LOCATION-UPDATE-DEMAND) and the decompiled code of the parsing function of U-SDS-STATUS is 382 lines. The packet parsing code of (2) is not very long and much simpler: the decompiled code of the parsing function of U-SDS-STATUS is 168 lines of code.

We did not find any issues related to memory corruption in (1), (2) and (3). We then found that the packet parsing code we fuzzed is automatically generated code (further discussed in Section 8.1.1), which greatly reduces the chance of finding issues related to memory corruption in the code.

8.1.1 Automatically generated code

After we have fuzzed the packet parsing implementation, we did some additional investigation on the characteristics of the code we have fuzzed since we have not found any memory corruption bugs in our research. One of the characteristics is the `xGetPrd` function always being called before calling another parsing function. A Google search for `xGetPrd` led us to a page hosted by the Michigan State University describing the use of "Telelogic Tau" [28]. From this page, we have found that the packet parsing code is automatically generated using an Specification and Description Language (SDL) called "Telelogic Tau" since the output of Telelogic Tau matches the characteristics of the

packet parsing code exactly. An SDL is a specification language used to describe and define the specification of systems. Using an SDL, it is possible automatically generate C-code based on the specification of the system.

From the resource hosted by the Michigan State University [28], we can confirm that the `yPAD_z0001002H_DataTransferEntityType` function from Section 5.1.3 is indeed a state machine. This state machine behaves as the 'packet parsing process', where it listens for incoming packets ('signals') from pSOSystem in order to parse the packets and send them to another process. We can also identify that all the target functions in Table 5 on page 33 contain automatically generated code, since they all call the `xGetPrd` function and `xAddPrdCall` function. This is because the resource hosted by the Michigan State University describes that before a SDL procedure is called, the functions `xGetPrd` and `xAddPrdCall` are called. Therefore, we can conclude that all our fuzzing efforts have been limited to auto-generated code. This explains why no issues related to memory corruption were found, since the automatically generated code follows the TETRA specification strictly and therefore is very unlikely to have any issues related to memory corruption. Only if the TETRA specification or the SDL contains any mistakes, the automatically generated code can contain mistakes.

8.1.2 Limitations of fuzzing

As shown in Table 5 on page 33, there are more targets to fuzz. We have decided to fuzz only some of the targets listed in Table 5 since the time for this research is limited and we focused on the most complex packet parsing code to increase the chance of finding a bug. Some of the packet parsing code is very short and not complex, which results in very few paths in the code and are therefore not the most interesting targets to fuzz. Moreover, there are many more parsing routines in the Base Radio firmware and Site Controller firmware (not listed in Table 5) which all can be fuzzed. However, the Base Radio firmware and Site Controller firmware are far too large to audit in the context of a thesis. Therefore, we have limited our fuzzing research scope to only some of the target functions listed in 5.

There were also functionalities we could not fuzz since we were limited to only one Mobile Station. For example, creating a call requires two Mobile Stations in order to trigger. Since we could not trigger these functionalities, we were unable to fuzz the corresponding packet parsing code.

For the packet parsing code we were able to fuzz, we were forced to fuzz with only one seed (see Section 2.5) for the initial fuzz case for AFL++. AFL++ may be able to find more interesting fuzz cases faster if we would be able to provide more seeds. However, this is not possible since every memory dump would only contain one packet that it is parsing at the moment of dumping the memory. Dumping the memory multiple times to collect multiple seeds would also not be possible, because the state of the memory dump would be different. For example, the registers of the CPU could contain different values depending on the previous packets that are parsed. In order to emulate as close as possible to the real Base Station, we can only use the packet that is being processed in the emulated memory dump as the seed.

We were not able to fuzz all the code from the target functions we have fuzzed. This is due to lots of pSOSystem interactions in some parts of the code. For example, some error paths will send a message to a different task in pSOSystem (see Section 2.2 for context). During fuzzing, we detected these faulty paths since they would cause a crash. This is because in the Unicorn harness it was programmed to signal a crash on a not-implemented syscall. In such a situation, we cancel the fuzzing and implement, stop at,

or ignore the syscall causing the crash. We cannot run the code that interacts with the pSOSystem kernel since we do not have access to the kernel functionalities of pSOSystem in Unicorn. This prevents us from end-to-end running the code and thus fuzzing the complete code flow triggered by a packet. Improving the emulation done by Unicorn to be able to fuzz the whole code flow is also mentioned in the future work in Section 7.3.

8.2 Insecure design of the Site Controller firmware compromises the encryption keys

By manually reverse engineering the Site Controller firmware, we have shown in Section 6 that it is possible to extract encryption keys used for secure communication between Mobile Stations and the Base Station. The extraction can be done by having physical access to the serial port of the MBTS Base Station (BS), without having to modify the Site Controller firmware and without having to perform any hardware attacks. This is due to the following insecure design of the Site Controller firmware:

1. A backdoor password that gives engineer access (highest privileges) to the Site Controller of the MBTS BS.
2. The possibility to trigger a crash by entering the `.crashdumptest -read` command.
3. A crash results in an interactive debugger session which gives full low-level access to the whole firmware, operating system code and data.
4. Encryption keys are stored in a predefined address range, which can be dumped using the interactive debugger session. This gives access to the Key Encryption Key, Sealed Static Cipher Key and Common Cipher Key.
5. The Sealed Static Cipher Key can be decrypted by an adversary using the proprietary TA32 algorithm to obtain the Static Cipher Key.

With knowledge of the Key Encryption Key and Static Cipher Key or Common Cipher Key, it is possible for an adversary to do the following:

- An adversary can send spoofed or decrypt intercepted encrypted one-to-one over-the-air communication in class 2 networks using the proprietary TB5 algorithm and the Key Stream Generator algorithm.
- An adversary can send spoofed or decrypt intercepted encrypted over-the-air group data transportation in class 2 or class 3 networks using the proprietary TB5 algorithm and the Key Stream Generator algorithm.

From this, we can conclude that it is trivial to compromise the confidentiality of one-to-one data transportation in a class 2 network, and group data transportation in class 2 or class 3 networks if the adversary has physical access to the serial port of the MBTS BS, and access to the specification of the proprietary TA32, TB5 and Key Stream Generator (KSG) algorithms. The TA32 algorithm is the inverse of the proprietary TA31 encryption algorithm. The TB5 and Key Stream Generator algorithms are used to generate the Key Stream Segment from the encryption keys and are used for encryption and decryption of data sent over the air interface. No firmware modifications are required in order to extract the encryption keys and no hardware attacks have to be performed.

However, having physical access to the MBTS BS already compromises the confidentiality and integrity of the network since the adversary can upload modified firmware to the MBTS BS via the telnet connection. No cryptographic signature checks are performed when an attempt is made to update the firmware. Therefore, even without the

interactive debugger, it is possible for an adversary to compromise the MBTS BS by uploading modified firmware that dumps encryption keys for example. The insecure design only makes it trivial to extract the encryption keys, since obtaining a firmware binary, patching the binary to include malicious code and uploading the patched binary to the MBTS BS is a time consuming and difficult process.

The extraction of the encryption keys is detectable, since entering the interactive debugger requires a crash which causes downtime of the MBTS BS. The MBTS BS may be unavailable for some minutes due to the debugger printing the output, entering the command to extract the memory and waiting for the watchdog to reset the MBTS BS. The boot sequence of the Site Controller firmware also take some minutes. Therefore, the unavailability of the MBTS BS may indicate that the encryption keys of the MBTS BS are being extracted. Authentication logs can be viewed to check if an adversary has been connected to the MBTS BS. However, these authentication logs can also be wiped through the interactive debugger when the storage address of these logs is known.

8.3 Reflecting on earlier research for the TETRA Base Station

As stated in Section 3.3, there is no previous work done on the research of the TETRA Base Station, except for the research done at the TU Darmstadt by Müller et al. [16]. That research does not include research on the Air Interface which we have fuzzed in this research. Moreover, the fuzzer that has been developed by Müller et al. was of no use in this research, since it did not have coverage guidance which is important to have to be effective, as stated in Section 2.5. The lack of earlier research resulted in a slow start, since all the reverse engineering still had to be done. Furthermore, the TETRA standard is big and complex. The TETRA standard had to be studied to understand the important parts of the protocol, since no other research has summarized this. This all has taken up a considerable portion of the available time.

The firmware analysis of the Site Controller firmware done by Müller et al. was useful. This gave us some insight in the naming convention of the symbols and the purposes of some functions (see Table 2, page 24). However, it had limited usefulness since their research was focused on the Base Radio firmware while our research focused on the Site Controller firmware. Moreover, we also had access to the firmware of the MTS2 BS, the Base Station used in the research of Müller et al. Even though the firmware of our MBTS Base Station (BS) and the firmware of their MTS2 BS contain a lot of differences (such as the lack of C++ code in our MBTS BS firmware), it was still useful for our research. The firmware of their MTS2 BS includes symbols, which greatly helped understanding the firmware during reverse engineering. Moreover, it allowed us to successfully find functions in the firmware of our MBTS BS, as described by Section 4.1.1. We think the approach for finding functions, which is explained in Section 5.1.1, greatly increased the efficiency of the research during the reverse engineering phase (see Section 5.1) and has achieved its goals.

8.4 Emulating the Site Controller of the MBTS Base Station in Unicorn

In Section 2.6 we have argued why we have used Unicorn as the emulator. We think this was a good decision since Unicorn was perfectly suitable for the research we have done in this thesis. Unicorn has a small but powerful API to write a harness from. With UnicornAFL [24], it was straightforward to attach AFL++ [10] to the harness. The feasibility study we did in Section 4.2 also greatly helped organizing the code of the Rust harness we wrote. Since we used a patched version of udbserver [4] in order to attach

GDB to the Unicorn emulator, it was very easy to inspect the state of the emulated code and debug any issues. The downside of Unicorn is that it lacks documentation and examples. The examples published on the AFL++ repository to use AFL++ with Unicorn[3] are small and not very useful. This made the start of using Unicorn more difficult.

We showed that it is possible to run the code of the MBTS BS in an emulator and fuzz the code without using the hardware of the MBTS BS during fuzzing. The research done by Müller et al. [16] suggested that this was not possible due to hardware dependencies of the firmware. We bypassed these hardware dependencies by creating a memory dump (see Section 5.3) and running the code in the emulator at the start of the packet parsing code, see also Section 4.1.2. Thus we skipped the booting sequence of the Site Controller firmware and therefore we did not encounter many hardware dependency issues. We had to implement some syscalls (see Section 5.4.2), but this was only a very small subset from all existing syscalls in pSOSystem.

8.5 The effectiveness of the fuzzing

We created a Unicorn harness (see Section 2.6.1) to fuzz the MBTS Base Station using AFL++. With our fuzzing method, we managed to fuzz some packet parsing code in the Site Controller firmware of the MBTS Base Station. We did this by creating a memory dump just before the Base Station starts running the packet parsing code (see Section 5.3). Then we ran the packet parsing code in Unicorn, an emulator, and we connected AFL++ to Unicorn to fuzz the packet parsing code using the emulator (see Section 5.4 and Section 5.5). This method is visualized in Figure 13 on page 29. With this method, we managed to get coverage guidance which made our fuzzing effective since AFL++ can determine when new execution paths have been discovered by directly inspecting the state of the Unicorn emulator. This effectiveness can be seen by the output of AFL++ (see Appendix A): on all targets it found many new inputs and new paths. From this we can conclude that we fuzzed the packet parsing code thoroughly.

Since we decided to write the Unicorn harness in Rust, the fuzzer ran at a good speed (between 400 to 1000 executions per second). It was also straightforward to implement the harness because the Rust compiler gives great suggestions and the Unicorn-Rust bindings are easy to understand and powerful to use.

8.6 Timetable

In Table 6 you can find an overview of an approximation of the amount of time spent on certain tasks.

Task (1) and (2) are about Section 5.1, which is the manual reverse engineering of the Base Radio firmware and Site Controller firmware. Task (3) relates to Section 4.2, which includes writing code for the Unicorn harness and experimenting with the UnicornAFL bindings. Task (4) is about Section 5.2, which includes writing a C-program, the memory dump module, compatible with Fiddle and Cydia Substrate (see Section 2.3) to dump all the memory regions and CPU registers of the Base Station. Task (5) is about running the memory dump module on the Base Station, which required almost no human interaction. We have done this multiple times where each run takes about 4 hours. There are also some failed attempts included in the approximate amount of hours spent. Task (6) is about emulating the execution of the Site Controller firmware with Unicorn by writing Rust-code for the Unicorn harness. This includes researching the syscalls in the Site

[3] `https://github.com/AFLplusplus/AFLplusplus/tree/stable/unicorn_mode/samples`

Controller firmware and implementing the syscalls in the Unicorn harness. Task (7) is about running the fuzzer with the Unicorn harness, to let AFL++ find bugs related to memory corruption. Task (7) does not require human interaction. Task (8) is about the research done in Section 6. Task (8) is done by manually reverse engineering the Site Controller firmware and having interaction with the Site Controller firmware.

Task	Approximate amount of hours spent
(1) Reverse engineering the Base Radio firmware	20 hours
(2) Reverse engineering the Site Controller firmware	40 hours
(3) Feasibility study with *fuzzgoat* (see Section 4.2)	10 hours
(4) Writing the memory dump module	10 hours
(5) Dumping the memory regions	20 hours
(6) Emulating the memory dump by writing a Unicorn harness	40 hours
(7) Fuzzing in the Unicorn emulator	80 hours
(8) Dumping the encryption keys	15 hours

Table 6: Timetable

References

[1] National Security Agency. *Ghidra*. URL: https://ghidra-sre.org/.

[2] S. Baskiyar and N. Meghanathan. "A Survey of Contemporary Real-time Operating Systems". In: vol. 29. 2. Informatica Slovenia, 2005, pp. 233–240. URL: https://www.eng.auburn.edu/~baskiyar/MyArticles/Survey-of-RTOS-Informatica.pdf.

[3] bet4it. *udbserver - Unicorn Emulator Debug Server*. 2022. URL: https://github.com/bet4it/udbserver (visited on 10/06/2022).

[4] bet4it and Jonathan Jagt. *udbserver - Unicorn Emulator Debug Server*. 2022. URL: https://github.com/JJ-8/udbserver (visited on 11/17/2022).

[5] GNU C Library Contributors. *The GNU C Library (glibc)*. 2022. URL: https://www.gnu.org/software/libc/.

[6] ENEA. *Operating systems for telecom, networking, and embedded*. URL: https://www.enea.com/products-services/operating-systems.

[7] ETSI. *Terrestrial Trunked Radio (TETRA); Voice plus Data (V+D); Designers' guide; Part 1: Overview, technical description and radio aspects*. 1997. URL: https://www.etsi.org/deliver/etsi_etr/300_399/30001/01_60/etr_30001e01p.pdf.

[8] ETSI. *Terrestrial Trunked Radio (TETRA); Voice plus Data (V+D); Part 2: Air Interface (AI)*. 2000. URL: https://www.etsi.org/deliver/etsi_ts/100300_100399/10039202/02.01.01_60/ts_10039202v020101p.pdf.

[9] ETSI. *Terrestrial Trunked Radio (TETRA); Voice plus Data (V+D); Part 7: Security*. 2001. URL: https://www.etsi.org/deliver/etsi_en/300300_300399/30039207/02.01.01_60/en_30039207v020101p.pdf.

[10] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX, 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[11] Jean loup Gailly and Mark Adler. *zlib, A Massively Spiffy Yet Delicately Unobtrusive Compression Library*. 2022. URL: https://www.zlib.net/.

[12] Motorola Inc. *MBTS DIMETRA BASE STATION*. 2004. URL: https://studylib.net/doc/18750327/mbts-dimetra-base-station-the-motorola-mbts-is-the-flexible.

[13] Motorola Inc. *MPC750 RISC Microprocessor Technical Summary*. 1997. URL: https://www.nxp.com/docs/en/data-sheet/MPC750.pdf.

[14] Martin Meyers. *Fuzzing HTTP inputs of IoT binaries emulated in Qiling*. Research internship report at Secura, Radboud University. 2021.

[15] LLC Motorola Trademark Holdings. *MTS2 TETRA Base Station*. 2022. URL: https://www.motorolasolutions.com/en_xl/products/tetra/infrastructure/mts2.html.

[16] Uwe Müller, Eicke Hauck, Timm Welz, Jiska Classen, and Matthias Hollick. "Dinosaur Resurrection: PowerPC Binary Patching for Base Station Analysis". In: *Proceedings 2021 Workshop on Binary Analysis Research*. Internet Society, 2021. DOI: 10.14722/bar.2021.23009. URL: https://www.ndss-symposium.org/wp-content/uploads/bar2021_23009_paper.pdf.

[17] *QEMU: A generic and open source machine emulator and virtualizer*. 2022. URL: https://www.qemu.org/ (visited on 10/04/2022).

[18] *Qiling Framework*. URL: https://qiling.io/ (visited on 10/07/2022).

[19] Nguyen Anh Quynh and Dang Hoang Vu. *Unicorn: The Ultimate CPU emulator*. 2022. URL: https://www.unicorn-engine.org/ (visited on 10/04/2022).

[20] LLC SaurikIT. *Cydia Substrate*. 2012-2014. URL: http://www.cydiasubstrate.com/.
[21] Fuzz Stati0n and Joseph Carlos. *Fuzzgoat*. URL: https://github.com/fuzzcorp/fuzzgoat.
[22] Ischa Stork. *Fuzzing of emulated firmware using the Qiling emulation framework*. Research internship report, Radboud University. 2020.
[23] AFL++ team. *AFL++ documentation*. 2022. URL: https://aflplus.plus/.
[24] AFL++ team. *UnicornAFL*. 2022. URL: https://github.com/AFLplusplus/unicornafl.
[25] GCC Team. *3.11 Options That Control Optimization*. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.
[26] GCC Team. *6.47.5.2 Specifying Registers for Local Variables*. URL: https://gcc.gnu.org/onlinedocs/gcc/Local-Register-Variables.html.
[27] *TETRA Around the World*. 2012. URL: https://web.archive.org/web/20120313035919/http://www.tetrahealth.info/worldCountries.htm.
[28] Michigan State University. *Integration with Operating Systems*. URL: http://sens.cse.msu.edu/Software/Telelogic-3.5/locale/english/help/htmlhlp/rtos.html#908400.
[29] Nathan Voss. *afl-unicorn: Part 2 — Fuzzing the 'Unfuzzable'*. 2017. URL: https://hackernoon.com/afl-unicorn-part-2-fuzzing-the-unfuzzable-bea8de3540a5 (visited on 10/07/2022).
[30] Jason Williams. *FSK_Messaging_Service*. URL: https://github.com/trailofbits/cb-multios/tree/master/challenges/FSK_Messaging_Service (visited on 10/07/2022).
[31] Nikola Zlatanov. *Architecture and Operation of a Watchdog Timer*. July 2014. DOI: 10.13140/RG.2.1.1149.1605.

Appendices

A AFL++ outputs

This appendix includes screenshots of all the AFL++ outputs after the fuzzing has been terminated. The outputs are referred in Section 5.5.3, Section 5.5.4 and Section 5.5.5. The outputs are listed here for completeness and validation of the research. For example, it may be interesting to see the exact amount of corpus counts found for each AFL++ instance to get an indication of the amount of found inputs. The exact meaning of all the statistics can be found in the AFL++ documentation[4].

```
american fuzzy lop ++4.02a {parent} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 15 hrs, 3 min, 30 sec       │  cycles done : 1014   │
│   last new find : 0 days, 2 hrs, 51 min, 41 sec       │ corpus count : 294    │
│last saved crash : none seen yet                       │saved crashes : 0      │
│ last saved hang : none seen yet                       │  saved hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 21.1013 (7.1%)    │    map density : 0.91% / 1.99%         │
│  runs timed out : 0 (0.00%)         │ count coverage : 2.86 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : splice 9              │ favored items : 62 (21.09%)            │
│ stage execs : 18/55 (32.73%)        │  new edges on : 79 (26.87%)            │
│ total execs : 68.0M                 │ total crashes : 0 (0 saved)            │
│  exec speed : 482.8/sec             │  total tmouts : 2535 (60 saved)        │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 18/86.4k, 10/86.1k, 1/85.5k           │    levels : 7         │
│  byte flips : 2/10.8k, 2/10.3k, 1/9754              │   pending : 0         │
│ arithmetics : 1/591k, 2/336k, 1/125k                │  pend fav : 0         │
│  known ints : 4/46.7k, 16/222k, 7/391k              │ own finds : 228       │
│  dictionary : 0/0, 0/0, 0/0, 0/0                    │  imported : 63        │
│havoc/splice : 137/25.1M, 28/40.9M                   │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused        ├───────────────────────┘
│    trim/eff : disabled, 61.52%                      │          [cpu000:100%]
└─────────────────────────────────────────────────────┘
```

Figure 18: AFL++ output of the parent AFL++-process of fuzzing U-LOCATION-UPDATE-DEMAND

[4] https://aflplus.plus/docs/status_screen/

```
american fuzzy lop ++4.02a {child1} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 15 hrs, 1 min, 15 sec       │  cycles done : 348    │
│   last new find : 0 days, 2 hrs, 52 min, 0 sec        │ corpus count : 286    │
│last saved crash : none seen yet                       │saved crashes : 0      │
│ last saved hang : none seen yet                       │  saved hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage─┴───────────────────────┤
│  now processing : 277.253 (96.9%)   │    map density : 0.81% / 1.99%        │
│  runs timed out : 0 (0.00%)         │ count coverage : 2.86 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : splice 15             │ favored items : 64 (22.38%)           │
│ stage execs : 9/146 (6.16%)         │  new edges on : 78 (27.27%)           │
│ total execs : 68.8M                 │ total crashes : 0 (0 saved)           │
│  exec speed : 467.7/sec             │  total tmouts : 2433 (51 saved)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 7/72.9k, 1/72.6k, 1/72.1k              │    levels : 4         │
│  byte flips : 0/9112, 1/8674, 2/8156                 │   pending : 3         │
│ arithmetics : 7/499k, 2/277k, 1/100k                 │  pend fav : 0         │
│  known ints : 1/39.7k, 4/187k, 4/330k                │ own finds : 133       │
│  dictionary : 0/0, 0/0, 0/0, 0/0                     │  imported : 151       │
│havoc/splice : 85/37.0M, 18/30.1M                     │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused         ├───────────────────────┘
│    trim/eff : 3.71%/2818, 59.29%                     │          [cpu001:116%]
└──────────────────────────────────────────────────────┘
```

Figure 19: AFL++ output of the first child AFL++-process of fuzzing U-LOCATION-UPDATE-DEMAND

```
american fuzzy lop ++4.02a {child2} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 15 hrs, 1 min, 13 sec       │  cycles done : 453    │
│   last new find : 0 days, 2 hrs, 52 min, 9 sec        │ corpus count : 254    │
│last saved crash : none seen yet                       │saved crashes : 0      │
│ last saved hang : none seen yet                       │  saved hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage─┴───────────────────────┤
│  now processing : 64.1633 (25.2%)   │    map density : 0.85% / 1.99%        │
│  runs timed out : 0 (0.00%)         │ count coverage : 2.86 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : splice 15             │ favored items : 62 (24.41%)           │
│ stage execs : 9/36 (25.00%)         │  new edges on : 78 (30.71%)           │
│ total execs : 69.7M                 │ total crashes : 0 (0 saved)           │
│  exec speed : 460.5/sec             │  total tmouts : 82 (17 saved)         │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 8/67.6k, 1/67.3k, 2/66.9k              │    levels : 3         │
│  byte flips : 0/8449, 1/8050, 2/7579                 │   pending : 4         │
│ arithmetics : 1/463k, 0/258k, 0/90.7k                │  pend fav : 0         │
│  known ints : 0/36.8k, 6/175k, 2/308k                │ own finds : 97        │
│  dictionary : 0/0, 0/0, 0/0, 0/0                     │  imported : 154       │
│havoc/splice : 62/37.7M, 14/30.5M                     │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused         ├───────────────────────┘
│    trim/eff : 3.58%/2559, 59.55%                     │          [cpu002:150%]
└──────────────────────────────────────────────────────┘
```

Figure 20: AFL++ output of the second child AFL++-process of fuzzing U-LOCATION-UPDATE-DEMAND

```
american fuzzy lop ++4.02a {child3} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 15 hrs, 1 min, 14 sec       │  cycles done : 428    │
│   last new find : 0 days, 2 hrs, 52 min, 13 sec       │ corpus count : 286    │
│last saved crash : none seen yet                       │saved crashes : 0      │
│ last saved hang : none seen yet                       │  saved hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage─┴───────────────────────┤
│  now processing : 24.757 (8.4%)    │    map density : 0.80% / 1.99%        │
│  runs timed out : 0 (0.00%)        │ count coverage : 2.86 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : splice 7             │ favored items : 63 (22.03%)           │
│ stage execs : 9/36 (25.00%)        │  new edges on : 80 (27.97%)           │
│ total execs : 70.0M                │ total crashes : 0 (0 saved)           │
│  exec speed : 459.9/sec            │  total tmouts : 1884 (51 saved)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 6/69.8k, 1/69.5k, 0/69.0k             │    levels : 4         │
│  byte flips : 0/8721, 1/8308, 2/7792                │   pending : 3         │
│ arithmetics : 6/478k, 1/262k, 0/94.0k               │  pend fav : 0         │
│  known ints : 0/38.1k, 6/180k, 4/316k               │ own finds : 124       │
│  dictionary : 0/0, 0/0, 0/0, 0/0                    │  imported : 159       │
│havoc/splice : 78/34.6M, 21/33.8M                    │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused        ├───────────────────────┘
│    trim/eff : 3.40%/2653, 58.01%                    │          [cpu003:100%]
└─────────────────────────────────────────────────────┘
```

Figure 21: AFL++ output of the third child AFL++-process of fuzzing U-LOCATION-UPDATE-DEMAND

```
american fuzzy lop ++4.02a {child4} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 15 hrs, 1 min, 17 sec       │  cycles done : 401    │
│   last new find : 0 days, 2 hrs, 52 min, 23 sec       │ corpus count : 285    │
│last saved crash : none seen yet                       │saved crashes : 0      │
│ last saved hang : none seen yet                       │  saved hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage─┴───────────────────────┤
│  now processing : 105.328 (36.8%)  │    map density : 1.04% / 1.99%        │
│  runs timed out : 0 (0.00%)        │ count coverage : 2.86 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc                │ favored items : 63 (22.11%)           │
│ stage execs : 180/291 (61.86%)     │  new edges on : 78 (27.37%)           │
│ total execs : 70.3M                │ total crashes : 0 (0 saved)           │
│  exec speed : 463.4/sec            │  total tmouts : 1872 (54 saved)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 7/73.2k, 1/72.9k, 2/72.4k             │    levels : 5         │
│  byte flips : 0/9149, 1/8789, 3/8269                │   pending : 2         │
│ arithmetics : 3/505k, 3/280k, 0/100k                │  pend fav : 0         │
│  known ints : 1/40.2k, 4/191k, 4/335k               │ own finds : 128       │
│  dictionary : 0/0, 0/0, 0/0, 0/0                    │  imported : 153       │
│havoc/splice : 74/33.4M, 28/35.2M                    │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused        ├───────────────────────┘
│    trim/eff : 2.90%/2874, 59.47%                    │          [cpu004:100%]
└─────────────────────────────────────────────────────┘
```

Figure 22: AFL++ output of the fourth child AFL++-process of fuzzing U-LOCATION-UPDATE-DEMAND

```
american fuzzy lop ++4.02a {parent} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─┐
│        run time : 0 days, 4 hrs, 18 min, 25 sec       │  cycles done : 888 │
│   last new find : 0 days, 2 hrs, 28 min, 49 sec       │  corpus count : 27 │
│last saved crash : none seen yet                       │saved crashes : 0   │
│ last saved hang : none seen yet                       │  saved hangs : 0   │
├─ cycle progress ────────────────────┬─ map coverage ─┴────────────────────┤
│  now processing : 26*30 (96.3%)    │    map density : 0.66% / 0.84%      │
│  runs timed out : 0 (0.00%)        │ count coverage : 1.25 bits/tuple    │
├─ stage progress ────────────────────┼─ findings in depth ─────────────────┤
│  now trying : splice 13            │ favored items : 10 (37.04%)         │
│ stage execs : 30/384 (7.81%)       │  new edges on : 9 (33.33%)          │
│ total execs : 7.93M                │ total crashes : 0 (0 saved)         │
│  exec speed : 524.8/sec            │  total tmouts : 239 (12 saved)      │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───┤
│   bit flips : 2/2960, 1/2934, 0/2882                │    levels : 3     │
│  byte flips : 0/370, 0/344, 0/301                   │   pending : 0     │
│ arithmetics : 1/20.6k, 0/9327, 0/2678               │  pend fav : 0     │
│  known ints : 0/1682, 0/7740, 0/12.5k               │ own finds : 25    │
│  dictionary : 0/0, 0/0, 0/0, 0/0                    │  imported : 0     │
│havoc/splice : 21/4.42M, 1/3.44M                     │ stability : 100.00%│
│py/custom/rq : unused, unused, unused, unused        ├───────────────────┘
│    trim/eff : disabled, 23.73%                      │          [cpu000:100%]
└─────────────────────────────────────────────────────┘
```

Figure 23: AFL++ output of the parent AFL++-process of fuzzing U-SDS-DATA

```
american fuzzy lop ++4.02a {child1} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─┐
│        run time : 0 days, 4 hrs, 18 min, 24 sec       │  cycles done : 651 │
│   last new find : 0 days, 2 hrs, 28 min, 19 sec       │  corpus count : 28 │
│last saved crash : none seen yet                       │saved crashes : 0   │
│ last saved hang : none seen yet                       │  saved hangs : 0   │
├─ cycle progress ────────────────────┬─ map coverage ─┴────────────────────┤
│  now processing : 3.651 (10.7%)    │    map density : 0.27% / 0.84%      │
│  runs timed out : 0 (0.00%)        │ count coverage : 1.25 bits/tuple    │
├─ stage progress ────────────────────┼─ findings in depth ─────────────────┤
│  now trying : splice 14            │ favored items : 10 (35.71%)         │
│ stage execs : 30/40 (75.00%)       │  new edges on : 9 (32.14%)          │
│ total execs : 7.92M                │ total crashes : 0 (0 saved)         │
│  exec speed : 501.9/sec            │  total tmouts : 177 (13 saved)      │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───┤
│   bit flips : 2/2808, 1/2780, 0/2724                │    levels : 3     │
│  byte flips : 0/351, 0/323, 0/275                   │   pending : 0     │
│ arithmetics : 0/19.6k, 0/10.9k, 0/4023              │  pend fav : 0     │
│  known ints : 0/1558, 0/6728, 0/10.5k               │ own finds : 26    │
│  dictionary : 0/0, 0/0, 0/0, 0/0                    │  imported : 0     │
│havoc/splice : 18/3.59M, 6/4.27M                     │ stability : 100.00%│
│py/custom/rq : unused, unused, unused, unused        ├───────────────────┘
│    trim/eff : disabled, 25.45%                      │          [cpu001:100%]
└─────────────────────────────────────────────────────┘
```

Figure 24: AFL++ output of the first child AFL++-process of fuzzing U-SDS-DATA

```
american fuzzy lop ++4.02a {child2} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─┐
│        run time : 0 days, 4 hrs, 18 min, 26 sec       │  cycles done : 794 │
│   last new find : 0 days, 2 hrs, 28 min, 19 sec       │  corpus count : 27 │
│last saved crash : none seen yet                       │saved crashes : 0   │
│ last saved hang : none seen yet                       │  saved hangs : 0   │
├─ cycle progress ────────────────────┬─ map coverage ─┴────────────────────┤
│  now processing : 3.793 (11.1%)    │    map density : 0.27% / 0.84%      │
│  runs timed out : 0 (0.00%)        │ count coverage : 1.25 bits/tuple    │
├─ stage progress ────────────────────┼─ findings in depth ─────────────────┤
│  now trying : splice 10            │ favored items : 9 (33.33%)          │
│ stage execs : 27/55 (49.09%)       │  new edges on : 9 (33.33%)          │
│ total execs : 7.99M                │ total crashes : 0 (0 saved)         │
│  exec speed : 491.4/sec            │  total tmouts : 153 (11 saved)      │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───┤
│   bit flips : 2/2720, 1/2693, 0/2639                │    levels : 3     │
│  byte flips : 0/340, 0/313, 0/267                   │   pending : 0     │
│ arithmetics : 0/19.0k, 0/7217, 0/879                │  pend fav : 0     │
│  known ints : 0/1588, 0/7524, 0/11.5k               │ own finds : 25    │
│  dictionary : 0/0, 0/0, 0/0, 0/0                    │  imported : 0     │
│havoc/splice : 20/3.82M, 3/4.11M                     │ stability : 100.00%│
│py/custom/rq : unused, unused, unused, unused        ├───────────────────┘
│    trim/eff : disabled, 21.82%                      │          [cpu002:100%]
└─────────────────────────────────────────────────────┘
```

Figure 25: AFL++ output of the second child AFL++-process of fuzzing U-SDS-DATA

```
american fuzzy lop ++4.02a {child3} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─┐
│        run time : 0 days, 4 hrs, 18 min, 29 sec       │  cycles done : 865 │
│   last new find : 0 days, 2 hrs, 28 min, 22 sec       │  corpus count : 26 │
│last saved crash : none seen yet                       │saved crashes : 0   │
│ last saved hang : none seen yet                       │  saved hangs : 0   │
├─ cycle progress ────────────────────┬─ map coverage ─┴────────────────────┤
│  now processing : 8.865 (30.8%)    │    map density : 0.64% / 0.84%      │
│  runs timed out : 0 (0.00%)        │ count coverage : 1.25 bits/tuple    │
├─ stage progress ────────────────────┼─ findings in depth ─────────────────┤
│  now trying : havoc                │ favored items : 9 (34.62%)          │
│ stage execs : 324/440 (73.64%)     │  new edges on : 9 (34.62%)          │
│ total execs : 7.92M                │ total crashes : 0 (0 saved)         │
│  exec speed : 490.1/sec            │  total tmouts : 162 (11 saved)      │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───┤
│   bit flips : 2/1888, 1/1864, 0/1816                │    levels : 3     │
│  byte flips : 0/236, 0/212, 0/173                   │   pending : 0     │
│ arithmetics : 0/13.2k, 0/5972, 0/1188               │  pend fav : 0     │
│  known ints : 0/1071, 0/4828, 0/7205                │ own finds : 25    │
│  dictionary : 0/0, 0/0, 0/0, 0/0                    │  imported : 0     │
│havoc/splice : 20/4.25M, 2/3.63M                     │ stability : 100.00%│
│py/custom/rq : unused, unused, unused, unused        ├───────────────────┘
│    trim/eff : disabled, 17.50%                      │          [cpu003:100%]
└─────────────────────────────────────────────────────┘
```

Figure 26: AFL++ output of the third child AFL++-process of fuzzing U-SDS-DATA

```
american fuzzy lop ++4.02a {child4} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 4 hrs, 18 min, 32 sec       │  cycles done : 924    │
│   last new find : 0 days, 2 hrs, 33 min, 4 sec        │ corpus count : 28     │
│last saved crash : none seen yet                       │saved crashes : 0      │
│ last saved hang : none seen yet                       │  saved hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 11.924 (39.3%)    │    map density : 0.66% / 0.84%         │
│  runs timed out : 0 (0.00%)         │ count coverage : 1.25 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : splice 7              │ favored items : 9 (32.14%)             │
│ stage execs : 54/55 (98.18%)        │  new edges on : 9 (32.14%)             │
│ total execs : 7.94M                 │ total crashes : 0 (0 saved)            │
│  exec speed : 477.4/sec             │  total tmouts : 153 (12 saved)         │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 2/2000, 1/1973, 0/1919                │    levels : 6         │
│  byte flips : 0/250, 0/223, 0/177                   │   pending : 0         │
│ arithmetics : 0/14.0k, 0/7078, 0/932                │  pend fav : 0         │
│  known ints : 0/1129, 0/5319, 0/7555                │ own finds : 25        │
│  dictionary : 0/0, 0/0, 0/0, 0/0                    │  imported : 0         │
│havoc/splice : 23/4.20M, 1/3.70M                     │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused        ├───────────────────────┘
│    trim/eff : disabled, 13.33%                      │          [cpu004:116%]
└─────────────────────────────────────────────────────┘
```

Figure 27: AFL++ output of the fourth child AFL++-process of fuzzing U-SDS-DATA

```
american fuzzy lop ++4.02a {parent} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 13 hrs, 23 min, 7 sec       │  cycles done : 2499   │
│   last new find : 0 days, 11 hrs, 47 min, 41 sec      │ corpus count : 57     │
│last saved crash : none seen yet                       │saved crashes : 0      │
│ last saved hang : none seen yet                       │  saved hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 53*75 (93.0%)     │    map density : 0.72% / 1.30%         │
│  runs timed out : 0 (0.00%)         │ count coverage : 1.29 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : splice 7              │ favored items : 23 (40.35%)            │
│ stage execs : 65/441 (14.74%)       │  new edges on : 26 (45.61%)            │
│ total execs : 71.6M                 │ total crashes : 0 (0 saved)            │
│  exec speed : 614.2/sec             │  total tmouts : 128 (19 saved)         │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 2/5688, 1/5635, 0/5529                │    levels : 6         │
│  byte flips : 0/711, 0/658, 0/557                   │   pending : 0         │
│ arithmetics : 2/39.8k, 2/26.4k, 2/11.5k             │  pend fav : 0         │
│  known ints : 0/2871, 0/13.4k, 0/21.4k              │ own finds : 44        │
│  dictionary : 0/0, 0/0, 0/538, 0/661                │  imported : 10        │
│havoc/splice : 32/27.9M, 5/43.6M                     │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused        ├───────────────────────┘
│    trim/eff : disabled, 24.78%                      │          [cpu000: 66%]
└─────────────────────────────────────────────────────┘
                                                      ^C
```

Figure 28: AFL++ output of the parent AFL++-process of fuzzing U-SDS-STATUS

```
american fuzzy lop ++4.02a {child1} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 13 hrs, 22 min, 54 sec      │  cycles done : 1632   │
│   last new find : 1 days, 0 hrs, 36 min, 28 sec       │ corpus count : 62     │
│last saved crash : none seen yet                       │saved crashes : 0      │
│ last saved hang : none seen yet                       │  saved hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage─┴───────────────────────┤
│  now processing : 18.927 (29.0%)    │    map density : 0.74% / 1.30%        │
│  runs timed out : 0 (0.00%)         │ count coverage : 1.29 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc                 │ favored items : 23 (37.10%)           │
│ stage execs : 277/440 (62.95%)      │  new edges on : 24 (38.71%)           │
│ total execs : 69.0M                 │ total crashes : 0 (0 saved)           │
│  exec speed : 553.5/sec             │  total tmouts : 2392 (27 saved)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 2/5776, 1/5722, 0/5614                │    levels : 4         │
│  byte flips : 0/722, 0/668, 0/568                   │   pending : 1         │
│ arithmetics : 1/40.3k, 3/24.8k, 3/10.2k             │  pend fav : 0         │
│  known ints : 0/3391, 1/14.2k, 0/22.3k              │ own finds : 50        │
│  dictionary : 0/0, 0/0, 0/587, 0/729                │  imported : 9         │
│havoc/splice : 29/36.7M, 12/32.2M                    │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused        ├───────────────────────┘
│    trim/eff : 12.90%/225, 26.36%                    │          [cpu001: 66%]
└─────────────────────────────────────────────────────┘          ^C
```

Figure 29: AFL++ output of the first child AFL++-process of fuzzing U-SDS-STATUS

```
american fuzzy lop ++4.02a {child2} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 13 hrs, 22 min, 40 sec      │  cycles done : 1817   │
│   last new find : 0 days, 12 hrs, 7 min, 40 sec       │ corpus count : 62     │
│last saved crash : none seen yet                       │saved crashes : 0      │
│ last saved hang : none seen yet                       │  saved hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage─┴───────────────────────┤
│  now processing : 11.43128 (17.7%)  │    map density : 0.60% / 1.30%        │
│  runs timed out : 0 (0.00%)         │ count coverage : 1.29 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc                 │ favored items : 23 (37.10%)           │
│ stage execs : 12/440 (2.73%)        │  new edges on : 26 (41.94%)           │
│ total execs : 69.7M                 │ total crashes : 0 (0 saved)           │
│  exec speed : 518.8/sec             │  total tmouts : 1614 (27 saved)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 2/5120, 1/5064, 0/4952                │    levels : 4         │
│  byte flips : 0/640, 0/584, 0/478                   │   pending : 0         │
│ arithmetics : 1/35.6k, 2/15.1k, 3/3518              │  pend fav : 0         │
│  known ints : 0/3335, 0/13.9k, 0/20.2k              │ own finds : 50        │
│  dictionary : 0/0, 0/0, 0/432, 0/555                │  imported : 9         │
│havoc/splice : 32/39.1M, 11/30.5M                    │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused        ├───────────────────────┘
│    trim/eff : 18.41%/178, 12.62%                    │          [cpu002:100%]
└─────────────────────────────────────────────────────┘          ^C
```

Figure 30: AFL++ output of the second child AFL++-process of fuzzing U-SDS-STATUS

```
american fuzzy lop ++4.02a {child3} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 13 hrs, 22 min, 35 sec      │  cycles done : 1641   │
│   last new find : 1 days, 2 hrs, 16 min, 34 sec       │ corpus count : 62     │
│last saved crash : none seen yet                       │saved crashes : 0      │
│ last saved hang : none seen yet                       │  saved hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage─┴───────────────────────┤
│  now processing : 26.1112 (41.9%)   │    map density : 0.74% / 1.30%        │
│  runs timed out : 0 (0.00%)         │ count coverage : 1.29 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : splice 13             │ favored items : 22 (35.48%)           │
│ stage execs : 19/55 (34.55%)        │  new edges on : 25 (40.32%)           │
│ total execs : 70.1M                 │ total crashes : 0 (0 saved)           │
│  exec speed : 503.4/sec             │  total tmouts : 2131 (25 saved)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 3/4608, 2/4551, 0/4437                │    levels : 3         │
│  byte flips : 0/576, 0/519, 0/411                   │   pending : 0         │
│ arithmetics : 0/32.2k, 2/17.8k, 2/3368              │  pend fav : 0         │
│  known ints : 0/2480, 0/12.1k, 0/17.3k              │ own finds : 48        │
│  dictionary : 0/0, 0/0, 0/444, 0/606                │  imported : 11        │
│havoc/splice : 30/37.1M, 11/32.9M                    │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused        ├───────────────────────┘
│    trim/eff : 13.64%/145, 4.30%                     │          [cpu003:116%]
└─────────────────────────────────────────────────────┘          ^C
```

Figure 31: AFL++ output of the third child AFL++-process of fuzzing U-SDS-STATUS

```
american fuzzy lop ++4.02a {child4} (...rget/release/fuzzgoat-fuzzer) [fast]
┌─ process timing ──────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 13 hrs, 22 min, 30 sec      │  cycles done : 1400   │
│   last new find : 1 days, 1 hrs, 10 min, 6 sec        │ corpus count : 67     │
│last saved crash : none seen yet                       │saved crashes : 0      │
│ last saved hang : none seen yet                       │  saved hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage─┴───────────────────────┤
│  now processing : 8.30251 (11.9%)   │    map density : 0.60% / 1.30%        │
│  runs timed out : 0 (0.00%)         │ count coverage : 1.29 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc                 │ favored items : 22 (32.84%)           │
│ stage execs : 104/440 (23.64%)      │  new edges on : 25 (37.31%)           │
│ total execs : 72.6M                 │ total crashes : 0 (0 saved)           │
│  exec speed : 523.4/sec             │  total tmouts : 1895 (30 saved)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 2/5440, 1/5378, 0/5254                │    levels : 4         │
│  byte flips : 0/680, 0/618, 0/502                   │   pending : 2         │
│ arithmetics : 0/38.0k, 2/19.2k, 2/2969              │  pend fav : 0         │
│  known ints : 0/3023, 1/15.1k, 2/21.3k              │ own finds : 50        │
│  dictionary : 0/0, 0/0, 0/494, 0/640                │  imported : 11        │
│havoc/splice : 25/34.9M, 20/37.6M                    │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused        ├───────────────────────┘
│    trim/eff : 15.79%/160, 16.98%                    │          [cpu004:150%]
└─────────────────────────────────────────────────────┘          ^C
```

Figure 32: AFL++ output of the fourth child AFL++-process of fuzzing U-SDS-STATUS