Title: Fuzzing the Pacemaker Home Monitoring Unit

Students: Jakob Stenersen Kok & Bendik Aalmen Markussen

Problem description:

Medical devices are not only products whose function is to simply assist in everyday life. In fact, some medical devices provide life-critical functionality, and certain patients are entirely dependent on such devices to live long and healthy lives. The pacemaker is an excellent example of one such device. However, modern technology has had a significant impact on pacemakers. They are no longer standalone devices but take part in a larger ecosystem of both medical and non-medical devices. It has become increasingly important that the entire pacemaker ecosystem is sufficiently secured. If not, the very device that keeps patients alive and well, could be utilized in a targeted cyberattack to breach their privacy or inflict physical harm upon patients.

Specifically, we set out to research the security of the network interface of the latest Home Monitoring Unit (HMU) from a specific vendor. The HMU is an embedded device that relays private patient data from the pacemaker to the backend servers of the vendor. From there, medical professionals can access the patient data, thereby minimizing the patient's number of visits to the hospital. The HMU is a convenient device. It can even be critical for patients that have conditions that require close follow up. Since the HMU is physically accessible and easily available to purchase on second-hand online markets, it is crucial that it is sufficiently secured against hacking. We will in this master thesis perform a security evaluation, to uncover and report potential security vulnerabilities in the HMU.

Our contributions are two-folded. First, we will examine whether a range of vulnerabilities found in research into older HMUs, also exists in the latest device. This will lay a foundation for our next stage - which is our main focus and contribution in our thesis. Research has uncovered that the latest HMU is also capable of receiving data, and we wish to research if remotely exploitable vulnerabilities exist in the HMU. This portion of our research entails the development of a fuzzer to automate attacks against the HMU through its networking interface. The goal is any attack which interferes with the HMU's normal operation. To assess the security level of the HMU, we will be using a Black-Box testing methodology.

Responsible professor: Marie Elisabeth Gaup Moe, NTNU and Mnemonic

Supervisor: Guillaume Bour & Ravishankar Borgaonkar, SINTEF

Abstract

With the advent of Internet of Things (IoT)-devices, an increasing number of devices are connected to the Internet. Within the medical world, this is no different. New technology is used to ensure the best possible treatment for patients. However, with this proliferation of Internet-connected devices, a broad attack-surface is opened. Cybercriminals can potentially leverage new attack vectors to monetize on illegal activities. In this thesis work, we set out to investigate whether the newest version of the pacemaker HMU from the German vendor Biotronik, contains vulnerabilities that can be exploited by an attacker wanting to compromise either the safety or the privacy of a patient. In the first scope of this research, by incorporating a black-box methodology, we found that the HMU, and its corresponding firmware version, has inherent hardware security deficits. These can possibly aid an attacker having physical access in developing sophisticated malware that impacts patients' safety or privacy. In the second scope of this thesis, we scrutinized the SMS interface of the HMU. To uncover vulnerabilities in the SMS interface, we have developed a universally applicable SMS fuzzer framework. It can be used to fuzz the SMS interface of any device that implement the SMS protocol, with monitoring implemented at the network side. The code is made open source and is a contribution to the cybersecurity research community. Our research confirmed that the SMS interface is vulnerable, and we uncovered several SMS-messages that crashes the modem of the HMU. Moreover, we outline how these findings can be used to launch a remote Denial of Service (DoS) attack on the HMU. Our contributions include techniques for mitigation of the identified risks. The offending SMSs we uncovered are not unique to the HMU, but affects all IoT-devices with the same modem. The affected vendor has been contacted according to a coordinated vulnerability disclosure process.

Sammendrag

Ved introduksjonen av IoT-teknologi har vi sett en økning av antall enheter som er koblet på internett. Det preger også den medisinske verden, hvor denne teknologien blir benyttet for å sikre best mulig pasientbehandling. Med det økende antallet enheter som er koblet på nett, følger det også et større trusselbilde. Hackere kan potensielt utnytte nye angrepsvektorer med mål om å skade pasientens liv og helse, eller utnytte informasjonen til økonomisk vinning. I denne masteroppgaven gjør vi en sikkerhetsanalyse av den siste generasjonen hjemme-monitoreringsenhet (HMU) for pacemakere fra den tyske produsenten Biotronik. Arbeidet redegjør for enhetens sårbarheter som kan utnyttes av en angriper som ønsker å skade pasientens helse eller personvern. Masteroppgaven er todelt. Ved å følge en black-box metode for sikkerhetstesting, avdekkes det at HMUen har iboende mangler hva angår sikkerhet. Potensielt kan dette utnyttes til å utvikle mer omfattende angrep av angripere som har fysisk tilgang til HMUen. I det andre prosjektområdet testes SMS-implementasjonen til HMUen. For å avdekke sårbarheter i SMS-implementasjonen har vi utviklet et universelt SMS fuzzer-rammeverk. Dette verktøyet kan brukes til å teste SMS-implementasjoner til alle enheter som kan sende og motta SMS, hvor vi monitorerer fra nettverkets perspektiv. Koden for rammeverket er open source og bidrar til videre forskning innen cybersikkerhet. Våre funn bekrefter at SMS-implementasjonen har vesentlige sårbarheter. En rekke SMS-er viste seg å være i stand til å krasje modemet til HMUen. I sin tur kan dette brukes i et DoS-angrep på enheten. Til slutt presenterer vi teknikker og metoder for skadebegrensning av de identifiserte risikoområdene. SMS-meldingene vi avdekket er ikke spesifikke til HMUen, men kan krasje alle IoT-enheter som benytter samme modem. De påvirkede leverandørene er, i tråd med prinsippet om ansvarlig sårbarhetsformidling, blitt informert om funnene i studien.

Preface

This Masters's Thesis is the final deliverable of Jakob Stenersen Kok and Bendik Aalmen Markussen of their Master of Science degrees in Communication Technology and Digital Security, with a specialization in Information Security, at the Department of Information Security and Communication Technology, Norwegian University of Science and Technology (NTNU).

Acknowledgements

This work would not have been possible without the help of a handful of people. We would like to express our deepest gratitude to those directly involved with the project at SINTEF. *Marie Elisabeth Gaup Moe* has been our responsible professor and has always shown interest in our project and contributed with great guidance, reviews, and comments. *Ravishankar Borgaonkar* has been our supervisor and contributed with his substantial knowledge on mobile network security and experimental setup and testing techniques within the field. The results of this project would have looked very different if it was not for him. Lastly, we would like to give a special thanks to *Guillaume Bour*, who is also our supervisor, for his considerable involvement throughout the project. He has continuously provided his profound expertise on every subject and always been easily available. Furthermore, he has been responsible for the acquisition of necessary testing equipment, for which we are very thankful.

We would also like to express gratitude to a few individuals, who are not directly involved, but contributed positively to the quality of this project. *Martin Lima*, a friend and fellow student at NTNU, took time to help out with the extraction of the flash memory and later proofread our thesis unsolicited. *Altaf Shaik*, a PhD student at the Technical University of Berlin, offered his assistance towards properly configuring the simulated mobile network. We very much appreciate these contributions.

Lastly, due to the outbreak of the Covid-19 virus, we were forced to work from home the majority of this semester. Thanks to our flatmates *Aleksander Walde*, *Eivind Høydal*, *Nils Folvik Danielsen*, and *Per Krisitan Gravdal* for maintaining a healthy working environment at home and keeping spirits high.

Contents

List of Figures

List of Tables

List of Acronyms

3G Third Generation.

AHB Advanced High-Performance Bus Architecture.

API Application Programming Interface.

APN Access Point Name.

APT Advanced Persistent Threat.

AT ATtention.

BTS Base Transceiver Station.

CIA Confidentialty, Integrity and Availability.

CM Connection Management.

COTS Commercial Off-The-Shelf.

CPU Central Processing Unit.

CS Chip Select.

CVS Concurrent Version System.

DoS Denial of Service.

EMS Enhanced Messaging Service.

EU European Union.

FCC Federal Communication Commission.

FSMC Flexible Static Memory Controller.

GSM Global System for Mobile Communication.

HMU Home Monitoring Unit.

ICD Implantable Cardioverter Defibrillator.

IMSI International Mobile Subscriber Identity.

IoT Internet of Things.

IP Internet Protocol.

JTAG Joint Test Action Group.

LTE Long Term Evolution.

MISO Master In/Slave Out.

MitM Man in the Middle.

MM Mobility Management.

MOSI Master Out/Slave In.

NTNU Norwegian University of Science and Technology.

OS Operating System.

OWASP Open Source Web Application Security Project.

PCB Printed Circuit Board.

PDU Protocol Data Unit.

RAM Random Access Memory.

RDP Readout Protection.

RPDU Relay Protocol Data Unit.

RRM Radio Resource Management.

SDR Software Defined Radio.

SM-AL Short-Message Application Layer.

SME Short Message Entity.

SM-RL Short-Message Relay Layer.

SMS Short Message Service.

SMSC Short Message Service Center.

SM-TL Short-Message Transfer Layer.

SPI Serial Peripheral Interface.

TAP Test Access Port.

TCP Transmission Control Protocol.

TPDU Transfer Protocol Data Unit.

TP-UDH TP-User-Data-Header.

UART Universal Asynchronous Receiver/Transmitter.

UDP User Datagram Protocol.

UE User Equipment.

ULPAMI Ultra Low Power Active Medical Implant Systems.

USRP Universal Software Radio Peripheral.

Chapter 1
Introduction

1.1 Definitions of medical devices

Medical devices is a term coined to encompass a broad set of benign technological innovations and products. Overall, medical devices are made to benefit humans, and in one way or another, provide functions related to our health. While some medical devices function as mere assistants in monitoring or improving upon the health of their users, other medical devices provide critical functions towards maintaining human life itself. Therefore, multiple formal definitions of what a medical device is, exists today.

The formal definition of a medical device provided by Norwegian authorities is: *"Any instrument, apparatus, piece of equipment, material or other object, that is to be used alone or in combination, intended for use on humans, which has diagnosis, prevention, monitoring, treatment or alleviation of disease or compensation for an injury or handicap as its purpose"* [Lov05]. Furthermore, the formal definition of an *active medical device* by Norwegian authorities is: *"Any medical device dependent on an energy source, electrical or other, excluding energy sources which arise from the human body or gravity"* [Lov05]. In addition to active medical devices, the Norwegian government also defines *active implantable medical devices*. Such devices are defined as: *"Any active medical device which is intended to be totally or partially introduced, surgically or medically, into the human body or by medical intervention into a natural orifice, and which is intended to remain after the procedure"* [Lov05].

Clearly, these definitions provide classifications, separating trivial and mundane products from more high-end products, which are implanted into their users. Also, these definitions correspond to how critical it is that the medical device performs its functionality. Active implantable medical devices provide functions that the users might be entirely dependent on, as opposed to devices encompassed by the broader definition of medical devices.

For this thesis report, the relevant definition of medical devices is the one defining active implantable medical devices. More specifically, we are focusing on *pacemakers* and *Implantable Cardioverter Defibrillators (ICDs)*, which are devices encompassed by this definition. Both pacemakers and ICDs are battery-powered, surgically implanted into the patient's body, and physically wired to their heart. Patients with pacemaker-implants or an ICD suffer from certain heart conditions in which implanting the device is the settled upon treatment. These devices continuously monitor the heart rhythm and are capable of pacing the heart by issuing electrical stimuli if it is beating too slowly or too rapidly, thereby ensuring a normal heart rhythm. ICDs are also capable of providing a higher voltage shock to the heart of the patient if necessary. As these devices are so highly similar, throughout this thesis, they will be solely referred to as *pacemakers*.

1.2 The pacemaker ecosystem

Although implanting pacemakers in patients with certain heart conditions is a settled-upon treatment, it is not exempt from technological challenges. Advancements in technology have had a significant impact on pacemakers. Their battery lasts for several years, minimizing the number of surgical procedures required. Besides, to ensure the best possible treatment for pacemaker patients, they are no longer standalone devices. In fact, pacemakers take part in a broader ecosystem of both medical and non-medical devices. These devices, except for the pacemaker itself, are:

The programmer is an active medical device itself, although not implantable. It is essentially a computer that typically resides in hospitals or clinics. It wirelessly communicates with the pacemaker and is used to configure/program individual settings of the pacemaker. This requires that the pacemaker is in close proximity to the programmer, which is operated by a medical professional.

The Home Monitoring Unit (HMU) is also an external active medical device. It indeed resides in a patient's home, given that the patient has received a HMU, and wirelessly communicates with the patient's pacemaker. By utilizing wireless communication, the HMU extracts patient data from the pacemaker and forwards this to the vendor's backend server over a different communication interface. This enables medical professionals to access patient data remotely, and the patient can visit the hospital less frequently.

The vendor's backend servers is a cluster of non-medical devices within the ecosystem. The HMU forwards it's retrieved patient data via the operator's network, to the vendor's backend servers where it is stored. Medical professionals can access patient data from these servers by utilizing an online platform.

The operator network is also a non-medical part of the ecosystem. The operator network is the communication network that the HMU utilizes to forward it's retrieved patient data to the vendor's backend servers. Different HMUs utilize different communication technologies. Typical examples include a telephone line, a wireless mobile communications network such as Global System for Mobile Communication (GSM), or the Third Generation (3G) of mobile networks.

Clearly, pacemakers take part in a complex ecosystem of different devices and technologies. Also, several different stakeholders are implicitly involved in the ecosystem. These stakeholders are:

Patients naturally constitute stakeholders in the pacemaker ecosystem, as they are implanted with the pacemakers themselves, and the ecosystem's primary function is to provide them with the possible care.

Medical professionals are also stakeholders in the pacemaker ecosystem. Doctors and practitioners constitute users of the ecosystem as they implant and configure the pacemakers. Besides, they also review patient data.

The vendor manufactures pacemakers, HMUs, and programmers.Therefore, the vendor constitutes an essential stakeholder in the pacemaker ecosystem. The quality of the products and services in the ecosystem, or lack thereof, is due to activities made by the vendor.

Regulatory bodies also constitute important stakeholders. As active implantable medical devices give rise to challenges within health care, products and services in the pacemaker ecosystem are strictly regulated by various authorities.

Researchers constitute the ultimate stakeholders within the pacemaker ecosystem. Research into various aspects such as cardiology, electrical engineering, and cybersecurity of the pacemaker ecosystem is crucial to ensure sufficient quality in products and services.

A figure depicting the pacemaker ecosystem is included below. The figure consists of all the active medical devices, as well as non-medical devices. Also, the pacemaker, which is an active implantable medical device, is included. The figure also depicts patients and medical professionals as stakeholders in the ecosystem.

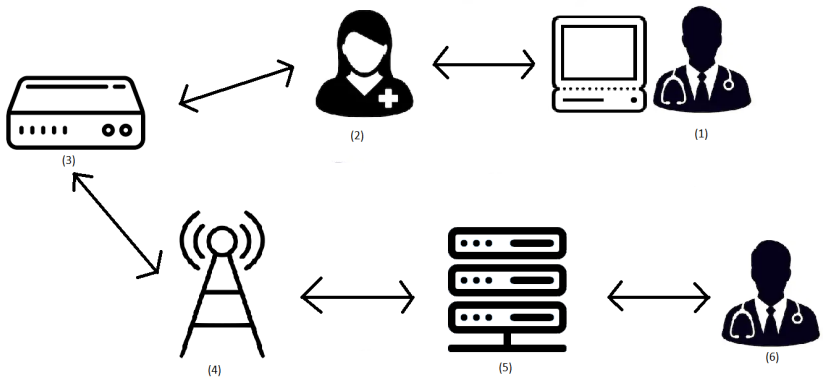


Figure 1.1: 1) The programmer operated by a medical professional. 2) The patient with an implanted pacemaker. 3) The HMU. 4) The operator network. 5) The back-end servers of the vendor. 6) The medical professional reviewing patient data remotely.

1.2.1 Security considerations

As previously mentioned, in this thesis, we are concerned with pacemakers, which are active implantable medical devices. More specifically, since pacemakers take part in the complex infrastructure that is the pacemaker ecosystem, we are concerned with the cybersecurity of this ecosystem. Since the pacemaker ecosystem entails patient care and private patient data, security is of utmost importance. The consequences of a successful attack could lead to the disclosure of private patient data, or even worse, physical harm to the patient. Moreover, a security breach could potentially impact a high number of people. An estimate for Norway alone is that approximately 23 000 patients have pacemakers implanted [SP18]. Globally the estimates are 1.14 million[1].

Based on the severity of potential consequences, one would believe that the cybersecurity of pacemakers and the pacemaker ecosystem has been an important objective and a real concern for the vendors. However, recent research has shown that vulnerabilities do exist within the ecosystem [HHR+08] [RB17] [MSG+16] [KW18] [NB19] [WL19]. A common finding in these reports is the tendency to secure the products by attempting to hide and conceal their technical details. Within the cybersecurity community, this phenomenon is referred to as *security through obscurity* and is generally considered to be an insufficient effort in securing data associated with a patient in the pacemaker ecosystem.

[1] According to `www.statista.com`, accessed on 29.01.2020

The security through obscurity phenomenon is characterized by when vendors attempt to camouflage the inner workings and technical specifications of their devices and services. By keeping such information secret, vendors remain confident that it cannot be exploited for malicious purposes. After all, malicious adversaries cannot use information they do not possess. However, this is considered a bad practice, since secrecy alone is susceptible to brute-force attacks and reverse-engineering. An example is the work by Marin et al. [MSG+16], who reverse-engineered the proprietary communication protocol between a pacemaker and the programmer. Security through obscurity is the opposite of *security by design*, which considers security aspects from an early stage within the design process of devices, thereby ensuring a sufficient security level. This is in line with Kerckhoff's principle, which experts advocate to follow when designing systems that need to be secure. The principle states that *"a system must remain secure if everything about the it except the key is known to the adversary"* [Doe19a].

Physically harming people through digital means is no longer a fictional scenario. Recently, there have been several cyber-attacks whose aim was to cause physical destruction and harm. To illustrate, Ukraine experienced a flood of digital attacks back in 2015 [LC16]. Several power companies were struck, and the malware left many households without electricity, affecting thousands of people. Another example was the ransomware known as WannaCry, which spread reached a global level. In the UK, the ransomware disrupted a range of medical devices and hospitals. According to The Times, estimates state that 70 000 devices were rendered unavailable [UTHG17]. Among these devices were MRI scanners, blood-storage refrigerators, and computers. Furthermore, according to the National Health Services in the UK, almost 20 000 appointments were canceled [Par18].

Since the pacemaker is no longer a standalone device, but in fact, part of a bigger ecosystem, a broad attack surface is opened. A compromise of any of the components in the ecosystem could have severe implications for the patient. Exposing the ecosystem to the internet will inevitably - through sending data between HMUs and the backend servers - open the opportunity for a remote attack on a patient.

1.2.2 Regulatory bodies

A variety of public authorities regulates medical device security. These are specific to nations or unions of nations. If a vendor wishes to sell their medical devices within the jurisdiction of a regulatory authority, it is required by law to comply with current regulations.

The European Union (EU) has recently renewed its regulations of medical devices. Previous regulations dated back to the 1990s, and since there have been significant advancements in technology since then, a renewal was imminent. These new regu-

lations, which will be in full effect as of May 2020, impose stricter control over the quality, safety, and reliability of medical devices. Besides, enhanced mechanisms intended to ensure sufficient information towards patients will be required from vendors. Lastly, the new EU regulations will impose stricter post-market surveillance on the performance of medical devices [Eur17a].

The new EU regulations specifically mention cybersecurity related to medical devices. Electronic programmable systems are explicitly covered in section 17 of chapter 1 since many medical devices also fall within this category. Subsection 17.2 states that: *"For devices that incorporate software or for software that are devices in themselves, the software shall be developed and manufactured in accordance with the state of the art taking into account the principles of development life cycle, risk management, including information security, verification and validation"* [Eur17b]. Also, subsection 17.4 states: *"Manufacturers shall set out minimum requirements concerning hardware, IT networks characteristics and IT security measures, including protection against unauthorized access, necessary to run the software as intended"* [Eur17b].

These new regulations are a measure taken to compensate for the fact that many medical devices, such as pacemakers, are not standalone devices anymore. They are interconnected with a variety of other devices and technologies, which leads to new implications in cybersecurity for these devices. The responsible authority for medical device regulations in Norway is The Norwegian Medicines Agency[2]. They are currently in the process of incorporating the renewed EU regulations into the Norwegian law and aim at completing this process by the EU deadline [The20]. The Norwegian Medicines Agency is also the authority that oversees medical devices in Norway. More specifically, they evaluate side-effects and safety-related incidents of medical devices, in addition to verifying whether a vendor complies with current regulations [The18].

1.3 Scope of our project

Our project considers explicitly the security of the most recent version of HMU manufactured by the German vendor Biotronik, namely the Cardiomessenger 3G Smart. Although this is the most recent HMU from Biotronik, older versions are also still in use. Previous research by Bour [NB19] and Lie [WL19] involved testing the security of older versions of HMUs from Biotronik. Notably, the Cardiomessenger II-S, both the T-line and GSM version, were thoroughly tested.

Our project is a continuation of this previous research and will contribute to the overall knowledge that exists concerning HMU security by performing similar

[2] `https://legemiddelverket.no/english/about-us`, accessed on 05.02.2020

research on the most recent version. Furthermore, our project will extend the security research that has previously been performed by utilizing *fuzzing* as a security testing technique and applying it to the most recent HMU.

Bour and Lie were successful in sending SMSs to the Cardiomessenger II-S from a fake base station. Moreover, when reverse engineering the firmware of the Cardiomessenger II-S, Bour obtained references to code in the firmware, which seemed to parse incoming SMS. This opens up for using SMS as an attack vector. We believe that the most recent HMU is also capable of receiving SMS and that it also contains code in its firmware that parses incoming SMS, as were the case for the previous HMUs from the same vendor. Besides, as its name implies, the most recent HMU is capable of 3G communication. This means that the device is capable of communicating using the Transmission Control Protocol (TCP)/Internet Protocol (IP) software stack.

Thus, the Cardiomessenger 3G Smart implements two different communication technologies towards the vendor's backend servers. Initially, we envisioned to fuzz both these interfaces as that would allow us to obtain the most knowledge on the security of its networking capabilities. However, as we deemed such a scope too extensive, we chose to solely scrutinize the SMS interface of the HMU. An important note is that when fuzzing is used as a testing technique, it will happen in a closed environment and not affect the vendor in any way. We are performing our research without crossing ethical boundaries, meaning that we will not do any active probing of the vendor's infrastructure without their consent.

The following two scopes have been defined. They constitute our two primary focuses throughout this thesis project.

Scope 1: The most recent version of HMU, the Cardiomessenger 3G Smart, as a standalone embedded medical device.

Scope 2: The implementation of SMS capabilities within the HMU, considering it as an endpoint in a network.

Given that the Cardiomessenger 3G Smart is an embedded device, the implementations of its communication capabilities are, in fact, the device's cellular modem and the modem's separate firmware - which is developed by another manufacturer. Throughout this thesis, when referring to the implementations of communication capabilities, we will use terminology such as cellular modem, SMS- and data-communication interface, baseband processor and firmware, interchangeably.

Noteworthy, there is a clear link that exists between the scopes above. They are linked in the sense that security research within scope 1, might prove to be

valuable for the security research into scope 2. The idea is that it could become less challenging to perform security research on the implementation of the HMU's SMS capabilities, given that valuable security-related properties are uncovered within the hardware of the HMU itself.

1.3.1 Hypotheses, research questions, and research objectives

To generate new knowledge within our two scopes of the pacemaker ecosystem, we have formulated hypotheses, research questions, and research objectives. The research questions and research objectives were first formulated in our pre-project [KM19], but they were revised and reformulated for this thesis report.

We will work according to two main hypotheses. Each hypothesis relates to one of the scopes that are defined above. For the security testing of the Cardiomessenger 3G Smart as a standalone embedded medical device, we will try to examine whether the following hypothesis is true:

H_1 *The Cardiomessenger 3G Smart HMU contains vulnerabilities as a standalone embedded medical device that can be exploited by an attacker having physical access to the device to compromise a patient's privacy or safety.*

Next, when we examine the SMS interface towards the vendor's backend server, we will try to answer whether the following hypothesis is true:

H_2 *The implementation of SMS within the HMU is insecure and contains vulnerabilities that can be exploited by an attacker determined to compromise either the safety or the privacy of a patient.*

The following research questions are derived to verify our two main hypotheses:

RQ_1 Is the Cardiomessenger 3G Smart sufficiently secured against attackers, and if not, how can the confidentiality, integrity, and availability of patient data be compromised?

RQ_2 Are there any vulnerabilities in the cellular modem's firmware that parses the incoming SMSs?

RQ_3 Is it feasible to use the SMS interface as an attack vector, considering attackers with varying resources?

RQ_4 Are there any security mechanisms in the firmware of the HMU to protect against vulnerabilities in the cellular modem used for SMS communication?

RQ_1 relates to H_1, while RQ_2, RQ_3 and RQ_4 relates to H_2. Furthermore, to answer the research questions stated above, we will pursue the following research objectives:

RO_1 To verify whether the hypotheses in table 1.1 are true for the Cardiomessenger 3G Smart as a standalone medical device.

RO_2 To investigate potential vulnerabilities in the SMS interface of the Cardiomessenger 3G Smart through fuzzing.

RO_1 refers to a set of pre-defined hypotheses. Hence an elaboration is required. These hypotheses are the same hypotheses that were deemed true during the testing of the Cardiomessenger II-S HMUs, which can be found in Bour's thesis [NB19]. Naturally, a successful validation of these hypotheses corresponds to vulnerabilities and a lack of security. We will test whether we can find the same vulnerabilities in the Cardiomessenger 3G Smart as the ones found in the previous versions of the HMUs, when pursuing RO_1. That way, we can describe the evolution and development of security features of Biotronik's HMUs. Validating these hypotheses for the Cardiomessenger 3G Smart will also ultimately validate H_1. Therefore, in table 1.1 they are denoted $H_{1.\mathrm{X}}$.

As previously mentioned, when working towards RO_2, we are extending the security testing performed on HMUs by utilizing fuzzing as a means to discover potential vulnerabilities in the implementations of the Cardiomessenger 3G Smart's SMS capabilities.

1.4 Motivation

Since medical devices are devices that perform functions that alleviate diseases, possibly to a life-saving extent, they indeed lead to a higher quality of life for the affected patients. Patients can suffer from chronic diseases, but by the utilization of a medical device, they can still lead long and fulfilling lives. Medical devices can transform the way we treat individual patients, and since their overall quality is potentially of such a high level, they are most welcome into modern healthcare. Performing research within the field of medical device security is highly motivating.

However, by entrusting medical devices with the responsibility of treating individual patients, several technological and ethical challenges arise. An example of a

Hypothesis number	Hypothesis
$H_{1.1}$	An attacker can open the device, and there is no obfuscation of the electronics that would harden the identification of components.
$H_{1.2}$	Both JTAG and UART interfaces are identifiable and enabled on the HMU.
$H_{1.3}$	The memory is unencrypted, and an attacker can obtain cleartext credentials, debug strings and the data that is sent to the backend server.
$H_{1.4}$	The firmware is unencrypted and not obfuscated, easing the reverse-engineering process for an attacker.
$H_{1.5}$	The use of debug/log strings in the firmware, ease to process of reverse-engineering the communication protocol.
$H_{1.6}$	The AES-key used for encryption of patient data, is hardcoded in the HMU's memory.
$H_{1.7}$	The Access Point Name (APN)'s credentials can be gathered in cleartext from eavesdropping on the communication between the modem and the microcontroller.
$H_{1.8}$	There is no mutual authentication between the microcontroller and the backend server of the vendor.
$H_{1.9}$	Data is sent from the HMU to the backend servers of the vendor over a TCP-service, and credentials are sent in cleartext.
$H_{1.10}$	Credentials used to connect to the APN is reused to connect to the TCP-service.
$H_{1.11}$	A proprietary communication protocol is used to send data to the backend server.

Table 1.1: Hypothesis deemed true by Bour [NB19] for previous versions of HMUs, namely the different versions of the Cardiomessenger II-S.

challenge related to medical devices, which is both technical and ethical, is whether medical devices are sufficiently secured and cannot be exploited to harm the affected patients. This challenge is mainly present for patients whose medical device is implanted into their bodies and not removable. Researching whether various responsible stakeholders meet this challenge is highly rewarding work as it can potentially make a significant difference within the field of medical device security.

If medical devices are not sufficiently secured and can be the very thing that an attacker can make use of to harm patients, it would negatively affect human lives in our digital society. In such a case, one could argue that vendors are primarily focused on making profits, and not the safety and security of the end-users of their products. In the end, the vendors of medical devices are also regular companies that aim to make a profit for their shareholders and are met by the inherent trade-off between security and cost. Most companies do not directly monetize from security

investments. Hence, gaps between security and costs might be present in today's products. Products may be cheap, but if they are insecure, such gaps should be filled. Our research could contribute to revealing this fact. Thus, our work can highlight this issue and contribute positively to securing medical devices.

As mention in section 1.2.1, cyber-attacks that give rise to physical harm are no longer a fictional scenario. It is reasonable to assume that exploiting medical devices to inflict harm on patients is a highly relevant threat. Again, this is particularly relevant for patients whose medical device is implanted into their bodies, as they are left with no choice. These patients simply have to accept the threat level imposed on them because of insecure medical devices, since surgically removing their device leaves them with a significantly worsened health condition.

We feel strongly about the need for secure medical devices and believe that users should be able to trust their device fully. Thus we are highly motivated to research within the field.

1.5 Outline of thesis report

The remainder of this thesis report is structured as follows:

Chapter 2 provides the necessary technical knowledge, in addition to familiarization with security concepts and terminology, necessary for the understanding of the remainder of this thesis report.

Chapter 3 presents relevant research previously conducted within the fields of both pacemaker security and of fuzzing an SMS interface. Our work is primarily based on this research.

Chapter 4 presents and argues for our choice of research methodology, as well as outlines the details of which phases the methodology contains.

Chapter 5 presents the most essential and specialized tools and software utilized within our research.

Chapter 6 presents how we obtained our results and findings from our research within scope 1, and consequently, our research into RO_1. Results and findings themselves are also detailed.

Chapter 7 presents how we obtained our results and findings from our research within scope 2, and consequently, our research into RO_2. Results and findings themselves are also detailed.

Chapter 8 provides a thorough discussion on our results and findings from both

chapter 6 and 7, and what they imply. Techniques that mitigates our findings are also presented.

Chapter 9 concludes this thesis.

Chapter 2
Technical Background

This chapter provides adequate theoretical background knowledge for readers of this thesis. Such background knowledge is necessary to fully comprehend how we conducted our research, in addition to our final contribution related to the security of pacemaker HMUs.

2.1 Security concepts and terms

This section defines essential terminology within cybersecurity that is important to clarify before continuing to read this thesis. The terminology will be used frequently, and to substantiate our claims. Conventionally, in the world of cybersecurity, the terms *Confidentialty, Integrity and Availability (CIA)* are mostly used. An extension includes the *Non-repudiation* and *Authentication* security properties. Definitions provided by the Norwegian government is given below [Reg15]:

Confidentiality entails the protection against disclosure of information such that unauthorized people cannot access it. An example in the context of the pacemaker ecosystem could be that only healthcare professionals should be able to access patient data.

Integrity ensures that a receiver of data can be sure of its legitimacy, meaning that it has not been altered/manipulated in any way by an unauthorized party. A breach of integrity could be that a healthcare professional reads patient data that have been tampered with by an adversary.

Availability ensures that the services and the information are available to authorized personnel whenever needed. A breach of availability could be a DoS-attack on the backend servers of the vendor. If so, attackers successfully drain the resources of those backend servers, yielding the service and information unavailable.

Non-repudiation is closely related to integrity and ensures that someone cannot

deny that an event, a transaction, or that some communication took place. More specifically, it means that a party cannot deny the authenticity of their signature. An example of breaching this property is the logging of data on the programmer on behalf of someone else.

Authentication refers to the process of verifying the identity of a user or process. An example of an authentication breach could be to be able to somehow login as a medical professional without knowing their password.

2.2 Threat model

Threat modeling plays a crucial role in security research. According to the Open Source Web Application Security Project (OWASP), is a procedure defined as: *"To identify, communicate, and understand threats and mitigations within the context of protecting something of value."* [OWAa]. Thus, it is important to understand the threat model for the pacemaker in order to grasp the complexity of potential attacks and to identify adequate security mechanisms. Threat modeling should also be a part of the development of a product, preferably in the earlier stages, such that it can influence the design.

Consequently, we will create a threat model, which will help us to identify and assess possible threats and attacks against stakeholders of the ecosystem. Mitigation mechanisms will be provided in the later chapters once we confirm whether vulnerabilities exist. Furthermore, it is essential to emphasize that we will only outline threats against assets within our defined scopes.

2.2.1 Assets

Firstly, we identify which assets that might be interesting to an attacker. Below is a list of our identified assets:

1. Patient safety.
2. Private patient information.
3. Biotronik's reputation.
4. Biotronik's backend servers.

2.2.2 Threat actors

Next, we present different adversaries that might be interested in attacking the pacemaker ecosystem, and their respective motivations and likelihood. We use the

often classified stereotypes of different threat profiles to distinguish the different threat actors that are relevant for the pacemaker ecosystem [Doe19b], which can be seen in table 2.1 below. In the table, we provide estimates for likelihood based on the corresponding incentives.

Role	Likelihood	Incentives
Organized Crime	Moderate	The main motivation behind why organized crime would be interested in attacking an asset in the ecosystem is monetary gain. Such monetary gain could be achieved through either blackmailing patients/Biotronik, or through selling valuable personal information regarding patients.
Advanced Persistent Threat (APT)	Low	This threat actor is normally engaged in political, economical or military matters. Biotronik itself, or users of their equipment, are thus unlikely targets.
Insider Threats	Low/moderate	Insider threats can have devastating consequences. This threat actor may be motivated by money or they might wish to ruin the reputation of their employer.
Hacktivists	Low	Normally this actor operates to bring attention to an issue, often politically motivated, and target controversial organizations or groups. Hence, Biotronik as a vendor of medical equipment is not a typical target.
Script Kiddies	Low	Script kiddies do not have enough skills do perform such a task nor any incentives do to so.

Table 2.1: The threat model for the pacemaker, including various threat actors, their likelihood, and incentives.

2.2.3 Attacks

There exist multiple hypothetical attacks that an adversary could carry out within the pacemaker ecosystem. If we assume that an attacker has successfully compromised the communication channel between the HMU and the backend servers of the vendor in a passive attack[1], eavesdropping is possible. Such an attack breaches the confidentiality of patient data. In an active attack[2], different Man in the Middle (MitM) attacks are possible. Depending on the absence/presence of mutual authentication and integrity

[1] An attack where the adversary does not interact with the communicating parties
[2] An attack where the adversary makes changes to data an interacts with systems or users

schemes on the communication channel, replay attacks, and modification of data en route is possible. Moreover, merely blocking traffic towards the vendor's backend servers is possible, which will also constitute a MitM attack.

More intrusive attacks are also possible and must be accounted for in the development process. An attacker who has physical access to a HMU might retrieve data from memory, modify existing firmware, or establish a connection to the vendor's backend servers. Data retrieved from memory could, for example, be medical records, encryption keys, PINs, and passwords. Assuming an advanced attacker who can modify firmware, it might be able to install malicious code designed to threaten the patient's safety on the HMU. If, for example, the HMUs are designed with the capability of prompting the pacemaker, a battery-draining attack of the pacemaker is possible by continuously requesting data from it. Lastly, an attacker might be able to access the vendor's private network through the HMU and inflict damage to it.

Finally, given that the HMU receives data, various remote attacks might be possible. If the implementation of the data communication interfaces contains vulnerabilities, a range of different exploits such as memory leaks, remote code execution, and DoS, could be triggered. Ultimately, the attacker could exploit vulnerabilities in the networking interfaces of the HMU to modify the firmware in order to manipulate the pacemaker itself.

2.3 Hardware terminology

Throughout this thesis, we will frequently use terminology which is specific to hardware. Therefore a brief definition of those terms is included below:

1. **Printed Circuit Board (PCB)**, refers to the board that electronically connects different physical components together. The components are soldered onto the PCB and connected by lines and pads that allow for electrical current to flow through.
2. **Microcontrollers** are small computers. This typically encompasses Random Access Memory (RAM), Central Processing Unit (CPU), input/output peripherals and memory. Microcontrollers are dedicated to run one specific application and are not general-purpose devices like microprocessors. The microcontroller of the Cardiomessenger 3G Smart, for example, is dedicated to run the firmware that is developed by Biotronik.
3. **Modem** refers to the component that implements the modulation/demodulation process and converts signals from digital to analog (and vice versa), preparing data for transmission/reception over the air. Modern modems in embedded systems typically implement the cellular software stack.

4. **Firmware** refers to the software that controls the hardware of a vendor-specific embedded system.

5. **Embedded systems** are systems that serve a dedicated function, consisting of a combination of hardware modules and software.

2.4 UART

Universal Asynchronous Receiver/Transmitter (UART) is a hardware technology that enables serial communications between hardware entities. In general, there are two methods in digital communications for transmitting data between components: serial and parallel. In parallel communications, multiple wires/data buses can transmit data simultaneously. In contrast, serial devices communicate by transmitting one bit at a time, using a single wire. It is customary in a microcontroller design to incorporate UART functionality, because it is rather simplistic method for a microcontroller to communicate with other UART-compatible devices. Furthermore, UART is realized through three different pins. These pins are Tx, Rx, and GND (Ground). A typical UART connection between two components is shown in the figure below.

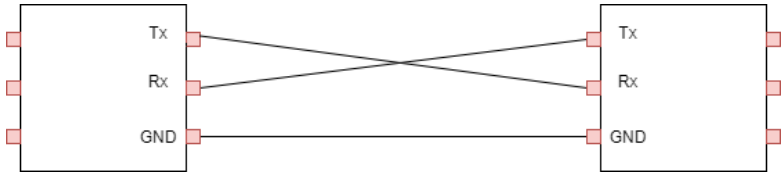


Figure 2.1: UART communication. Data is transmitted on the Tx pin and is received on the Rx pin of the other device.

UART has a simplistic design because of the absence of a clock signal. The communicating entities transfer data asynchronously and rely on start and stop bits encoded in the packet structure to detect correct information, contributing to much overhead. Another concept within UART is the *baud rate*. The baud rate defines the speed at which symbols transfer over the wire, and this rate needs to be equal for both entities to be able to decode correctly. For UART, one symbol is equal to one bit. Therefore the baud rate corresponds to bits per second.

Identifying the UART pins on a PCB can be of enormous advantage for a security researcher. It can provide debugging information, or even provide a means for communicating with chips and components on the board directly.

2.5 SPI

The Serial Peripheral Interface (SPI) provides a full-duplex synchronous method for a microcontroller to communicate with different peripherals such as RAM and flash chips. In comparison to UART, SPI is master-slave oriented and relies on the rising or falling edge of the clock signal to synchronize between the entities, where UART is asynchronous. Thus, SPI enables for less overhead in the communication channel. When the receiver detects a rising/falling edge on the clock wire, it immediately looks for data on the data wire. To be able to communicate over a SPI interface, the entity must have the following wires, illustrated in figure 2.2. The entity which constitutes the master is responsible for generating the clock signal.

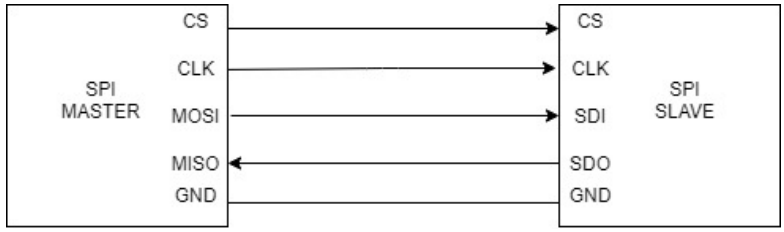


Figure 2.2: SPI communication. Data is transmitted between master and slave entities.

The Chip Select (CS) is used to select the desired peripheral on a shared data bus. It indicates the beginning of a communication session. Hence, the master sends CS at the beginning of data transmission, along with the clock signal. Moreover, data going from master to slave is sent over the Master Out/Slave In (MOSI) wire, whereas data sent in the opposite direction is sent over the Master In/Slave Out (MISO) wire. GND and CLK are the ground and clock signals.

2.6 JTAG

Because of the rapid increase in the complexity of PCBs, verifying, and testing designs have become difficult. Electrical engineers in the late 1980's saw the need for a standard way to verify their designs. Hence, JTAG was developed as the new industry standard by the Joint Test Action Group (JTAG) and later entitled *IEEE Standard 1149.x Standard Test Access Port and Boundary-Scan Architecture* [IEE13].

Essentially, JTAG is the standard architecture for manufacturers to design debugging features and verification mechanisms on embedded systems. It enables a computer to interact with its integrated chips on the PCB directly. By using a technique commonly known as boundary scanning, engineers can test if the con-

nection between chips on the board works as specified in the designs. What JTAG fundamentally does is to provide the developer with the capabilities to write and read single bits directly of pins. JTAG enables, for example, to test whether pin X on chip X has a working connection to pin Y on chip Y.

The test logic and functionality provided by JTAG are realized mainly by three building blocks; the Test Access Port (TAP) controller, the instruction register, and the test data register [MT90]. The TAP contains four (and a fifth optional) pins, listed in the table below. These pins need to be identified on the PCB before a JTAG connection can be made on the Cardiomessenger 3G Smart.

Pin	Acronym	Description
Test Clock Input	TCK	The system clock that synchronizes the time between the various chips on the PCB
Test Mode Select	TMS	The operation of test logic is controlled by this input
Test Data Input	TDI	The test data input that goes in the system
Test Data Output	TDO	The resulting output data from the system
Test Reset Input	TRST	The signal indicating to reset system state

Table 2.2: The TAP pins of JTAG.

Researchers often use a connection to JTAG ports because it yields multiple advantages during a security test. Some handy features it can provide are, for instance, reading and writing of memory, fine-grained control over the CPUs registers without halting the ongoing processes, and provide access to both hardware and software breakpoints on the device under investigation. Using JTAG in security testing relies on its ports not being blocked after production. Even though JTAG is the industry's standard architecture guideline, we will address the debug features commonly associated with this architecture only as JTAG throughout this thesis.

2.7 Short message service (SMS)

Arguably, the most prevalent text messaging service in telecommunications has, for many decades, been the SMS technology. This technology was presented during the development of the GSM series of standards. Simply put, SMS is a protocol for exchanging simple messages between entities on a mobile network. It traverses intermediary components in a GSM network on the way to the receiver. At last, the

SMS reaches a Short Message Service Center (SMSC), which stores the SMS and ensures successful delivery to the receiving mobile station.

2.7.1 The SMS formats

Different formats of SMS exist to support exchanges of messages between different entities. The two message formats that are most relevant for this thesis are the SMS_DELIVER and SMS_SUBMIT format. These message formats are used depending on whether the SMS is mobile-terminated or mobile-oriented [3rd02]. As the name implies, SMS_DELIVER is used when a SMS is delivered from the SMSC to the mobile station. SMS_SUBMIT is therefore used when the message is sent from a mobile station and forwarded to the SMSC. In the SMSC, the message changes format, from SUBMIT to DELIVER, and is then forwarded to its destination. SMS responses, which have a different encoding, are then generated by the recipient. These are based on the two message formats described above and are similar in structure. All SMS messages are sent within the SMS protocol stack.

2.7.2 The SMS protocol stack

The SMS protocol resides within layer 3 of the GSM protocol stack. It is, in fact, a sub-protocol of the Connection Management (CM) protocol, which in turn, is one of the three main protocols that constitute the third and topmost layer in GSM. The other two being the Radio Resource Management (RRM) protocol and the Mobility Management (MM) protocol. We are only concerned with layer 3 of GSM protocol stack, and more specifically, the SMS protocol within the CM protocol. The simplified figure below depicts the GSM protocol stack, including only the relevant protocols in layer 3 for our project.

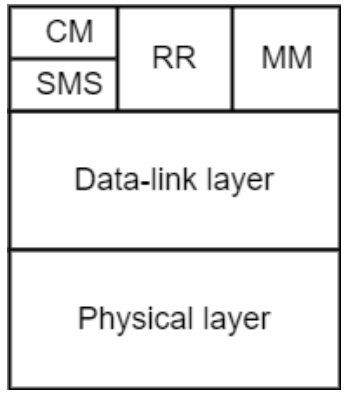


Figure 2.3: A simplified GSM protocol stack, including only the relevant protocols in layer 3 for our project.

Furthermore, the SMS-protocol itself divides into three separate sub-layers. These are, in descending order, the Short-Message Application Layer (SM-AL), Short-Message Transfer Layer (SM-TL) and the Short-Message Relay Layer (SM-RL). These layers of the SMS protocol are included in figure 2.4 below. The SM-RL layer exists to hold state between delivered SMSs and their acknowledgments. It also ensures error reporting. SM-TL manages transfer information and contains information that determines how the SMS should be interpreted. The SM-AL interfaces with the different applications.

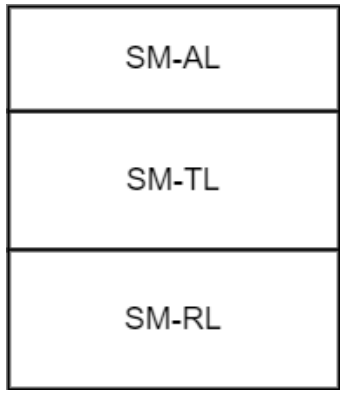


Figure 2.4: The three layers of the SMS protocol in GSM.

When an SMS_DELIVER message first arrives in a mobile station, it is transmitted from the SMSC as CP-Data, which is a message within the CM protocol. It is encapsulated in this header to counter the losses caused by changing the dedicated channels and ensures successful delivery. This message contains an Relay Protocol Data Unit (RPDU), which is interpreted by the SM-RL layer in the SMS protocol. In turn, the RPDU contains a Transfer Protocol Data Unit (TPDU), which is interpreted by the SM-TL layer of the SMS protocol. The payload of the TPDU can then be interpreted in the SM-AL by an appropriate application, depending on the use case [3rd00].

2.7.3 Fields in the SMS_DELIVER format

We are mostly concerned with the SMS_DELIVER message format in this project. The reason for this is that the messages that originate in the network must have this format. Hence, solely SMS_DELIVER is further described. Table 2.3 is taken from the original GSM specification [3rd02] and shows the fields of a SMS_DELIVER message.

Abbr.	Reference	M/O	Representation	Description
TP-MTI	TP-Message-Type-Indicator	M	2b	Parameter describing the message type.
TP-MMS	TP-More-Messages-to-Send	M	b	Parameter indicating whether or not there are more messages to send.
TP-RP	TP-Reply-Path	M	b	Parameter indicating that Reply Path exists.
TP-UDHI	TP-User-Data-Header-Indicator	O	b	Parameter indicating that the TP-UD field contains a Header.
TP-SRI	TP-Status-Report-Indication	O	b	Parameter indicating if the Short Message Entity (SME) has requested a status report.
TP-OA	TP-Originating-Address	M	2-12o	Address of the originating SME.
TP-PID	TP-Protocol-Identifier	M	o	Parameter identifying the above layer protocol, if any.
TP-DCS	TP-Data-Coding-Scheme	M	o	Parameter identifying the coding scheme within the TP-User-Data.
TP-SCTS	TP-Service-Centre-Time-Stamp	M	7o	Parameter identifying time when the SC received the message.
TP-UDL	TP-User-Data-Length	M	I	Parameter indicating the length of the TP-User-Data field to follow.
TP-UD	TP-User-Data	O	(*)	User data

Table 2.3: The fields in the SMS_DELIVER format.

In table 2.3, `M` stands for mandatory, indicating that the field is required. `O` stands for optional, indicating that this field can be omitted. The different values of the Representation column are: `I` is an integer, `b` is a bit, `2b` is 2 bits, `o` is an octet, `7o` is 7 octets, and `2-12o` is 2-12 octets. Lastly, the user data length (`TP-UDL`) is dependent on the coding scheme declared in `TP-DCS`.

2.7.4 SMS modes

The SMS_DELIVER message format can be sent to the modem of a device in two different modes. That is, through ATtention (AT) commands, which is referred to as *text mode*, or through *Protocol Data Unit (PDU) mode*. Only PDU mode is relevant for this thesis, meaning that AT command mode will not be explained further. The PDU mode is a hexadecimal encoded message format and is more flexible compared to text mode as it can be used to express binary and compressed data as well.

As previously mentioned in 2.7.2, within the SMS protocol, different PDUs exist in the different layers. A TPDU in the SM-TL layer of the SMS protocol typically looks like the hexadecimal string below. The TPDU contains default values on all fields and is set with sender address, recipient and payload to "12345678", "87654321" and "Hello World", respectively.

0008912143658700000230323152720 00BC8329BFD065DDF723619

Based on table 2.3, we can dissect the TPDU above and map the hexadecimal to the header fields of the SMS_DELIVER message format. Figure 2.5 shows the

exact mapping between header fields and the hexadecimal TPDU. The first octet is `0x00` and represents that this TPDU is of type DELIVER. It is an ordinary SMS without extra features and no user data header. Next, the six following octets are the address of the sender, corresponding to "12345678" in the GSM default 7-bit encoding. `TP-Protocol-Identifier` and `TP-Data-Coding-Scheme` have default `0x00` values, which are followed by 7 octets of the `TP-Service-Center-Time-Stamp`. At last, there is the octet corresponding to the length of the user data and the actual payload.

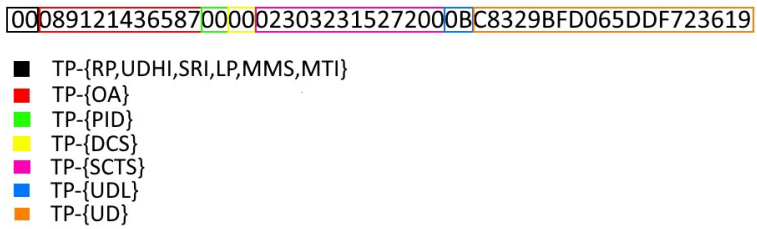


Figure 2.5: The dissected TPDU containing the payload "Hello World". Mapping between hexadecimals and header fields of the SMS SMS_DELIVER message format.

2.8 Fuzzing

Fuzzing is a testing technique that involves injecting malformed or semi-malformed data into a target system in an automated fashion. Examples of targets can be software applications, protocols, and file-formats. The idea is that injecting random input will cause the behavior of the target system to deviate from what is usually the case. Thereby, it is possible to conclude that a particular input leads to particular abnormal behavior. This will correspond to having detected a bug in the target system, as it does not handle such input appropriately. The main advantage of fuzzing a system is that it is relatively easy to implement, as it does not require in-depth knowledge of the system's internals to uncover vulnerabilities. Also, due to its random approach, fuzzing often uncovers vulnerabilities that are missed by developers and engineers. Because fuzzing uncovers vulnerabilities that developers often miss, researchers often incorporate it into the development process of a system.

Since researchers commonly employ fuzzing on closed systems, a significant challenge is related to the assessment of how a given input affected the target. Overcoming this challenge entails that some means of monitoring the system while fuzzing has to be part of the fuzzer framework. Thereby, knowledge can be inferred from how the system behaved. The need for monitoring capabilities is indeed a limitation of fuzzing, as it is commonly the most difficult to implement within a fuzzer framework [OWAb].

2.9 3G jamming - downgrade attack

An attacker's motivation behind a downgrade attack is based on the fact that older mobile networks are generally less secure than newer versions. In GSM, for example, there is no mutual authentication between mobile stations and the network. Only the network authenticates the mobile station, not visa versa. This means that mobile stations will connect to an attacker's fake GSM network if the attacker performs jamming on other networks in the area. Thereby, the attacker can force a mobile station to use the GSM network instead of newer versions of mobile networks, such as 3G and Long Term Evolution (LTE). Since the different generations of mobile technology utilize different parts of the radio spectrum, an attacker has the opportunity to jam a network's frequency range. By transmitting noise to those channels, an attacker can effectively deny all communication on those frequencies. Because mobile stations have the inherent design issue of supporting older networks, if no connection to a new network is possible, mobile stations will connect to the GSM network if it currently provides the strongest signal to a mobile station.

Chapter 3
Related Work

This chapter familiarizes readers with related research that has previously been carried out. Both research related to the security of the pacemaker ecosystem and fuzzing of mobile communication interfaces are presented.

3.1 The pacemaker ecosystem

The first published security assessment of pacemakers, and their ecosystem, was done in 2008 by D. Halperin et al. [HHR+08]. The researchers demonstrated how to partially reverse-engineer the wireless communication protocol between the pacemaker and the programmer, using tools such as software-based radios and oscilloscopes. Their work disclosed insufficient privacy and security properties of the communication protocol. Moreover, they implemented several attacks against a pacemaker and demonstrated that confidentiality, integrity, and availability mechanisms were either broken or non-existing. Lastly, they introduced new mitigation techniques aiming to improve the security of pacemakers.

More recently, in a paper from 2016, Marin et al. [MSG+16] described how they fully reverse-engineered the proprietary wireless communication protocol between a pacemaker of the latest generation and its programmer. They realized this by using only Commercial Off-The-Shelf (COTS) equipment and by following a black-box testing methodology, implying that a weak adversary with limited resources could perform similar work. Next, they demonstrated privacy-compromising and DoS attacks against the pacemaker, among which were replay and spoofing attacks. The authors emphasize that security-by-obscurity is an alarming design paradigm, and they advocate for the industry to migrate to standard protocols that are well-examined and approved by security experts. Furthermore, they proposed both short-term and long-term countermeasures.

Also, in 2016, the short-selling firm Muddy Waters Capital LLC published a report regarding vulnerabilities in the pacemakers and HMUs manufactured by St.

Jude Medical (now Abbot), in the U.S. [Blo16]. The research was carried out by the cybersecurity research firm MedSec Inc. The vulnerabilities found could be exploited in several ways, among which included a battery-draining attack of pacemakers, as well as forcing pacemakers to pace at a dangerous rate. These attacks were possible due to vulnerabilities in the St. Jude Medical HMU, which was, in turn, used to compromise pacemakers. There have been no reports of these attacks being actively carried out, and they were only demonstrated in a lab. Muddy Waters and MedSec also argued that St. Jude Medical was indifferent to cybersecurity, and put profits over patients. At the time, St Jude Medical had deployed 260,000 HMUs in patients' homes. Thus, the research by Muddy Waters and MedSec had a severe impact. Consequently, St. Jude Medical issued a firmware update, which was reviewed by the U.S. regulatory authority, at the beginning of 2017. The firmware update patched the vulnerabilities in their HMU [A$^+$17].

In 2017, the cybersecurity firm Whitescope performed an exhaustive security analysis and evaluation of the pacemaker ecosystem. The researchers, Rios and Butts [RB17], had acquired multiple devices from four major pacemaker vendors and revealed vulnerabilities in all of them. Their findings implied that the architecture, and the implementation of the devices, are susceptible to hypothetical cyberattacks that could breach the confidentiality, integrity, and availability of the pacemaker ecosystem. Some of the discoveries they made were related to vulnerabilities in third-party software, lack of authentication mechanisms between devices, unencrypted file systems and firmware, removable hard-drives, and unsigned firmware. Rios and Butts point out that the whole industry is quite immature in its workings with cybersecurity, and therefore has a challenge with securing its ecosystem against cyber threats. Moreover, the authors stress the fact that similar vulnerabilities existed in devices from different vendors, implying industry-wide cross-pollination. This implies that an attacker might leverage an already identified vulnerability in the products of one specific vendor when carrying out attacks on a different one. Lastly, the author's main contribution is a list of questions regarding cybersecurity designs that should guide developers of pacemaker ecosystems to implement fully secure systems. These questions might prove useful within our research as we wish to test the latest generation of the HMU.

NTNU and SINTEF have collaborated and researched a range of Biotronik's pacemakers and their ecosystem for multiple years. These contributions have laid the foundation of work that is necessary for us to start working within the field. Their work has left us with a broad range of vendor-specific information regarding security properties, and our project constitutes yet another contribution within the collaboration between NTNU and SINTEF.

The first contribution of their collaboration project is a master thesis done by

Kristiansen et al. [KW18] in 2018. They examined the security level of one specific pacemaker programmer and revealed several vulnerabilities that could potentially breach confidentiality, integrity, and availability. In short, the vulnerabilities can be placed in the following categories; the operating system, lack of physical access defenses, authentication, commercial third-party software, data export and import, and encryption.

In 2019, Bour [NB19] examined the security level of several HMUs from Biotronik. In his work, he showed that a physical compromise of the HMU could be used as an attack vector against the pacemaker, potentially harming a patient's life. He realized this by using COTS equipment and open-source software. His work, in particular, will serve as a baseline for research within our first scope.

Lastly, in her master thesis from 2019, Lie [WL19] investigated the security properties of the communication link between the HMU and the vendor's backend servers. Her research showed multiple weak implementations and insufficient security mechanisms that could impact a patient's safety and privacy. However, her results show that Biotronik has made some effort to implement basic security mechanisms and that their security level tends to increase with newer generations of HMUs. This work will also serve as a baseline for our research.

3.2 SMS-fuzzing and mobile communication security

Some research papers are of greater interest for this thesis than others, as they might share a similar methodology, use the same technology, or have similar setup and environment.

One such example is the work of Mulliner et al. [MGS11] from 2011. The authors investigated the security of feature phones, and the possibility of a large scale attack through exploiting potential vulnerabilities in the SMS-client software. Initially, this research started as a master thesis, which was published in the same year [Gol11]. The master thesis contains a more in-depth explanation of the implementation. In the paper, however, a novel approach to test closed-system implementations is described. It encompasses sending fuzzed SMS messages over a software-based GSM network that is enhanced with monitoring capabilities, for the device under investigation. The monitoring capabilities are based on response messages from the handsets. It is noteworthy that the framework the authors propose is platform-independent and can be used for all devices that are capable of receiving SMSs. Parts of this work will constitute a foundation for our thesis. A plethora of different bugs was discovered that could lead to sophisticated attacks against mobile users. Among the vulnerabilities that they discovered was the opportunity for an attacker to send malformed SMSs to receivers that could result in the phone disconnecting from the

mobile network, forcing reboot or white-screening the device. Conclusively, these SMSs could effectively carry out DoS-attacks. These messages could also be sent over a legitimate mobile network. Since the testing was based on automated test generation (fuzzing), it can be applied to many use-cases. The testing generalizes well and therefore suits a system such as the HMU.

Weinmann [Wei12] conducted similar work, but rather than focusing on the SMS-interface, the entire cellular baseband stack of widely deployed phones was scrutinized. Reverse engineering of firmware was performed to discover implementation flaws. In this research, the same setup and environment were used as in Mulliner's experiments. The setup involved setting up an illegitimate base station and sending malicious payload in the form of layer three messages within the GSM protocol. In the modern architecture of smartphones and IoT devices, it is not uncommon to run baseband stacks and applications on separate processors. This is also true for the Cardiomessenger 3G Smart. Baseband stacks typically run on baseband processors, which has its own Operating System (OS) and memory. This paper demonstrated the risk involved with having vulnerabilities residing in the baseband processors. The authors managed to remotely launch several attacks exploiting memory corruptions in the baseband firmware, completely compromising the integrity of the handsets. Their work shed light on how bugs in the baseband processor could impact the security of the entire device, resulting in extensive code reviews of many manufacturers. This work shows the importance of hardening all segments of a device to ensure security.

Another fuzzed-based approach to SMS security testing was proposed by Mulliner et al. [MM09b] in 2009. They altered the respective operating systems for Windows, Android, and Apple phones such that they were able to inject fuzzed SMSs directly without traversing a mobile network. Their fuzzing lead to the discovery of multiple bugs in smartphones. Some bugs triggered system crashes and could be exploited in a DoS-attack, whereas others could potentially be used in remote code execution. When fuzzing the devices, the authors leveraged that they had full access to debug tools and system logs to monitor. Unfortunately, to the best of our knowledge, we do not have the same access to the debug features on HMU, which makes this fuzzing approach impossible.

Finally, an exciting contribution to the field of GSM security and SMS-fuzzing is the master thesis by Hond from 2011 [Hon11]. He utilized specific tools and developed a framework capable of fuzzing several parts of the GSM protocol stack in mobile phones, among which were SMS. The goal of his research was to provoke strange behavior in the receiving mobile phones, due to the reception of fuzzed SMSs and call control messages. This was indeed successful, as fuzzed SMSs caused several phones to reboot. Given that it would be interesting to provoke similar strange behavior in the HMU, the approach to SMS-fuzzing by Hond is highly interesting.

Chapter 4
Methodology

This chapter will present and detail our choice of research methodology, as well as elaborate on why our research methodology is suitable for our research. The methodology itself will serve as a baseline in decision-making and provide a way to structure our work.

4.1 Preliminary considerations

Our research into the security of Biotronik's Cardiomessenger 3G Smart HMU, is divided in two. Referring back to section 1.3, we have determined one scope considering the HMU as a standalone embedded medical device (scope 1), and another considering the implementations of its SMS capabilities towards the backend servers (scope 2). These scopes are, however, not entirely distinct. Research into scope 1 might provide valuable input into the research of scope 2, as it can reveal valuable properties of the HMU's SMS interface. Due to the semi-distinct scopes, it is reasonable to consider that a single research methodology cannot be sufficient for the entirety of our master thesis project, but instead, that two methodologies can benefit from each other.

The Cardiomessenger 3G HMU is a proprietary device, and there exists little available information regarding its technical specifications. The only information revealed by the device's user-manual is necessary information related to how properly use the device, and not information about its inner workings[1]. Specifically, no information about the HMU's security, or implemented security mechanisms, is obtainable from the vendor. This is commonly the case for most electronic products, however, it is particularly problematic for medical devices.

We have also defined a research objective for each scope, namely RO_1 and RO_2. When working towards RO_1, the only vendor-specific knowledge we possess is that

[1] `https://manuals.biotronik.com/emanuals-professionals/?country=US&product=HomeMonitoring/CardioMessenger/CardioMessengerSmart_US`, accessed on 18.02.2020

produced by Bour [NB19] and Lie [WL19]. However, this knowledge is valid only for the previous versions of HMUs from the vendor. Thus it is not directly transferable to the most recent version. Nonetheless, it will serve as a basis as we set out test whether the hypotheses in table 1.1 are also valid for the Cardiomessenger 3G Smart, as RO_1 states. When working towards RO_2, we do not possess any vendor-specific knowledge. We do, however, possess general knowledge of how to use fuzzing as a means to uncover potential vulnerabilities in the SMS interface, as RO_2 states.

To us, the security level of the Cardiomessenger 3G Smart itself, or the security of its implementation of the SMS technology, is unknown. Thus, we consider the device to essentially conform to a black-box. Therefore, we need a research methodology suitable for this kind of challenge. Our research methodology must provide a structured way to carry out our work since we know so little about the HMU, initially.

4.2 Black-box testing

Black-box testing is a testing approach that concludes on the behavior of a system based on the relationship between specific inputs and their corresponding outputs. What specifically happens internally within the system is not of interest. A given input can, for example, cause the system to output an unexpected value, which corresponds to unpredictable behavior provoked by that input. Another example is if a given input causes the system to crash. The absence of output can be considered an output in itself, and the conclusion could then be that the input provoked a system crash.

Figure 4.1: Visualization of Black-box testing.

There exist several different techniques that we can employ within black-box testing. Some of these techniques are highly suitable for a hardware device like the HMU, where no knowledge regarding its technical specification, or security, is obtainable. Therefore, our two research methodologies constitute a selection from the black-box testing methodology, where we have chosen to employ the most suitable testing techniques from within the methodology. The chosen testing techniques will differ depending on which research objective we are currently working on. For RO_1, we will be using black-box hardware testing, while for RO_2, we will make use of black-box fuzzing, as our testing techniques.

4.2.1 Black-box hardware testing

For RO_1, the black-box hardware testing is structured in different phases. Each phase is intended to gradually push our research forward, and ultimately uncover vulnerabilities in the hardware of the Cardiomessenger 3G Smart. The phases in black-box hardware testing are depicted in figure 4.2, and are detailed in the following.

Hardware analysis

Given that we have physical access to the HMU, the first phase is naturally to analyze and inspect the device itself. The motivation behind thoroughly examining the device is that we might be able to determine various properties of the device. Determining such properties could, in turn, provide an insight that forms a basis for further hardware analysis. For example, if we can determine that is is possible to open the device, we gain access to its PCB. Having access to the PCB yields the possibility to identify components such as a microcontroller, modem, and RAM. Also, if we acquire access to the PCB, we might discover labels identifying debug ports such as JTAG and UART, which are typical for embedded systems. Indeed, a hardware analysis of the HMU is the natural first phase of our hardware testing, but it is also a critical phase. Without knowledge regarding the hardware itself, it becomes virtually impossible for us to acquire a basis for further research into the device. There is simply no way for us to assess the security of the device itself without knowing more about how it functions on a technical level.

Investigation

The investigation phase consists of building upon our acquired knowledge from the hardware analysis. In this phase, we further investigate what the hardware analysis revealed. By obtaining some form of documentation of identified hardware components, we will necessarily acquire more knowledge concerning how the device functions, and which security mechanisms are implemented. From there, we can deduce how the hypotheses in 1.1 can be tested. For example, if we identify a typical microcontroller on the PCB, we might be able to acquire its datasheet. Such a datasheet will necessarily contain a high volume of technical information related to how the microcontroller functions. Any such information might aid us in deducing testing scenarios.

Hypothesis testing

In the hypothesis testing phase, we carry out a test aiming at validating one or more of the hypotheses in table 1.1, based on our preliminary findings from the two previous phases. Thereby, we rely on the previous two phases to have returned something interesting, worth putting to the test. Furthermore, we must construct

a test that ensures that the hypothesis under test is indeed tested. If not, we will not achieve reproducible findings. The outcome of a hypothesis test is either true or false. Regardless of the result, either outcome is of interest. If the hypothesis is validated, then we will consequently have produced a finding. If not, and we are fully confident in having carried out the hypothesis test successfully, we will also have produced a finding. Also, the work carried out throughout a hypothesis test might yield further insight into other aspects worth investigating further.

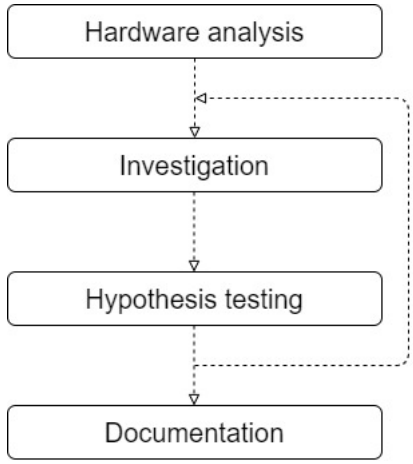


Figure 4.2: The different phases of black-box hardware testing, and how they are linked.

Documentation

In the documentation phase of the black-box hardware testing, we write up our findings in the appropriate result chapter in this thesis report. As previously mentioned, before doing so, we must ensure to have produced undeniable results. Also, it is worth pointing out that when working in the documentation phase, it is essential to present our findings in a structured and understandable manner. In this way, it becomes clear how others, for example, the vendor can reproduce our findings and verify our contributions within the field.

4.2.2 Fuzzing

When working towards RO_2, we employ fuzzing as our black-box testing technique. Fuzzing is a testing technique which in and of itself, considers its target to be a black-box. Therefore, it is a highly suitable approach for testing the security of the HMU's SMS interface. However, each fuzzing case is somewhat unique, and we cannot utilize particular types of fuzzing tools out-of-the-box. The fuzzing case is dependent on which system is to be fuzzed, as well as which interface of that system.

Therefore, we are required to build our own fuzzing framework, specifically targeting the Cardiomessenger 3G Smart, and its SMS interface. In our case, the fuzzing framework consist of a suitable architecture, some means of input generation, an executable computer program that forwards the input to the HMU, and some means of monitoring capabilities. Also, we are required to analyze the results and properly document them. To develop a fuzzing framework, we will carry out the phases of fuzzer development and execution, depicted in figure 4.3. The phases are further detailed in the following.

Architecture setup

Firstly, we need to realize some mechanism which enables us to provide input to the interface that is to be fuzzed. This entails a setup of an appropriate architecture of components and configuring them to work together. The goal is to automate the process of sending input to the HMU. As we wish to fuzz the SMS interface of the HMU, our architecture must include a controllable access point to which the HMU connects. This access point will enable us to forward semi-malformed and malformed data to the HMU. Since an access point implies that there is a network in place, we are also required to simulate parts of a mobile network to ensure that the HMU indeed connects to our access point. Furthermore, we must provide the strongest signal available for the HMU. Therefore, a jammer will also be part of our architecture.

Establish monitoring capabilities

The next phase in our fuzzer development is establishing some means of monitoring. Indeed, we are dependent on acquiring knowledge regarding how the HMU reacts to receiving the malformed input. If not, there is no way for us to conclude that a specific input leads to a particular output. There are several ways of implementing monitoring capabilities, and it can be implemented at different endpoints. We can either implement monitoring capabilities at the receiver's edge (the HMU), or at the sender's edge (the access point, or simulated network). Both of these options have their pros and cons. The superior option would be to have monitoring capabilities at the receiver's edge since this would yield the most sincere results regarding what is happening inside the HMU when it receives an input. However, this option is more difficult to realize than its counterpart as it would require control of both endpoints in the communication. Implementing monitoring capabilities at the sender's edge is, by far, the most comfortable option, as a lot of network monitoring software exists. However, this implementation of monitoring might not provide as detailed output as its counterpart because it would be based on response messages triggered by the input SMSs.

Input generation

Thirdly, if our fuzzing framework is to be successful, we are dependent on some means of input generation. This involves the development of a computer program, which, in our case, generates the test cases. The input generation is a crucial phase in the development of our fuzzing framework, as the outcome of the fuzzing experiment is directly dependent on the test cases. It is also important to emphasize that we can conduct input generation at different levels of intelligence. In the most unintelligent case, input generation is only generating completely random input and forwarding it to the HMU. While this approach could yield results, we base our input generation on the fact that we know which communication technologies the HMU is capable of utilizing. Since the incoming SMS interface is our target, we can generate input in a much smarter way, addressing the SMS technology specifically. Then, input generation would involve generating input in a semi-random way which considers how the SMS packet is constructed, and which header fields that should be mutated. Within our framework, we aim for the latter semi-random approach.

Analysis

Naturally, performing an analysis of the output from our monitoring implementation is necessary. This phase is carried out after an iteration of fuzzing is performed. The analysis phase might uncover vulnerabilities in the HMU's implementation of the SMS interface. Another outcome of the analysis phase might be some output which is not a result in itself, but worthy of further investigation. If so, this initiates a new iteration of fuzzing, where the input in the next iteration is a slightly mutated test case based on the previously provided input. The output can also indicate the presence of a bug, when there is, in fact, no bug present. The analysis face is, therefore, vital when validating our findings. Therefore, it is reasonable to claim that it is within the analysis phase that new knowledge is generated.

Documentation

Lastly, as for the black-box hardware testing, the final phase is proper documentation of our results. Again, it is essential to ensure that those results are undeniable before documenting them, as it must be completely transparent and reproducible by others. Therefore, it is of great importance to document the results in an understandable and structured manner. Only then can the affected vendor confirm that their product contains a vulnerability that might affect the users, and consequently patch it.

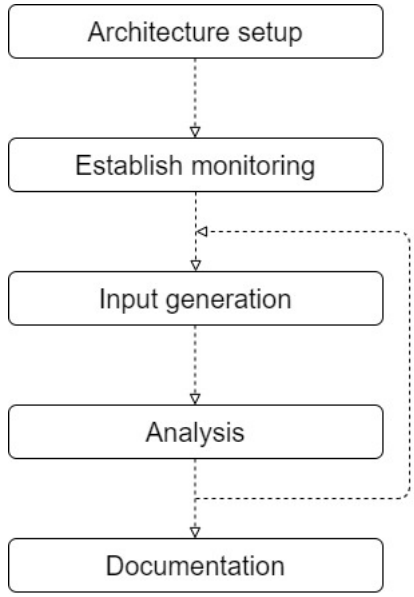


Figure 4.3: The different phases of black-box fuzzing, and how they are linked.

4.3 Intersection between research objectives

As mentioned in the introduction, this thesis centers on examining two research objectives. In the context of that, two different methodologies are created and applied, which is described in detail above. However, these methodologies are not entirely distinct. They are complementary, and the fuzzer-methodology dramatically benefits from the output of the hardware testing. Specifically, findings and information from the black-box hardware testing can have enormous implications for the fuzzer development. For example, if the hardware testing reveals that we can obtain a privileged execution mode on the device, debugging can be conducted at a higher level on the device and thus provide detailed output. Generally, any valuable input to the fuzzer development from the hardware testing will reduce the "size of the black box". In other words, depending on the outcome of RO_1, we are moving towards more of a white-box testing scenario in RO_2.

4.4 Limitations of methodology

Although black-box testing is the most suitable methodology for our hardware testing and fuzzing, black-box testing comes with several limitations. We must be aware of these limitations when conducting our work as it can aid us in thinking critically regarding the results we obtain.

Related to the black-box hardware testing, the main limitation of the methodology is the fact that certain properties of the device might remain uninvestigated. This is because, to us, the device is a black-box. If we do not find something that indicates the presence of an interesting property, which in turn could lead to a test which validates one or more hypotheses from table 1.1, we will simply not look into it. This does not automatically mean that the property is not present on the device. The case can be that the vendor has successfully implemented mechanisms that obfuscate the property, making it unnoticeable. Another alternative is that the vendor has implemented sufficient security mechanisms that remove the property altogether. There is no way for us to tell confidently.

The limitations of black-box fuzzing is strongly related to which monitoring capability we can implement. If we are not able to implement monitoring capabilities at the edge of the receiver (the HMU), we are left with a less detailed error log. In turn, this implies that certain information related to vulnerabilities could remain undisclosed. One example is if a system crash is triggered in the HMU by a fuzzing test case, the software bug that produced the system crash remains unknown. We can only infer its presence. From the network point of view, we cannot fully detail why the system crashed. It has to be based on either a response code produced by our test case, or the absence of response altogether.

4.5 Research guidelines

Although not a specific phase in either of our methodologies, we will adhere to a few ethical guidelines throughout our research. When we conduct experiments within our defined scopes, this will happen in a closed environment. By doing so, we ensure that we are not in conflict with the vendor's infrastructure or interfere with the treatment of a patient. Since the acquired HMUs has once been active devices in patients' home, we will systematically redact any sensitive information we might find. Furthermore, any offending SMS that might cause trouble to the modem in scope 2 will also be redacted.

If we obtain findings which we believe to be severe, we will responsibly disclose our findings to the affected vendor. This will be done in a coordinated vulnerability disclosure process, in accordance with ISO/IEC 29147:2018 [ISO18]. Biotronik might not be the only affected vendor, as components within the HMU are possibly manufactured by other vendors. This is indeed the case for the HMU's modem.

Lastly, given that we wish to contribute to the community that is researching cybersecurity in medical devices, we will publish our code and fuzzer framework, making it available for future researchers to utilize. The code will be open source and available to anyone.

Chapter 5
Tools and software

This chapter presents the tools and software utilized in our research. Being familiar with these tools and software is important for grasping how we carried out our work. The tools and software are structured based on whether they are relevant for RO_1 or RO_2. That is, they are structured based on whether they are relevant for the black-box hardware testing, or the fuzzing of the HMU.

5.1 Black-box hardware testing

OpenOCD

OpenOCD (Open On-Chip Debugger) is a free, open source software project, developed by Dominic Rath in 2005[1]. Since then, the software has grown into a rich open-source project, and its latest release was in 2017. OpenOCD is dependent on a debug adapter, which in turn has to support a debugging protocol. This is what will enable a connection to the debug ports on the actual hardware. Thankfully, OpenOCD supports a wide range of JTAG adapters, which in turn supports the JTAG protocol, which can be used for boundary scanning and debugging of hardware. Also, OpenOCD supports a variety of different microcontrollers, through various included configuration files. Thereby, OpenOCD can be applied to several different hardware devices.

Raspberry Pi Zero as JTAG and SPI adapter

The Raspberry Pi Zero is a fairly cheap single-board computer released in 2015[2]. It is highly convenient for several different use cases. In order to access the device, one connects to the wireless network it sets up and makes use of an SSH-client on an ordinary laptop to connect to the device. In our research into RO_1, we made use of a Raspberry Pi Zero as JTAG adapter, as the device supports the JTAG protocol. To

[1] `http://openocd.org/doc/pdf/openocd.pdf`, accessed on 03.02.2020
[2] `https://www.raspberrypi.org/products/raspberry-pi-zero/`, accessed on 02.03.2020

achieve a JTAG connection using the Raspberry Pi Zero, correctly mapping the pins of the device is important. The left table below shows which pins on the Raspberry Pi Zero that corresponds to which of JTAG's TAP pins. The Raspberry Pi Zero also supports the SPI protocol. Therefore, also within our research towards RO_1, we made use of the device as an SPI-adapter. Again, this depends on a correct pin mapping on the Raspberry Pi Zero. Below to the right is a table that shows which pins on the Raspberry Pi correspond to which SPI pins, as well as a photo of the Raspberry Pi Zero.

RPi Zero pins	JTAG TAP pins
24	TRST
23	TCK
22	TMS
21	TDO
19	TDI

RPi Zero pins	SPI Flash pins
25	GND
24	CS
23	SCK
21	DO
19	DI
17	VCC 3.3V

Table 5.1: The Raspberry Pi pinout for both JTAG and SPI connection.

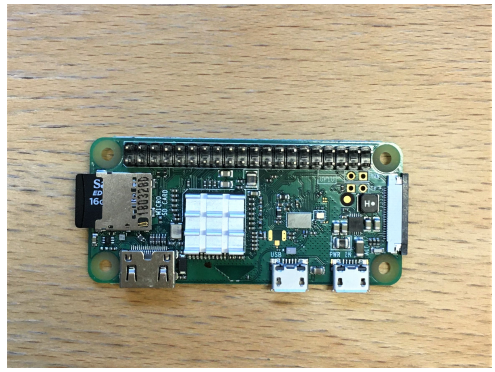


Figure 5.1: The Raspberry Pi Zero.

PCBite

The PCBite by Sensepeek[3] is a very useful piece of equipment. The PCBite effectively replaces soldering as a means to physically connect to hardware devices. Soldering requires a specific skillset and is difficult to reverse. The PCBite, on the other hand, is easy to use. Anyone can set up a connection to hardware debug ports in a matter

[3] `https://www.sensepeek.com/`, accessed on 25.02.2020

of minutes, and the connection easily tears down. The PCBite works by placing its flexible yet steady probes onto the hardware debug ports. The probes will remain in their position, which ensures connectivity.

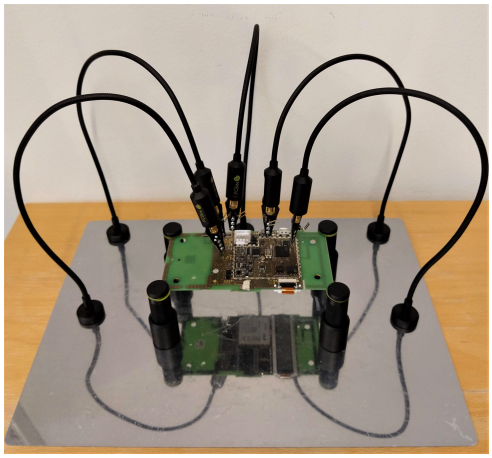


Figure 5.2: The PCBite. By attaching wires to the probes, a physical connection to the hardware is realized.

Flashrom

Flashrom is a software utility that can perform identification, reading, writing, erasing, and verification of flash chips[4]. It comes with support for several chips, among which are several different SPI flash chips. It is a highly convenient tool for a black-box hardware test, as SPI flash chips are quite common. It is easy to install and works in an out-of-the-box fashion, given that the software supports the flash chips under test. Flashrom is dependent on a stable SPI connection to the flash chips under test.

JTAGulator

The JTAGulator is an open-source hardware hacking tool. Its primary function is to assist in identifying debugging interfaces from various test points on a device[5]. The idea is that when a candidate debugging interfaces is located, the JTAGulator can be used to test which pins of that debugging interface correspond to the five TAP pins of JTAG. It does so by trying all possible permutations, so the correct pinout will be determined if the candidate interface was indeed the JTAG interface. The JTAGulator can be seen below.

[4] `https://flashrom.org/Flashrom`, accessed on 04.03.2020
[5] `http://www.grandideastudio.com/jtagulator/`, accessed on 04.03.2020

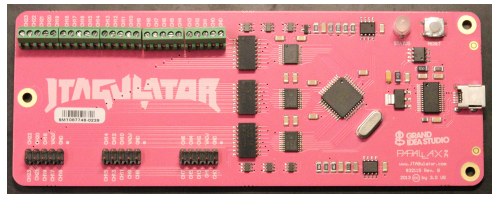


Figure 5.3: The JTAGulator.

Shikra

The Shikra is a hardware tool manufactured by Xipiter intended to provide an interface between a USB-port and several other lower-level data interfaces[6]. The Shikra supports JTAG, SPI and UART. Since we chose to use the Raspberry Pi Zero as an adapter for both JTAG and SPI, the Shikra was solely used to interface with UART. In order to do so, the correct pins on the Shikra had to be determined. The table below shows which pins on the Shikra correspond to which UART pins.

Shikra pins	UART pins
18	GND
1	Tx
2	Rx

Table 5.2: The Shikra pinout for UART connection.

When these pins are used on the Shikra, it is essential to remember what is depicted in section 2.1. The Tx pin has to be connected to Rx, and the Rx pin has to be connected to Tx on the device one attempts to interface with. The Shikra can be seen below.

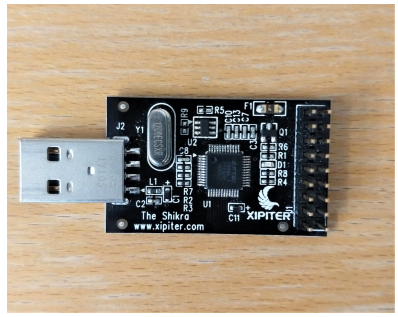


Figure 5.4: The Shikra.

[6] `https://int3.cc/products/the-shikra`, accessed on 04.03.2020

5.2 Fuzzing

OpenBTS 5.0

OpenBTS 5.0 is a free, open-source software project developed by Range Networks[7], which implements most of the GSM protocol stack on top of a radio modem. Thus, it can be used to simulate a 2G mobile network. The software operates together with a Software Defined Radio (SDR). The radio will then function as a Base Transceiver Station (BTS), that User Equipment (UE) can connect to. OpenBTS is highly configurable, and several parameters related to mobile networks can be set. It also includes a command-line interface, enabling specific functionality such as sending SMSs from the network to a connected device.

Ettus research N200 USRP

A Universal Software Radio Peripheral (USRP) serves as a hardware platform for a SDR. A SDR is a radio communication system that realizes components such as filters, amplifiers, and modulation/demodulation in software. These components have traditionally been realized through hardware. Specifically, we make use of the N200 USRP sold by the company Ettus Research[8]. In our research towards RO_2, the USRP is intended to function as an illegitimate base station within a mobile network, to which the HMU will connect.

Figure 5.5: The Ettus Research N200 USRP.

Signal Shield W-40 jammer

The Signal Shield W-40 jammer is capable of blocking out 3G signals. Thereby, it is a necessary tool for our research into RO_2, as this will force the HMU to connect to

[7] `https://github.com/RangeNetworks/openbts`, accessed on 05.03.2020
[8] `https://www.ettus.com/all-products/un200-kit/`, accessed on 05.03.2020

the GSM network which provides the strongest signal. This is further explained in section 2.9. A photo of the jammer can be seen below.

Figure 5.6: The Signal Shield w-40 jammer.

Chapter 6
Hardware Security testing of the Cardiomessenger 3G Smart

This chapter presents and details our findings and results from our research into the first scope of this thesis project. Initially, we performed preliminary hardware security testing on older HMUs, followed by the hardware security testing on the Cardiomessenger 3G Smart. Our results constitute vulnerabilities that are present at the physical level.

6.1 Preliminary HMU security testing

Our initial experiments was centered around acquiring sufficient knowledge and experience with the tools and methodology that we were going to use on the Cardiomessenger 3G Smart. By doing so, we would familiarize ourselves with the environment and be well prepared for future experimentation. In turn, this yields a more significant probability for success in our security testing of that device.

Previous research by Bour [NB19] validated a set of security-related hypotheses on older Biotronik HMUs. Specifically, the security of the Cardiomessenger II-S HMU, both the T-Line and GSM version, were thoroughly tested. Therefore, a natural starting point for us would be to attempt to replicate the results of his research on the same devices.

6.1.1 The Cardiomessenger II-S T-Line

Connection to debugging interfaces

In his thesis, Bour located and performed soldering on the JTAG pins of this PCB. Hence, we were able to connect to the JTAG pins swiftly, and by use of the JTAGulator hardware hacking tool, which is described in 5.1, we were also able to determine which of the JTAG connections corresponded to the five TAP pins. This was achieved by first launching a `idcode` scan and subsequently a `bypass` scan using the JTAGulator. A partial excerpt of the JTAGulator process is listed below.

```
...
JTAGulating! Press any key to abort...
---------
TDI: 11
TDO: 10
TCK: 9
TMS: 12
TRST#: 7
Number of devices detected: 1
-
BYPASS scan complete.
...
```

Code Listing 6.1: JTAGulator determining the TAP pins of JTAG.

A picture showing the lab setup can be found in figure 6.1 below.

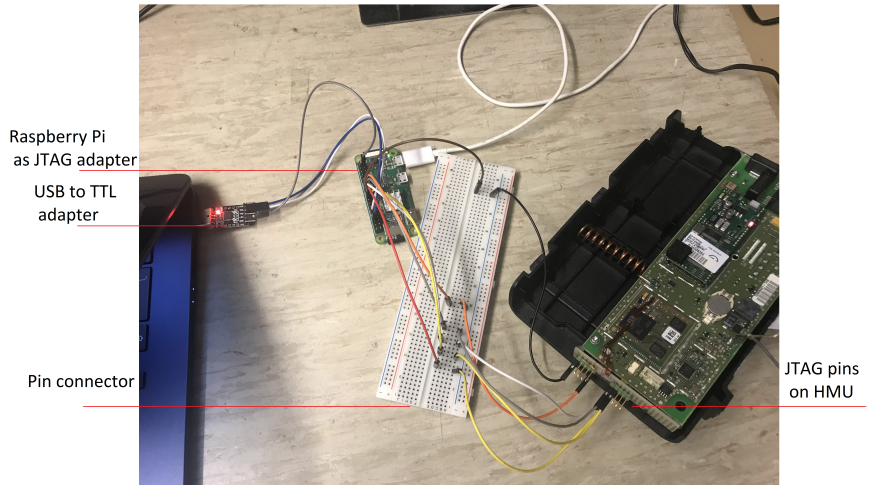


Figure 6.1: The basic setup required for connecting to JTAG on the T-Line version of the Cardiomessenger II-s HMU.

Firmware dump and analysis

From there, we successfully dumped the firmware of the device. This was possible due to the utilization of OpenOCD and corresponding configuration-files written by Bour, in addition to the Raspberry Pi Zero as a JTAG adapter. Both OpenOCD and the Raspberry Pi Zero is explained further in section 5.1. We also performed a brief analysis of the obtained files. Our analysis revealed identical results as those obtained within Bour's research. Listings 6.2 and 6.3 shows the successful reproduction of dumping and analyzing the firmware of the device.

```
...
Info : at91rm9200.cpu: hardware has 2 breakpoint/watchpoint units
    TargetName         Type       Endian TapName            State
--  ------------------ ---------- ------ ------------------ ------------
 0* at91rm9200.cpu     arm920t    little at91rm9200.cpu     halted
Dumping bootloader...
dumped 1048576 bytes in 51.357815s (19.939 KiB/s)
Done!
Dumping SRAM...
dumped 104576 bytes in 5.105570s (20.003 KiB/s)
Done!
Dumping Flash content...
dumped 4194304 bytes in 205.016098s (19.979 KiB/s)
Done!
Dumping RAM...
dumped 2097152 bytes in 102.836449s (19.915 KiB/s)
Done!
```

Code Listing 6.2: Dumping firmware of the T-Line version using OpenOCD, and Raspberry Pi Zero.

```
sintef@localhost:~/tmp/memory_dumps$ strings sdram.img | grep -i get
...
 GetDataFromEncryptionLayer: too many padding bytes
GetDataFromTransportLayer sanity
 CRC check in GetDataFromTransportLayer
GetDataFromTransportLayer:start
 TransportLayerToFifo: GetDataFromTransportLayer()
GetDataFromEncryptionLayer:start
TransportLayerToFifo: GetDataFromEncryptionLayer()
 GetDataFromCompressionLayer:start
TransportLayerToFifo: GetDataFromCompressionLayer()
GetDataFromMessageLayer:start
TransportLayerToFifo: GetDataFromMessageLayer()
...
GetContainerFromGroup: invalid source or address for message container
GetDataFromEncryptionLayer: wrong ID byte (%02Xh): expected TRIPLE_DES_CBC (%02Xh)
or AES_CBC (%02Xh)!
GetDataFromEncryptionLayer: wrong ID byte (%02Xh): expected DES (%02Xh),
TRIPLE_DES_CBC (%02Xh) or AES_CBC (%02Xh)!
...
```

Code Listing 6.3: Looking for debug strings in the firmware of T-Line version.

Network interfaces

Next, Bour performed multiple tests related to the networking interfaces of the device. Since the HMU ultimately sends patient data to the vendor's servers, experimenting with its network interface could reveal vulnerabilities that compromise patient privacy. In his thesis, Bour found the documentation for the modem on PCB, which was removable from the PCB. He also identified the UART pins of the modem on the PCB. By connecting to the UART pins using the Shikra, he was able to launch a passive eavesdropping attack on the communication channel between the modem and the microcontroller, and a attack which spoofs the modem.

When connecting to the modem's UART pins ourselves, using the Shikra, we were able to reproduce the results of Bour. The Shikra is further explained in section 5.1. By adapting the scripts that he developed as part of his thesis, we effectively carried out the different attacks without much effort. Listings 6.4 and 6.5 show the resulting communications that were captured.

```
python .\cm-serial-monitor.py COM4 115200

...
[2020-02-04 15:34:42] atatii5#vversionAT#MCOUNTRY?
at
AT+WOPEN=1
AT#DIALSELECT=1
AT#AUTHENT=PAP
at#atcmd=0,"-STE=7"
at#atcmd=1,"+A8E=6,5,0,1,0,0"
at#atcmd=2,"X3"
at
AT#DIALN1="[REDACTED]"
AT#ISPUN="[REDACTED]@cm3-homemonitoring.de"
AT#ISPPW="[REDACTED]"
at#atcmd=0,"-STE=7"
[2020-02-04 15:34:44] AT#CONNECTIONSTART
...
```

Code Listing 6.4: Passive eavesdropping of network communication.

```
...
< [2020-02-04 15:18:03] [REDACTED]@cm3-homemonitoring.de[REDACTED]
34383030353737364063 6d332d686f6d656d6f6e69746f72696e672e64650d514f376f487668364c350
INFO: [REDACTED]@cm3-homemonitoring.de[REDACTED] Command not registered
< [2020-02-04 15:18:03] aa00050dde020000[REDACTED]0809f61488d9a
...
```

Code Listing 6.5: Communication when spoofing the modem.

For an in-depth explanation of these results and what they imply, we encourage the reader to visit the thesis of Bour [NB19], as we will not cover any details of the results. However, we can confirm that the username and password are sent in cleartext over UART, and can be accessed by an attacker without much effort.

6.1.2 The Cardiomessenger II-S GSM

Connection to debugging interfaces

Due to the complexity of soldering on these devices' JTAG pins, only the Cardiomessenger II-S T-Line version had its firmware fully dumped and analyzed, during the research by Bour [NB19]. The firmware of the GSM version was only partially dumped, largely due to an unstable connection to the device's JTAG interface. However, thanks to new lab equipment, we managed to establish a stable connection

with the JTAG interface of the GSM version. Instead of soldering on the pins, we simply connected to it using the PCBite, which is further explained in section 5.1. Since Bour [NB19] had determined which pins of the JTAG interface on the PCB corresponded to the five TAP pins, we did not make use of the JTAGulator on the GSM version.

A picture of the lab setup can be found in figure 6.2 below.

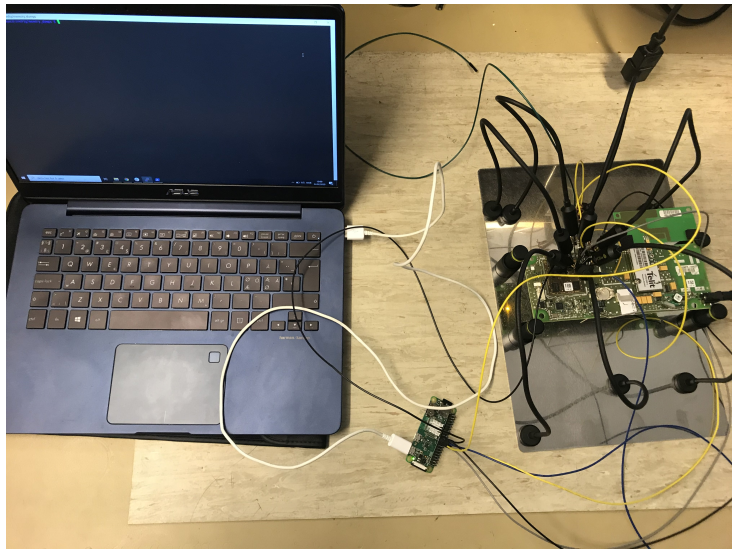


Figure 6.2: JTAG connection on the HMU GSM version.

Firmware dump and analysis

Given that both the T-Line and GSM version had identical microcontrollers, we used OpenOCD and Bour's configuration-files identically as for the T-Line version. We were successful in dumping the entire firmware of the GSM version.

When looking for the same debug strings in the firmware of the GSM version, like those found in the firmware of the T-Line version, we can observe many similarities. Those similarities imply that they might have reused most of their firmware between the two version. Listing 6.6 below show the output of a search for the same debug strings as the those found on the T-Line version.

```
sintef@localhost:~/tmp/memory_dumps/gsm$ strings sdram.img | grep -i get
...
 GetDataFromEncryptionLayer: too many padding bytes
GetDataFromTransportLayer sanity
CRC check in GetDataFromTransportLayer
```

```
GetDataFromTransportLayer:start
 TransportLayerToFifo: GetDataFromTransportLayer()
GetDataFromEncryptionLayer:start
TransportLayerToFifo: GetDataFromEncryptionLayer()
 GetDataFromCompressionLayer:start
TransportLayerToFifo: GetDataFromCompressionLayer()
GetDataFromMessageLayer:start
TransportLayerToFifo: GetDataFromMessageLayer()
...
```

Code Listing 6.6: Looking for the same debug strings in the firmware of GSM version, as those found on the T-Line version.

6.1.3 Summary of preliminary security testing

In summary, we were able to reproduce and confirm the results of Bour [NB19], in addition to obtain a full firmware dump of the Cardiomessenger II-S GSM. Through trial-and-error, we familiarized ourselves with the necessary tools and software extensively. Our motivation behind this preliminary testing was to acquire experience and knowledge, something that we indeed achieved.

6.2 The Cardiomessenger 3G Smart as a standalone device

In this section, we delve into the first scope of this master thesis, namely "The Cardiomessenger 3G Smart as a standalone embedded medical device". We will conduct experiments that aim to validate our first main hypothesis H_1, which is again included below.

The Cardio messenger 3G Smart HMU contains vulnerabilities as a standalone embedded medical device that can aid an attacker having physical access to the device to compromise a patient's privacy or safety.

As stated in section 1.3.1, in order to validate H_1, we will aim to validate the hypotheses in table 1.1. Those hypotheses have previously been confirmed for previous versions of HMUs from the same vendor, and by testing them on the most recent version, we will be able to conclude whether H_1 is true or false. Also, we will be able to describe the evolution of security in HMUs from Biotronik. Our approach when testing the device is the black-box hardware testing methodology, further detailed in section 4.2.1.

The Cardiomessenger 3G was initially approved for market in the U.S. by the Federal Communication Commission (FCC) in 2013[1], and is the only HMU that is shipped from the vendor today. Thus, if a malicious adversary would want to utilize a patient's Biotronik HMU in an attack to compromise either their privacy or safety, she would most likely be forced to look for vulnerabilities in the Cardiomessenger 3G Smart. The Cardiomessenger 3G differs from previous versions of HMU in its use of 3G mobile technologies when establishing a connection to Biotronik's backend servers. The figure below depicts both the front and back of the device.

(a) The front.

(b) The back.

Figure 6.3: The Cardiomessenger 3G Smart as it is shipped from Biotronik.

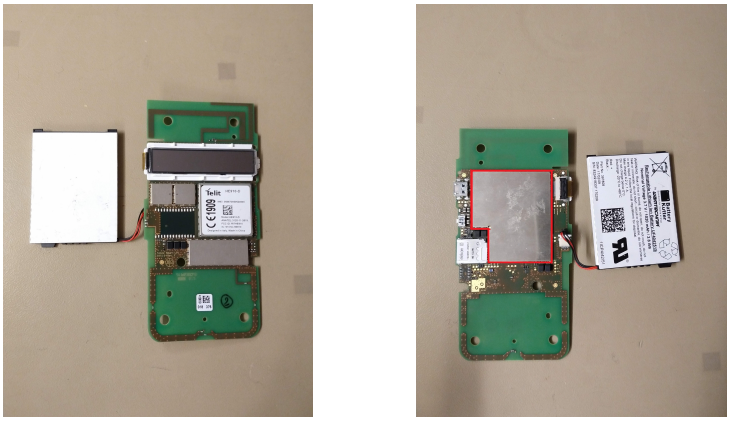
[1] `https://fccid.io/QRICMSMART/Letter/30-CMsmart3G-Class-II-Perm-Change-Letter-signed-3320541` accessed: 24.02.2020.

6.2.1 Hardware analysis

The first phase in our methodology is to analyze the hardware components of the device. To do so, we needed to open the device, hence remove the plastic cover to obtain access to the PCB. Luckily, this was already done on one of the Cardiomessenger 3G Smart HMUs by Bour, in his previous thesis work [NB19]. Since there are a total of three Cardiomessenger 3G Smart's available at our disposal in the SINTEF lab, we tried to open one ourselves and managed to get access to the PCB. We had to be careful not to damage the electronics of the device when removing the plastic.

Once we had removed the plastic, we discovered that the PCB was partially covered by a metal shield. The metal shield intends to protect against electromagnetic interference. In figure 6.4b, highlighted in red, one can see how the PCB was partially concealed due to the metal shield. It was removable as well since it was not soldered onto the PCB, but rather attached with small steel clips.

In comparison with the previous HMUs from the same vendor, there have been several enhancements regarding PCB design. The PCB of the Cardiomessenger 3G Smart consists of small chips and modules, none of which are detachable from the PCB. Additionally, the PCB conceals most pins of the various chips and components, and most communication busses are seemingly integrated into the plastic of the PCB itself. In general, such enhancements in PCB design make it less practical to security test the hardware of the device. Typical and well-known hardware vulnerabilities might be harder to uncover, or might not be present anymore, due to more advanced PCBs.



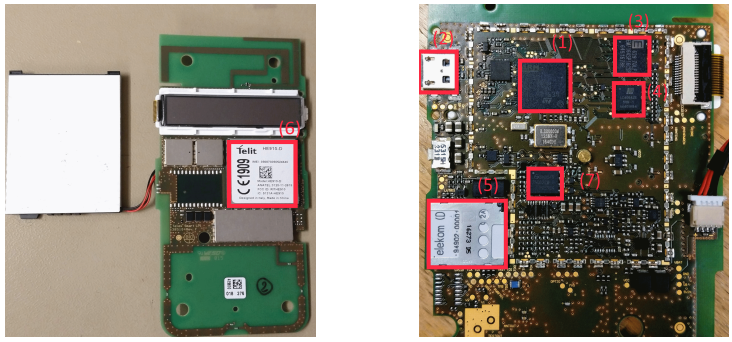
(a) The front. (b) The back.

Figure 6.4: The PCB of the Cardiomessenger 3G Smart.

Finding 1

The plastic cover of the device can be removed without damaging the PCB.

After we removed the metal plate which partially concealed the PCB of the HMU, the entire design and architecture were exposed. This can be seen in figure 6.5 below.



(a) The front. (b) The back.

Figure 6.5: The inner PCB of the Cardiomessenger 3G Smart.

As shown in figure 6.5, we could retrieve quite a lot of information regarding the HMU's design and architecture. Table 6.1 summarizes the different chips and modules identified on the inner PCB.

Number	Description
1	The microcontroller. It has an ARM processor, and the model number is STM32F4173196.
2	USB-mini port.
3	A 2MB SRAM from Jeju semiconductor. Product number is EM7164SP.
4	A programmable logic device from Lattice semiconductor. Product number is LC4064ZE.
5	A removable SIM card.
6	The modem. It is a Telit product and the model is He910-d.
7	External flash. The manufacturer is Giga Device and the product number is GD25Q32C. It has 4MB storing capacity.

Table 6.1: Identified components within the hardware analysis of the Cardiomessenger 3G Smart.

Finding 2

All the chips and modules of the HMU are identifiable.

Furthermore, the vendor has labeled the debugging interfaces, hence both the JTAG and UART pins were easy to identify. This removed the need for an identification process, as were required for the previous models of HMUs. We did not have to guess which pins corresponded to the debugging interfaces, and did not have to use the JTAGulator to determine the TAP pins of JTAG. Figure 6.6 shows the labelled pins, where (1) is the JTAG pins (described in section 2.6), (2) is the ground reference and (3) is the UART pins (described in section 2.4).

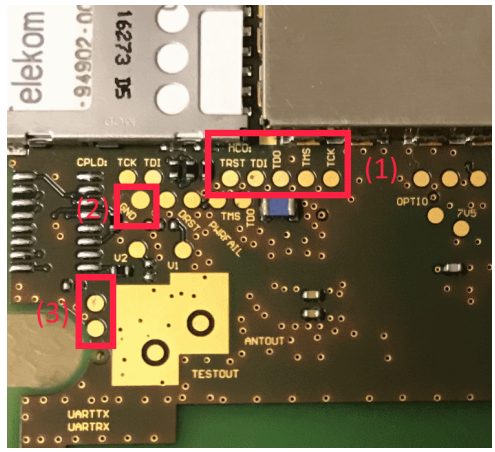


Figure 6.6: The labelled JTAG and UART pins on the PCB of the HMU.

Finding 3

Both JTAG and UART debugging interfaces are labelled on the PCB, and thus easily identifiable.

6.2.2 Finding relevant documentation

The next phase involved scouting the web for documentation and information related to the identified chips and modules that are summarized in table 6.1. The desired output from this phase was an insight into how to test the hypotheses in table 1.1 on the Cardiomessenger 3G Smart.

Luckily for us, the STM32-series is a rather popular and well-documented microcontroller for embedded devices, with active online forums. The datasheet for the microcontroller was found online[2]. It contains the memory mapping of the microcontroller, which can be seen in figure 6.7 below.

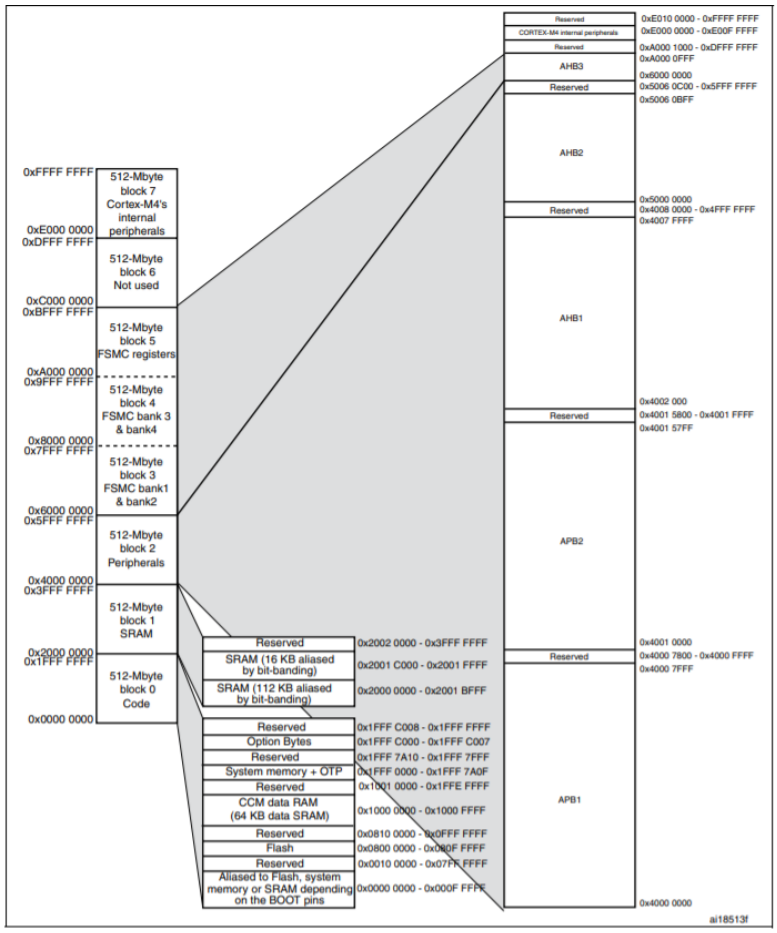


Figure 6.7: The memory map of the STM32F4 microcontroller.

Naturally, some parts of the memory are likely to be of more interest than others. Firstly, the sections which are given familiar names in the memory map, such as flash, CCM RAM, system memory OPT and SRAM, were all of immediate interest. We guessed that these memory sections would constitute the firmware of the HMU, and thereby also its bootloader. If so, obtaining the firmware could, in turn, prove

[2] `https://www.alldatasheet.com/view.jsp?Searchword=STM32F415&sField=2` accessed on 25.02.2020

to assist us in uncovering further vulnerabilities. An important note is that these memory sections are physically placed within the microcontroller.

Secondly, as detailed in figure 6.8 below, if we look at what resides in the Advanced High-Performance Bus Architecture (AHB) banks, we see that AHB3 maps to multiple Flexible Static Memory Controllers (FSMCs). A FSMC's main feature is that it interfaces with static memory-mapped devices like flash and RAM. We guessed that the vendor uses these controllers to interface with the peripheral flash and RAM chips on the board. It is noteworthy that each FSMC bank is 256MB, and that each bank again splits up in four different banks of 64MB each.

Table 10. STM32F41x register boundary addresses

Bus	Boundary address	Peripheral
	0xE00F FFFF - 0xFFFF FFFF	Reserved
Cortex-M4	0xE000 0000 - 0xE00F FFFF	Cortex-M4 internal peripherals
	0xA000 1000 - 0xDFFF FFFF	Reserved
AHB3	0xA000 0000 - 0xA000 0FFF	FSMC control register
	0x9000 0000 - 0x9FFF FFFF	FSMC bank 4
	0x8000 0000 - 0x8FFF FFFF	FSMC bank 3
	0x7000 0000 - 0x7FFF FFFF	FSMC bank 2
	0x6000 0000 - 0x6FFF FFFF	FSMC bank 1
	0x5006 0C00- 0x5FFF FFFF	Reserved
AHB2	0x5006 0800 - 0x5006 0BFF	RNG
	0x5006 0400 - 0x5006 07FF	HASH
	0x5006 0000 - 0x5006 03FF	CRYP
	0x5005 0400 - 0x5005 FFFF	Reserved
	0x5005 0000 - 0x5005 03FF	DCMI
	0x5004 0000- 0x5004 FFFF	Reserved
	0x5000 0000 - 0x5003 FFFF	USB OTG FS
	0x4008 0000- 0x4FFF FFFF	Reserved

Figure 6.8: A detailed memory map of AHB3, depicting how it maps multiple FSMCs.

Furthermore, the respective datasheets of all the identified chips and modules in table 6.1 were collected from the Internet and constitute the basis of information when we deduce testing scenarios for the hypotheses in table 1.1.

Finding 4

Datasheets of the microcontroller, as well as all other chips and modules on the PCB, are obtainable online.

6.2.3 Testing Scenarios

Connecting to JTAG

The first hypothesis that we tested was naturally whether the JTAG interface was enabled after production on the most recent HMU, given that this was indeed the case for previous versions. If so, it provides a potential attacker with means to control the microcontroller and the ability to execute code. We used OpenOCD and the Raspberry Pi Zero to interface with JTAG. An advantage of using OpenOCD was the opportunity to build upon the connection script that was developed by Bour [NB19]. We had already had familiarized ourselves with those scripts, as detailed in section 6.1. This helped speed up the entire process. It was also our motivation behind choosing the Raspberry Pi Zero as a JTAG adapter, which was already specified as the JTAG adapter in Bour's scripts. In comparison to system-specific software that interfaces JTAG, OpenOCD provides the user with less specific functionality and pre-made test cases. However, such system-specific software is often quite expensive, while OpenOCD is free of charge.

In order to use the Raspberry Pi as a JTAG adapter, we had to connect its pins according to table 5.1. We established a connection to the JTAG interface by simply doing a minor change to the existing connection script in Bour's thesis [NB19]. That is, we specified the correct microcontroller as the target. Listing 6.7 below shows the successful output of this. In appendix A.1 the entire script used to connect to the Cardiomessenger 3G Smart's JTAG interface, using OpenOCD and the Raspberry Pi Zero, can be found.

```
sintef@sharp:~/tools/hardware-hacking/openocd-config/3gTesting/:$ Open On-Chip
Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
BCM2835 GPIO config: tck = 11, tms = 25, tdi = 10, tdo = 9
BCM2835 GPIO config: srst = 24
srst_only separate srst_gates_jtag srst_push_pull connect_deassert_srst
adapter speed: 500 kHz
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
jtag_ntrst_delay: 100
srst_only separate srst_nogate srst_push_pull connect_deassert_srst
cortex_m reset_config sysresetreq
srst_only separate srst_nogate srst_push_pull connect_deassert_srst
adapter_nsrst_delay: 100
adapter_nsrst_assert_width: 100
Info : BCM2835 GPIO JTAG/SWD bitbang driver
Info : JTAG only mode enabled (specify swclk and swdio gpio to add SWD mode)
Info : clock speed 2030 kHz
Info : JTAG tap: stm32f4x.cpu tap/device found: 0x4ba00477 (mfg: 0x23b
    (ARM Ltd.), part: 0xba00, ver: 0x4)
Info : JTAG tap: stm32f4x.bs tap/device found: 0x06413041 (mfg: 0x020
    (STMicroelectronics), part: 0x6413, ver: 0x0)
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
    TargetName         Type       Endian TapName            State
--  ------------------ ---------- ------ ------------------ ------------
```

```
0* stm32f4x.cpu       cortex_m    little stm32f4x.cpu       halted
```

Code Listing 6.7: Successful JTAG connection using OpenOCD and Raspberry Pi Zero.

Once we established the connection, OpenOCD hosts a GDB server that interfaces with JTAG, which can we accessed through a TCP/IP socket. Throughout this thesis, `netcat` has been used to establish such a socket, which involves executing the command `nc 127.0.0.1 4444` on the Raspberry Pi. By establishing such a socket, we can pipeline OpenOCD commands to be executed on the target.

Finding 5

JTAG is enabled on the Cardiomessenger 3G Smart, and provides an attacker with fine-grained control over memory and the CPU registers, and the ability to execute code on the device.

Furthermore, we made attempts to interact with the HMU's UART interface. This was carried out via an appropriate UART connection, using the Shikra. Although we attempted to interact the HMU's UART both during boot, and during regular operation, it appears that UART has been disabled on the HMU after production.

Finding 6

UART is not enabled on the device.

Dumping different parts of memory

One of the many advantages JTAG access provides is the ability to read and write to memory. If one knows the specific memory address of something to extract, and the size of it, it is possible to query this through OpenOCD with the `dump_image` command. Naturally, if we are interested in dumping the peripheral flash or RAM, the memory addresses of these chips must be found.

Based on the available memory map for the `STM32F417` microcontroller, we extracted the previously mentioned interesting parts of its internal memory sections. Noteworthy, depending on the boot mode, the microcontroller loads different binaries in `0x0000 0000`. However, the entire memory block from `0x0000 0000 - 0x3FFF FFFF` was, as expected, the microcontroller's internal memory, which includes the HMU's firmware and thus also its bootloader.

Because there were no signs of any peripheral memory modules in these dumps, it left us guessing where the vendor could have mapped, for instance, the peripheral

RAM - as that is of major interest. We initially thought that the memory sections labeled "SRAM" could have been the main RAM chip, but further inspection convinced us otherwise. One observation was key to determine that we had not collected the peripheral RAM chip. The size of the SRAM memory block did not align with the documentation of the peripheral RAM chip found in the previous phase. Thus, the peripheral RAM chip had to be mapped elsewhere. As already mentioned, the FSMC banks were a natural next guess. This turned out to be correct, and the contents of the peripheral RAM chip was extracted from within the first FSMC in AHB3. Figures 6.9 and 6.10 below depict which memory sections we successfully extracted through JTAG.

Listing 6.8 shows the output in OpenOCD when we successfully executed our dump script, which is built on Bour's original dump script [NB19]. Only the memory addresses of the chips needed to be changed. Using the Linux command `diff`, we discovered that the memory blocks labeled "Flash" and "Aliased to Flash, system Memory or SRAM depending on the BOOT pins" were identical.

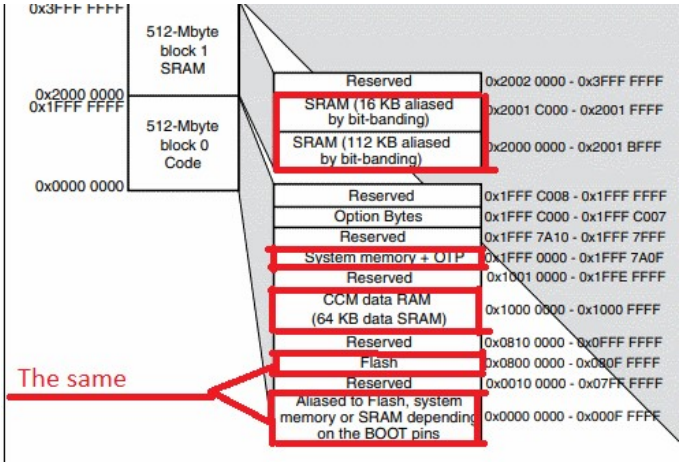


Figure 6.9: Extracted memory sections from the microcontrollers internal memory.

AHB3	0xA000 0000 - 0xA000 0FFF	FSMC control register
	0x9000 0000 - 0x9FFF FFFF	FSMC bank 4
	0x8000 0000 - 0x8FFF FFFF	FSMC bank 3
	0x7000 0000 - 0x7FFF FFFF	FSMC bank 2
	0x6000 0000 - 0x6FFF FFFF	FSMC bank 1

Figure 6.10: Extracted memory sections from the first FSMC, in AHB3.

```
...
Dumping flash...
dumped 1048575 bytes in 9.338343s (109.655 KiB/s)
Done!
Dumping CCM RAM...
dumped 65535 bytes in 0.596915s (107.216 KiB/s)
Done!
Dumping system memory OTP...
dumped 31247 bytes in 0.278609s (109.525 KiB/s)
Done!
Dumping SRAM...
dumped 131071 bytes in 1.166101s (109.767 KiB/s)
Done!
Dumping RAM...
dumped 2097152 bytes in 18.758638s (109.176 KiB/s)
Done!
```

Code Listing 6.8: The successful execution of the memory dump script in OpenOCD.

Finding 7

The entire firmware, including the bootloader, and the contents of the peripheral RAM, can be extracted from the device using JTAG and OpenOCD.

Reading the external flash directly

However, at this point, it was not evident whether the memory addresses in the microcontroller's memory map encompass the peripheral flash. It would be interesting to read, as it might store sensitive data. We guessed that the peripheral flash chip would map somewhere within an FSMC since this was the case for the peripheral RAM.

Thus, we made attempts to read the peripheral flash chip, although it was not obvious where it maps to, from the microcontroller's memory map. Since we never found the correct memory address by inspecting the documents, we pulled chunks equalling the size of the peripheral flash chip in the different FSMCs memory sections. According to the documentation of the flash chip, when the microcontroller wants to access the contents of the flash, it first sets the CS pin high and subsequently oscillate clock signals to synchronize. By coupling an oscilloscope to the CS pin of the external flash chip, we would, therefore, expect that it went from either high to low or vice-versa, if it indeed was read. However, this was never observed, and our brute-force approach gave no results. Thereby, we could conclude that the microcontroller did not map the peripheral flash chip, but accessed it by other means.

Two more options remained. The first was to somehow extract the data of the peripheral flash chip, through the JTAG interface of the PCB, using OpenOCD. The second was to connect to the peripheral flash chip via SPI since the chip's pins are exposed on the PCB. Although OpenOCD comes with some functionality to interface

peripheral flash chips, its complexity left us discouraged. A direct SPI connection to the flash chip could also be established by the use of the PCBite, which we now feel comfortable with, due to its exposed pins. The latter option was, therefore, chosen.

The Raspberry Pi Zero that was previously used as a JTAG adapter supports SPI communication as well, which is what was utilized in this experiment. In addition to the Pi, we needed some software to simulate the microcontroller's behavior by querying a read of the entire chip. Luckily, this already exists in the Flashrom package, which runs on all Linux distributions, including our Raspberry Pi image. Flashrom is further explained in section 5.1.

First, we had to couple the correct pins of the Raspberry Pi with jumper wires, and its SPI interface had to be enabled. The exact mapping can be found in table 5.1. The pins of the peripheral flash chip are not labeled on the PCB, so the correct pins were found in the datasheet of the peripheral flash chip. A figure depicting those can be seen in figure 6.11 below.

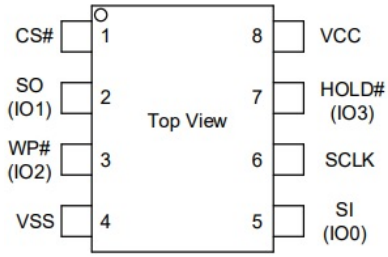


Figure 6.11: The flash pinout of the peripheral flash chip.

Listing 6.9 shows the interaction with the flash chip when using Flashrom to act as the microcontroller, after setting up the probes of the PCBite as shown in figure 6.12. Noteworthy, several tutorials that we found on the Internet which use Flashrom to extract data from flash chips via SPI do not involve the `VCC` pin. However, this experiment was only successful when we used this pin to induce a current through the chip, while the HMU was powered off.

```
sintef@sharp:~/tools/hardware-hacking/:$ sudo flashrom -p \
    linux_spi:dev=/dev/spidev0.0,spispeed=1000 -r flash.bin
flashrom on Linux 4.19.93+ (armv61)
flashrom is free software, get the source code at https://flashrom.org

Using clock_gettime for delay loops (clk_id: 1, resolution 1ns).
Found GigaDevice flash chip "GD25Q32(B)" (4096 kB, SPI) on linux_spi.
Reading flash... done.
```

Code Listing 6.9: Using Flashrom to read peripheral flash chip.

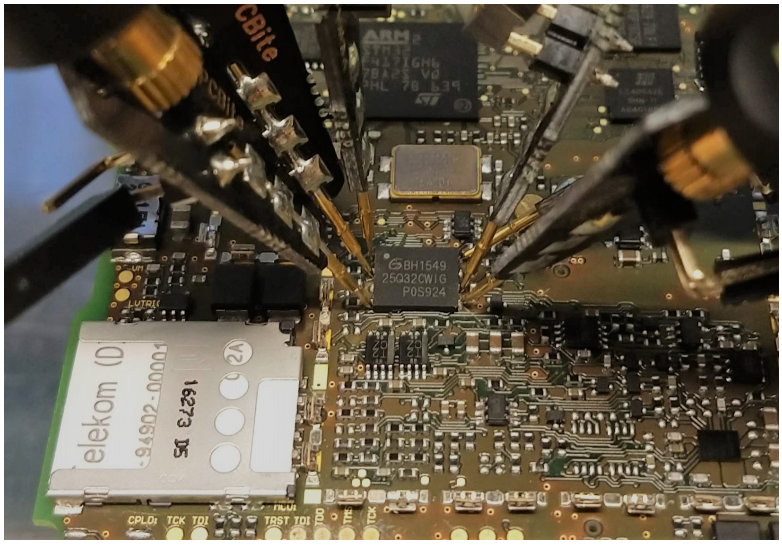


Figure 6.12: Connecting to the SPI of the flash directly.

Finding 8

The contents of the peripheral flash chip can be extracted from the device using SPI and Flashrom.

Investigating the memory dumps

Initially, we made attempts at searching for something as trivial as the word "pin" in the contents of the peripheral RAM chip, hoping to retrieve the pin-code of the SIM-card in the HMU. For this, the command `strings` was used, which will print strings in any file that are printable. Listing 6.10 shows the resulting output, and it appeared that the pin code was indeed stored in cleartext in the peripheral RAM as part of an AT-command. The fact that these strings could be retrieved implies that the peripheral RAM is unencrypted. In turn, this implies that all data the HMU forwards to the backend server will, at some point, be located unencrypted in RAM.

```
root@kali:~/Desktop/Master# strings ram.img | grep -i pin
AT+CPIN?
+CPIN: SIM PIN
AT+CPIN="[REDACTED]"
AT+CPIN?
AT+CPIN="[REDACTED]"
```

Code Listing 6.10: Searching for the word 'PIN'.

Using the `strings` command once again on the contents of the internal flash memory, where the firmware is stored, revealed more information. It seemed that the protocol that interfaces with the pacemaker is called Ultra Low Power Active Medical Implant Systems (ULPAMI). Thus, we attempted to search for "ulpami" in the firmware. Listing 6.11 shows parts of the output. It is not obvious what these strings are used for, but they are probably meant for debugging and logging for the developers. Again, if an attacker was interested in reverse-engineering the protocol between the pacemaker and the HMU, accessing these debug strings and mapping them to their respective functions in the binaries, will aid in the process. Searching for other strings such as "GSM" gave similar results and could be helpful when reverse-engineering the interface towards the mobile network.

```
root@kali:~/Desktop/Master# strings flash.img | grep -i ulpami
...
UlpamiInitRx: ERROR!! Length for RX string is too long! Turncated to
    ULPAMI_MAXSTRINGLENGTH
Ulpami_low/singlestringende: Callback FATAL ERROR: returned UlpamiRxLen too large:
    RxString-buffer overflow!
Ulpami_low/RxDone: Callback FATAL ERROR: returned UlpamiRxLen too large:
    RxString-buffer overflow!
ULP: CM - UlpStatDat->TxEDP == 1 | UlpamiDLMsgCnt == %d | UlpTxBytes2go = %d
...
```

Code Listing 6.11: Searching for the word "ULPAMI".

Finding 9

The use of debug/log strings in the firmware greatly eases the reverse engineering process of the communication protocols towards both the backend server and the pacemaker.

Reading up on the documentation from the microcontroller, we have reason to believe that the first section within the internal flash is, in fact, the bootloader of the firmware. The fact these strings are retrievable in the internal flash memory of the microcontroller means that the entire firmware is unencrypted.

Finding 10

The firmware of the HMU is not encrypted, which eases the process of reverse engineering.

Next, we searched for multiple words that could be of interest in the different memory-dumps that we collected. Although none of the memory-dumps were encrypted, not everything resulted in a hit. In Lie's thesis [WL19], she found the

APN and the username that the HMU uses when authenticating to the APN. Also, she identified the IP address of Biotronik's server and its port number. As it would be interesting to search for these strings in the memory-dumps that we collected and check if the Cardiomessenger 3G Smart also stores those credentials, we performed several different searches in all the different memory-dumps, using `strings`. We can thereby confirm that the APN, the IP address, the port number, and the username are all stored unencrypted on the Cardiomessenger 3G Smart. This information could potentially help an attacker to establish a connection to the backend servers of the vendor.

Finding 11

None of the memory-dumps, including the peripheral flash and RAM, are encrypted. Credentials (PIN, IP, port number, APN) are stored in cleartext.

Finally, by searching for the word "src", a list of what appears to be source files in the HMU is shown. This is included in listing 6.12. A similar listing was produced by Bour [NB19] when he researched previous versions of HMUs from the same vendor. We share his belief that these source files are not physically stored on the HMU. These strings come from the "keyword substitution" feature of the Concurrent Version System (CVS). This feature is probably used to generate the logging strings automatically.

```
root@kali:~/Desktop/Master# strings flash.img | grep -i src
Src Buffer Address 0x%p
$Source: src/variables.c $ $Revision: 1.1 $
$Source: src/variables.h $ $Revision: 1.2 $
Src Buffer Address 0x%p
$Source: src/UlpamiLow_nonsh_config.h $ $Revision: 1.47 $
$Source: src/UlpamiLow_nonsh_config.c $ $Revision: 1.66 $
$Source: src/UlpamiLow_nonsh_debug.c $ $Revision: 1.43 $
$Source: src/UlpamiLow_nonsh_debug.h $ $Revision: 1.42 $
$Source: src/variables.c $ $Revision: 1.40 $
$Source: src/variables.h $ $Revision: 1.40 $
$Source: src/UlpamiLow_nonsh_debug.h $ $Revision: 1.42 $
$Source: src/UlpamiLow_nonsh_Hw.h $ $Revision: 1.39 $
Src Buffer Address 0x%p
```

Code Listing 6.12: Searching for the word "src".

When using the `binwalk` command, which locates executable code in binary files, the only noteworthy discovery was in the contents of the internal flash of the microcontroller. Specifically, listing 6.13 shows the output of this command. The output indicates where known binaries are located in the firmware, and also outputs the header of those pieces of code. The output shows that code written by Jean-loup Gailly and Mark Adler, is used in the firmware. After a brief search online, it was

quickly determined that Jean-loup Gailly has written and copyrighted compression algorithms. Mark Adler has also written and copyrighted compression algorithms, as well as checksum algorithms. This coincides with the fact that the HMU forwards data, where such mechanisms are commonly employed. Also, the output shows that `CRC32` is used, which is a mechanism used for correcting bit-errors. This coincides with the fact that the HMU receives data from the pacemaker, where bit-errors could take place.

```
root@kali:~/Desktop/Master# binwalk flash.img

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
219932        0x35B1C         CRC32 polynomial table, little endian
224028        0x36B1C         CRC32 polynomial table, big endian
228139        0x37B2B         Copyright string: "Copyright 1995-2003 Jean-loup
    Gailly "
230535        0x38487         Copyright string: "Copyright 1995-2003 Mark Adler "
463731        0x71373         Copyright string: "Copyright 1995-2003 Mark Adler "
464016        0x71490         CRC32 polynomial table, little endian
468112        0x72490         CRC32 polynomial table, big endian
477087        0x7479F         Copyright string: "Copyright 1995-2003 Jean-loup
    Gailly "
842708        0xCDBD4         CRC32 polynomial table, little endian
846804        0xCEBD4         CRC32 polynomial table, big endian
850915        0xCFBE3         Copyright string: "Copyright 1995-2003 Jean-loup
    Gailly "
853311        0xD053F         Copyright string: "Copyright 1995-2003 Mark Adler "
```

Code Listing 6.13: Output using `Binwalk` on flash.img file.

It is interesting to attempt to identify the exact firmware version of the HMUs in our lab. If we manage to pinpoint the exact version, future work can look at the evolution of security features in different firmware versions. Looking for the string "SMARTAPP" in the memory dumps, they all revealed the number 1.20, which listing 6.14 shows. Considering the HMU is named Cardiomessenger 3G Smart, we assume this is the firmware version of this device. However, Cardiomessenger 3G Smart's that are shipped today might employ a newer firmware version.

```
root@kali:~/Desktop/Master# strings external_flash.bin  | grep -i smartapp
SMARTAPP 1.20
SMARTAPP 1.20
SMARTAPP 1.20
...
```

Code Listing 6.14: Identifying the firmware version of the HMU.

The same rationale applies to the firmware version of the modem, considering that it runs its own OS and firmware. Looking in the memory dumps for the string "HE910-D", which is the modem's product number, we hoped to reveal the exact

firmware version. The result of this is shown in listing 6.15. Interestingly, the string "`#CGMR: 12.00.024`" is printed to the terminal if we asked for some additional context with the `grep` command. This particular string is a command that asks the modems for the software revision number. Conclusively, the modem that is used on the HMU is running firmware version 12.00.024.

```
root@kali:~/Desktop/Master# strings ram.img  | grep -i -A5 -B5 "HE910-D"
AT\V1
AT+CMEE=2
AT&D0
AT&K3
AT+GMM
HE910-D
AT#CGMM
#CGMM: HE910-D
AT#CGMR
#CGMR: 12.00.024
AT+CREG=2
AT#RXDIV=0
AT#CGSN
```

Code Listing 6.15: Identifying the firmware version running of the HMU's modem.

Naturally, it would be interesting to further investigate the obtained memory-dumps in some reverse engineering software like Ghidra or IDA-pro and try to understand its structure, while potentially identifying vulnerabilities. However, given that reverse engineering is outside of the scope of this master thesis, our investigation of the obtained memory dumps stops here.

6.2.4 Summary of findings

To summarize, when working towards RO_1, aiming at validating our first main hypothesis H_1, we have successfully carried out different testing scenarios. Our testing scenarios have yielded a set of findings, which in turn correspond to vulnerabilities in the Cardiomessenger 3G Smart HMU. Below, table 6.2 includes a summary of our findings.

Number	Finding
1	The plastic cover of the device can be removed without damaging the PCB.
2	All the chips and modules of the HMU can be identified.
3	Both JTAG and UART debugging interfaces are labelled on the PCB, thus easily identifiable.
4	Datasheets of the microcontroller, as well as all other chips and modules on the PCB are obtainable online.
5	JTAG is enabled on the Cardiomessenger 3G Smart, and provides an attacker with fine-grained control over memory and the CPU registers, and the ability to execute code on the device.
6	UART is not enabled on the device.
7	The entire firmware, including the bootloader, and the contents of the peripheral RAM, can be extracted from the device using JTAG and OpenOCD.
8	The contents of the peripheral flash chip can be extracted from the device using SPI and Flashrom.
9	The use of debug strings in the firmware greatly eases the reverse engineering process of the communication protocols towards both the backend server and the pacemaker.
10	The firmware of the HMU is not encrypted, which eases the process of reverse engineering.
11	None of the memory-dumps, including the peripheral flash and RAM are encrypted. Credentials (PIN, IP, port number, APN) are stored in cleartext.

Table 6.2: Summary of our findings on the Cardiomessenger 3G Smart.

The testing scenarios we carried out were specifically constructed to validate the hypotheses found in table 1.1, as such a validation would indeed validate H_1. In table 6.3 on the next page, a summary of which of our findings that validate the hypotheses in table 1.1 can be found.

Hypothesis number	Hypothesis	Finding(s)	True/false
$H_{1.1}$	An attacker can open the device, and there is no obfuscation of the electronics that would harden the identification of components.	1	True
$H_{1.2}$	Both JTAG and UART interfaces are identifiable and enabled on the HMU.	3, 5 and 6	Partly true
$H_{1.3}$	The memory is unencrypted, and an attacker can obtain cleartext credentials, debug strings and the data that is sent to the backend server.	11	True
$H_{1.4}$	The firmware is unencrypted and not obfuscated, easing the reverse-engineering process for an attacker.	10	True
$H_{1.5}$	The use of debug/log strings in the firmware, ease to process of reverse-engineering the communication protocol.	9	True
$H_{1.6}$	The AES-key used for encryption of patient data, is hardcoded in the HMU's memory.		Unknown
$H_{1.7}$	The APN's credentials can be gathered in cleartext from eavesdropping on the communication between the modem and the microcontroller.		Unknown
$H_{1.8}$	There is no mutual authentication between the microcontroller and the backend server of the vendor.		Unknown
$H_{1.9}$	Data is sent from the HMU to the backend servers of the vendor over a TCP-service, and credentials are sent in cleartext.		Unknown
$H_{1.10}$	Credentials used to connect to the APN is reused to connect to the TCP-service.		Unknown
$H_{1.11}$	A proprietary communication protocol is used to send data to the backend server.		Unknown

Table 6.3: Hypothesis deemed true by Bour [NB19] for previous versions of HMUs, and which of our findings potentially validate those hypotheses.

As can be seen in table 6.3, our findings validated a subset of the hypotheses from table 1.1. Given that the device could be opened, and that the metal plate which partially concealed the PCB could easily be removed, $H_{1.1}$ is fully validated. The introduction of the metal plate is not an obfuscation mechanism, but intended as protection against electromagnetic interference. $H_{1.2}$ is however, partly validated. Both JTAG and UART are labelled on the PCB, however, UART is disabled on the device. $H_{1.3}$, $H_{1.4}$ and $H_{1.5}$ are completely validated, as our findings confirm that these hypotheses are indeed true, also for the Cardiomessenger 3G Smart.

Given that it is still unknown whether the remaining hypotheses, $H_{1.6}$ through $H_{1.11}$, are true for the Cardiomessenger 3G Smart, an elaboration is required. The reason for that is two-folded. Due to PCB enhancements in the 3G version, we were

not able to eavesdrop on the UART communication bus between the microcontroller and the modem - as were the case in the previous versions. Since the pins of the modem are not exposed anymore, this testing scenario was deemed infeasible. Thus, these hypotheses needed to be tested otherwise. One solution, which Bour and Lie conducted in their projects, was to emulate the vendor's server by setting up a fake base station and spoofing the server's IP address and port number. However, since we are only concerned with the HMU as a standalone embedded device, this falls out of scope. Conclusively, there was no apparent testing scenario that we could carry out. Instead of altering our scope to accommodate this, we chose to focus on our second scope instead.

Lastly, we will summarize the evolution of security features from the previous HMU compared to the Cardiomessenger 3G Smart. One could argue that the vendor has improved upon their security on the hardware level from previous versions of HMUs. The fact that UART is disabled is a step in the right direction. However, JTAG and UART debugging interfaces are now labeled, which was not the case in previous versions of HMUs. This is indeed a step in the wrong direction. Therefore, it is difficult to believe that the improvement was rooted in security, but rather a mere consequence of the specific PCB design. As most of the validated hypotheses turned out to be true, we do not see excellent security improvement from previous versions.

Chapter 7
Fuzzing the SMS-interface of the Cardiomessenger 3G Smart

In this chapter, we will examine the second scope of this thesis project will. We will describe the process and design decisions that we made when developing our fuzzer framework, followed by an analysis of the produced results. More generally, we will attempt to answer our second main hypothesis H_2, namely:

The implementation of SMS within the HMU is insecure and contains vulnerabilities that can be exploited by an attacker determined to compromise either the safety or the privacy of a patient.

The development of the fuzzer framework is an iterative process, as described in section 4.2.2. The steps in the process will guide our development of a fuzzing framework towards the end result.

7.1 Architecture

Figure 7.1 shows a simplified overview of what we aim to build and accomplish within this scope. It involves setting up an illegitimate base station that delivers fuzzed SMSs to the HMUs. The remainder of this section will describe the choices that we made in order to realize this.

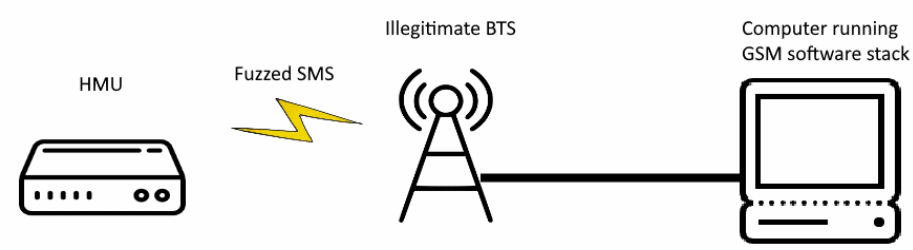


Figure 7.1: A simplified overview of the envisioned SMS fuzzing architecture.

7.1.1 Initial Considerations

There exists a broad range of different tools, equipment, techniques and approaches to create a fuzzer framework targeting the SMS interface of a device [Hon11] [MM09a] [Gol11] [DD12] [MSP20]. Preferably, a choice of technologies and tools that would lead to the most fine-grained control of the entire process would be the best choice and produce the most sound results. However, a choice like that oftentimes comes with a trade-off in complexity. This is certainly also true for the development of our framework. The monitoring capabilities, in particular, can be implemented in a lot of different ways, where each approach has its pros and cons. Choosing the adequate tools and methods when developing the fuzzer framework is a critical part of this project. Therefore, the following aims to weigh the pros and cons of different design decisions. The end result exhibits the previously described features and also contains trade-offs we are comfortable with.

When scrutinizing the SMS interface of the Cardiomessenger 3G Smart, a distinction between what segment of the HMU is affected by this must be established. When fuzzing the SMS interface in this thesis, we are targeting the baseband processor/modem of the HMU (and the baseband firmware), not the microcontroller and the application layer. The modem runs its own OS and has its own memory. When our fuzzer framework sends semi-malformed SMSs to the HMU, it is indeed the baseband processor that parses and handles the incoming messages before the payload is potentially sent to an application. Due to their semi-malformed formats, most SMSs within our test cases will be misinterpreted, and are likely to be dropped already in the modem. Therefore, potential vulnerabilities like memory corruptions and buffer overflow that may arise will happen in the modem. Previous work conducted by Weinmann [Wei12], showed the importance of secure baseband firmware and the implications it can have on the security of a system as a whole.

It is necessary to clarify what targeting the modem implies for the security test and potential uncovered vulnerabilities in a bigger picture. When targeting the modem of the HMU, any vulnerabilities that we might find are not specific to the HMU itself. On the contrary, all devices that have the same modem will have the same vulnerabilities. Vulnerabilities found in this thesis can, therefore, be exploited in other wireless devices that requires cellular connection and has the same modem. This can, for instance, be modern cars or other IoT-devices.

7.1.2 Setting up an simulated GSM network

Initially, we needed to establish the realization of a GSM mobile network for developing a SMS-fuzzer. Since we do not want to send SMSs over a real mobile network, we simulate our test GSM network. The reason for this is to avoid the risk of harming any telecommunication infrastructure when processing fuzzed messages and the

possible costs of sending thousands of them. There exist multiple free, open source software implementations of the GSM stack, and in literature, authors seem to be indifferent about their choice. Therefore the OpenBTS project was chosen, which is described in section 5.2. To give grounds for this choice, both Hond [Hon11], and Gorenc et al. [GM14] have already developed fuzzers on top of the OpenBTS source code targeting the SMS interface of mobile devices - thus it is already proven to be a compatible GSM network.

7.1.3 Automated packet delivery and manipulation

The next step was how to set up the architecture for our framework. Especially two facets of the fuzzer development showed to be crucial for the completion of the framework. The first was the automation of sending packets to the HMU from an Application Programming Interface (API), and the second was being able to control the entirety of the packet being sent. By controlling the entirety of a packet, we imply that possibly every bit will be manipulated, not only the payload, but also the SMS headers and layers further down in the SMS stack. OpenBTS does come with a pre-installed command called `sendsms`. However, this only allow users to set the payload equal to some string. Thus far from sufficient. After researching what opportunities that already exist to meet our requirements, it seemed like a viable solution existed in previous versions of OpenBTS.

This specific feature in previous versions of OpenBTS is called `testcall`, and it opens a GSM layer 3 channel to a specific International Mobile Subscriber Identity (IMSI) and forwards packets over this channel. Testcall simultaneously opens a User Datagram Protocol (UDP) server on the local machine - providing us with a API from which we can properly transmit and receive packets. Layer 3 is also the layer wherein the SMS service exists, hence a promising feature. The testcall feature has even been used previously by security researchers to fuzz SMS interfaces over OpenBTS [DD12]. Unfortunately for us, the developers of OpenBTS removed the testcall feature in release 2.8 because of safety reasons, claiming that they would not want to be liable in a possible attack where someone used testcall for malicious purposes. For the interested reader, a forum containing a discussion on testcall between the developers and security researchers can be found online[1]. Despite their removal of this feature, it has been implemented by independent developers and published online at Github. The original version can be found in Fairwaves Git repository[2] under the testcall branch. Also, another Git repository containing the testcall feature in the latest version of OpenBTS can be found[3].

[1] `https://sourceforge.net/p/openbts/mailman/openbts-discuss/thread/026f01ce30e8%24bf12f680%243d38e380%24%40schmid.xxx/#msg30679550`, accessed on 15.03.2020

[2] `https://github.com/fairwaves/openbts-2.8`, accessed on 15.03.2020

[3] `https://github.com/Djimmer/obts?fbclid=IwAR3XidP2DqT3q-AMGfXCHFo6Bw-lisCwLNvv5UP1h8HgDOOp6vvgrPwGOTo`, accessed on 20.03.2020

We then made attempts to install these versions in our lab. The latest version of OpenBTS, containing the additional testcall feature, was installed and built without errors. However, no traces of layer 3 connection establishment could be seen in the corresponding packet traces when issuing testcall - indicating that this extension probably contained bugs. We guess that it might be related to hardware. Specifically, we use different transceivers than what the developers have stated in their README. For the original old version (pre 2.8) containing testcall, we did not manage to install and build it because of old software dependencies. The package relied on several outdated software libraries that are not maintained anymore, hence challenging to acquire.

Despite testcall being a promising feature, we decided not to follow that route anymore. Rather than debugging someone else's code, which could lead nowhere, we decided to build our own modification to OpenBTS. The modification extends OpenBTS with the exact functionality we envisioned was required for fuzzing the SMS interface of the HMU.

7.1.4 Modifying OpenBTS to accommodate fuzzing

Having determined that we wanted to modify the existing source code of OpenBTS to accommodate fuzzing, we placed efforts in understanding the structure and components of the software. After investigating the files and having tried multiple different modifications to it, we successfully created a modification of OpenBTS. Our version contains two additional functions to send raw SMS PDUs. One function sends raw PDUs to a specified IMSI, and the other sends all PDUs from a text file to an IMSI. In combination with the default API provided in OpenBTS, the pipeline could be fully automated.

Moreover, the modified version of OpenBTS takes as input the entire RPDU layer, which has the TPDU layer encapsulated - enabling full control over both layers. The complete control over both layers is required as the fuzzer framework should be able to create a multitude of different SMSs. The entire process of installing the modified OpenBTS version on an `Ubuntu 16.04` computer can be found in Appendix B.

7.2 Establishing monitoring capabilities

When considering the different alternatives for establishing monitoring capabilities, there were five different approaches that we considered.

One method to monitor for system crashes at the receiver's edge would be to utilize the enabled JTAG interface on the device. In theory, after each SMS is sent, we could look for traces of system crashes by reading the CPU registers or the

RAM. However, using JTAG in such an approach would not provide insight into the baseband processor, only the microcontroller. Far from all the produced crashes are going to be visible in the microcontroller's registers [Wei12]. Thus, we did not pursue this approach further.

Another method to implement monitoring capabilities at the receiver's edge could be through UART. However, since this is disabled in the HMU, it would have been necessary to reverse engineer its firmware, such that it could be enabled again. Since reverse engineering firmware is not within the scope of this thesis, we did not attempt this method.

Relevant literature describes more creative methods that could realize extensive debugging. In the paper by Mulliner et al. [MM09b], they modified the firmware of phones to both inject SMSs and debug responses directly. Realistically this option is unsuitable for our scenario, as we do not have a similar type of access to the device.

A benefit of having conducted a hardware analysis in the first scope of this thesis is that we have acquired knowledge of the HMU's components. In section 6.2.1, we discovered that the cellular baseband processor that is used on the HMU is a Telit HE910-d chip. An option to achieve more fine-grained debugging from this chip could be to order it online and then connect it to another device that we are in control over - for instance, a Raspberry Pi. However, connecting to such debug/diagnostic ports assumes that the vendor of the modem enables these ports after shipping it. Ordering the modem would introduce the additional benefit of controlling which access point to which the device connects.

If we cannot achieve any of the alternatives described above, the last option is to look at response messages from the sender's edge (the attacker's terminal). Each sent SMS from a mobile network to a recipient triggers some kind of response. Responses can either be an acknowledgment or an error code, or neither, indicating that something went wrong. The apparent disadvantage of choosing this method is that system crashes must be inferred based on mere response codes or their absence. On the other hand, the main advantage of this method is that we can implement all intelligence at the sender's side.

Since the other alternatives were not applicable or had implications we were not comfortable with, we chose network side monitoring as the monitoring method for the development of our fuzzer framework. This is not all bad news, as the fuzzer framework can then be utilized on other devices as well since it does not consist of any HMU-specific implementation. Network side monitoring makes it universally applicable since every handset is required to reply to SMSs with the same response codes. However, we realize that our monitoring implementation can be extended given that we acquire the modem of the HMU. The `tcpdump` tool was used to capture

ongoing traffic between the HMUs and the base station during fuzzing sessions. A filter was applied to `tcpdump` to target only the GSM related processes. The resulting command was: `"sudo tcpdump -i lo udp port 'port' and not icmp -w 'filename'.pcap"`.

In order to parse and filter the resulting `pcap` files, the python library `Pyshark`[4] was used. It is a python library that allows for parsing and filtering using Wireshark dissectors. This makes it highly convenient for establishing the triggered response, or lack thereof, for each sent SMS.

7.2.1 Filtering the feedback from the HMU

Upon reception of a SMS message from the network, in practice, three things can happen. Either the SMS conforms to a known format and gets appropriately parsed by the modem. This results in an acknowledgment back to the network. If not, the SMS does not pass sanity checks in the modem, producing an error message, or an unexpected case is triggered. In this scope of this thesis work, we aim for the latter. There are multiple reasons why an SMS can trigger unwanted behavior in a receiving entity. For instance, the SMS can crash the modem due to logical errors in the software, thereby forcing system reboots, among many others.

To accurately detect crashes or unwanted behavior in the HMU, we had to devise intelligent heuristics that distinguish such behavior based on response codes or lack thereof. The desired outcome of this process would be a script that filters the `pcap` file after a fuzzing session and flag SMSs that potentially caused harm to the device. This means that legitimate error messages and acknowledgments produced by the HMU are of no interest and should be filtered out. Thus, in our monitoring script, we only flag unacknowledged SMS.

Several error messages are defined in the original GSM specification that needs to be filtered out [3rd00]. During our testing, the errors that were most frequently observed are `Protocol Error`, `Memory Exceeded Error` and `Invalid Mandatory Information`. When a `Memory Exceeded Error` occurs, we needed to take preemptive measures, which are described in section 7.6.3.

7.3 Input generation

Once we had finalized the setup of the appropriate architecture for sending arbitrary SMSs to the target IMSI, the next step was to generate test cases. To generate candidate test cases that are more likely to trigger bugs in the modem of the HMU, we have to revisit section 2.7.3. In general, we can conduct fuzzing in different degrees

[4] `https://kiminewt.github.io/pyshark/`, accessed on 22.04.2020

of "intelligence". A dumb approach with regards to SMS fuzzing would be to draw random values to construct a SMS PDU uniformly. However, for the most part, this would generate SMSs that do not conform to any known format or feature, and the OpenBTS software itself will likely drop them. Thus, there are more clever ways to construct these SMS PDUs.

Luckily, a library that can create arbitrary SMS PDUs already exist. The `Smspdu`[5] tool is a python implementation that offers full support of all data formats of the SMSs protocol and allows for the creation of virtually any SMS PDU. We created generic SMS packets from this library, and they served as a baseline for the subsequent fuzzing procedure.

7.3.1 Targeting header fields for fuzzing

Once we had decided upon the appropriate packet generating library, the next step was to decide which fields should be targeted for fuzzing. In Golde's master thesis from 2011 [Gol11] (which also formed the basis for the paper by Mulliner et al. [MGS11]), they developed a SMS fuzzing framework that could deliver fuzzed messages to feature phones. Golde's framework required SMS_SUBMIT formats, which differs slightly from the SMS_DELIVER format that we are working with. Nonetheless, we could take advantage of his findings when constructing fuzzing logic for our framework. Thus, based on his work, several header fields could already be ruled out from our fuzzer framework without conducting experiments ourselves. Most of the headers that we ignored are related to the delivery of the SMS, not the interpretation of it. To help visualize the explanation of the different header fields below, the example TPDU from section 2.7.4 is again presented in figure 7.2.

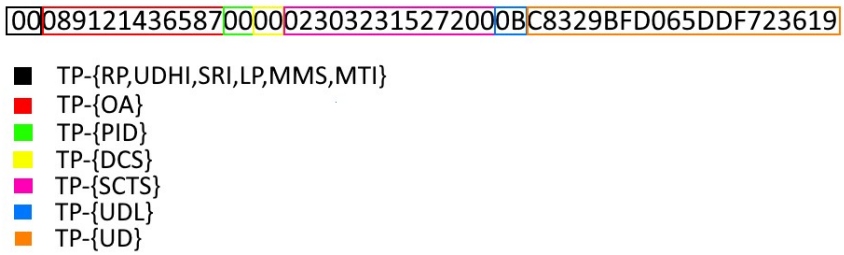


Figure 7.2: The dissected TPDU containing the payload "Hello World". Mapping between hexadecimals and header fields of the SMS SMS_DELIVER message format.

[5] `https://pypi.org/project/smspdu/`, accessed on 25.03.2020

Initially, the `TP-Message-Type-Indicator (TP-MTI)` is used to indicate what type of message we are constructing. Since we want to deliver SMS_DELIVER formatted messages. It does not make sense to fuzz this. Consequently, in our experiments, this field is set to `0`.

`TP-Reply-Path (TP-RP)` is used to tell the Short Message Entity (SME) that any reply for a specific message must use the same path as the original message, which can be useful if the originator and recipient networks are different. This field typically helps a mobile operator managing cost, but from a fuzzing perspective, it is irrelevant.

The `TP-Status-Report-Indication (TP-SRI)` bit is used to provide visual feedback of successful delivery to the sending entity [Gol11]. `TP-Originating-Address (TP-OA)` simply tells the terminating SME the address of the originating SME. `TP-Service-Centre-Time-Stamp (TP-SCTS)` is a timestamp of when the SMSC received the message. These fields will also be ignored in our experiments as we deem them unlikely to trigger vulnerabilities.

Theoretically, the `TP-User-Data-Length (TP-UDL)` field would be a natural candidate for fuzzing. However, in practice, it is not. According to Golde [Gol11], setting a different length indicator than the actual length of the data will only cause the GSM modems to discard the SMSs altogether. We later confirmed this ourselves, OpenBTS did not forward SMS that had invalid `TP-UDL`.

`TP-More-Messages-To-Send (TP-MMTS)` indicates whether the SME should expect more SMSs. As we do not believe altering this field can trigger any bug, this is set to the default value of `0`.

TP-User-Data (TP-UD), TP-Protocol-Identifier (TP-PID) and TP-Data-Coding-Scheme (TP-DCS)

Based on the above discussion, the only fields that are interesting to fuzz is the `TP-UD`, `TP-PID` and `TP-DCS`. Firstly, the `TP-PID` is referring to which protocol that is running in the layer above. Telefax and SIM-data download are only a few examples of the protocols that can be used. A value of `0` indicates a plain SMS message. `TP-DCS` defines the encoding of the user data field, the alphabet used, and the language. Moreover, it may also indicate a message class, which simply states if the message should be stored in the SME's storage or the SIM card. For SMSs, there exist the default 7-bit encoding, an 8-bit encoding, a 16-bit encoding, and reserved values. As there exists a variety of different coding schemes, the specific technicalities can be found in the original GSM specification [3rd99]. Based on the combination of these fields, the receiving SME will try to reassemble the SMS and potentially pass the user data to the right application.

For the TP-UD itself, we wanted to create payloads that are more likely to cause unusual behavior. Instead of sending purely random data, we leveraged the fact that there are many resources online to generate appropriate payloads. Therefore, a collection of specially crafted payloads known to be problematic to interpreters and, in general, poorly formatted, were imported from a Github repository[6].

The TP-User-Data-Header (TP-UDH)

All the different combinations of the three previously addressed fields are, however, not the entire SMS suite. To cover all features of SMS, the TP-User-Data-Header-Indicator (TP-UDHI) bit indicates whether or not the TP-User-Data-Header (TP-UDH) is present. This header is added to enrich the standard SMS formats, and it can be used to, for example, construct concatenated SMSs, add music, animations, colors, apply text formatting and much more. If the TP-UDHI bit is set to 1, the TP-UD contains a header following the layout presented in figure 7.3 (for default 7 bit encoding, that is). Furthermore, TP-UDH content and headers can be chained. This header greatly increases the complexity of the SMS service and SMS interpreters, thus a prime target for fuzzing.

The first field in the TP-UDH is the length field (TP-UDHL), indicating the total length of the header. As we can see from figure 7.3, TP-UDHL is followed by a sequence of one, possibly up to n, additional headers called Tag-Length-Value triplets. The triplets consist of an identifier, a length field, and the associated data (IEI, IEIDL, IED). The IEI number indicates what type of header it is, and how the associated data should be processed. For instance, an IEI value of 0x00 indicates concatenated SMS delivery and the resulting data contains transport-related information.

Since the TP-UDH feature introduces a vast range of additional functionality, it is not possible to target everything within this thesis project. Therefore, only the following TP-UDH features are targeted in this thesis project:

1. Concatenated SMS.
2. Enhanced Messaging Service (EMS).
3. (U)SIM data download.

[6] https://github.com/1N3/IntruderPayloads, accessed on 28.03.2020

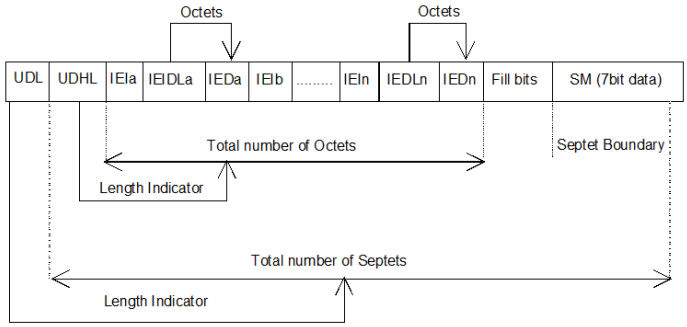


Figure 7.3: The layout of the `TP-UDH` in the `TP-UD` when the `TP-UDHI` bit is set to 1. This picture is taken from the original technical specification [3rd02].

7.4 Summary of the framework capabilities

At this point, a summary of what the framework is capable of is necessary. Firstly, we have built a fuzzing framework on top of a modified OpenBTS version. By running a simultaneous packet capture from the base station during ongoing communications with the HMU, we have established monitoring capabilities in the sender's edge. Based on the packet capture, we can filter out SMSs that might have triggered unusual behavior in the HMU. An overview of the outcome from the descriptions so far can be found in figure 7.4 on the next page, which is a diagram showing the architecture of the fuzzing system. In summary, our framework is capable of:

1. Injection of pre-encoded SMSs.
2. Delivery to any IMSI, thus not specific fuzzing to a HMU.
3. Delivery of malformed and semi-malformed formats.
4. High rate delivery of these SMS, which significantly outperforms what a regular phone is capable of.
5. Substantial monitoring capabilities in the sender's edge based on response codes and acknowledgments, or the lack thereof.
6. Fine-grained control of both the RPDU and TPDU layers.
7. Easy delivery through an appropriate API.

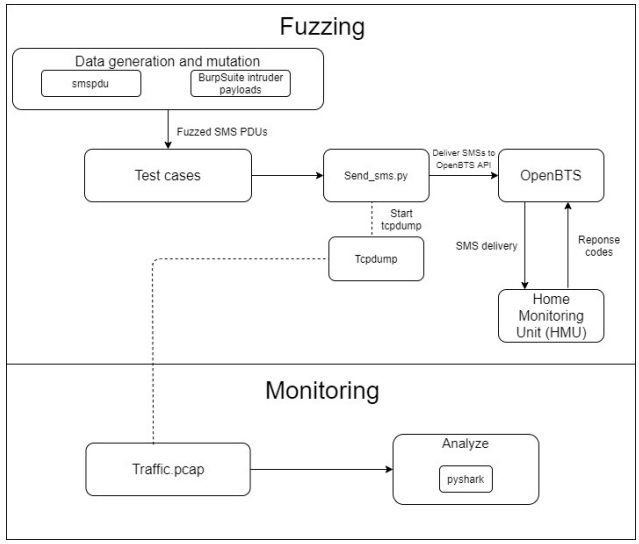


Figure 7.4: An architecture diagram of the fuzzing framework.

7.5 Fuzzing test cases

Based on the discussion in section 7.3 and inspiration from relevant literature [Gol11] [MGS11], multiple fuzzing test cases have been constructed. We group our test cases into two different categories, those aiming at fuzzing variations of `TP-UD`, `TP-PID`, and `TP-DCS`, and those aiming at targeting the `TP-UDH`.

7.5.1 Combinations of TP-PID, TP-DCS, and TP-UD

Random TP-PID, varying TP-DCS and semi-random TP-UD

The three test cases below aim to target combinations of random protocols and coding schemes that is supported by the SMS protocol, with semi-random payload, based on the work of Golde [Gol11].

1. A combination of random `TP-PID`, default `TP-DCS` (7-bit encoding, value of zero) and semi-random payloads.
2. A combination of random `TP-PID`/`TP-DCS` and semi-randomly generated payloads.
3. A combination of random `TP-PID`, `TP-DCS` using commonly used values in SMSs (0x00, 0x04, 0xF1, 0xF5)[7] and semi-random payload.

[7]Regular 7-bit encoding, regular 8-bit encoding, class 1 regular 7-bit encoding and class 1 regular 8-bit encoding. All encodings use default alphabets.

The code for these three test cases can be found in appendix C.4.

Flash SMS

Flash SMS, also called silent SMS or short message type 0, is a specific feature of the SMS suite and is indicated by a `TP-PID` value of `0x40` or a `TP-DCS` value of either `0xF0` or `0xF4`. These types of SMSs are special in the sense that they are not stored on the receiving cellular device. Also, on a normal smartphone, the content of the SMS would be displayed without user interaction, unless privacy settings are enabled. The primary use of flash SMSs is to check whether a subscriber's mobile phone is turned on or not. However, it is also used to deliver messages that require the recipients' immediate attention, for example, emergency messages from governmental agencies.

Flash SMS offers two advantages in a fuzzing experiment. Firstly, the SMSs are not stored on the device, circumventing the `Memory Capacity Exceeded` error. Additionally, according to Golde[Gol11], the code that renders flash SMSs could be different from the one displaying normal SMSs. This means another code-base with potential implementation flaws. However, since the HMU does not display any received SMS, we were not sure whether flash SMSs were parsed by a different code base, compared to normal SMSs.

Different combinations of the aforementioned values of `TP-PID` and `TP-DCS`, combined with semi-random payload, were constructed to create flash SMSs that were then delivered to the HMU. The code for this test case can be found in appendix C.4, and an example packet displayed in Wireshark can be found in figure 7.5.

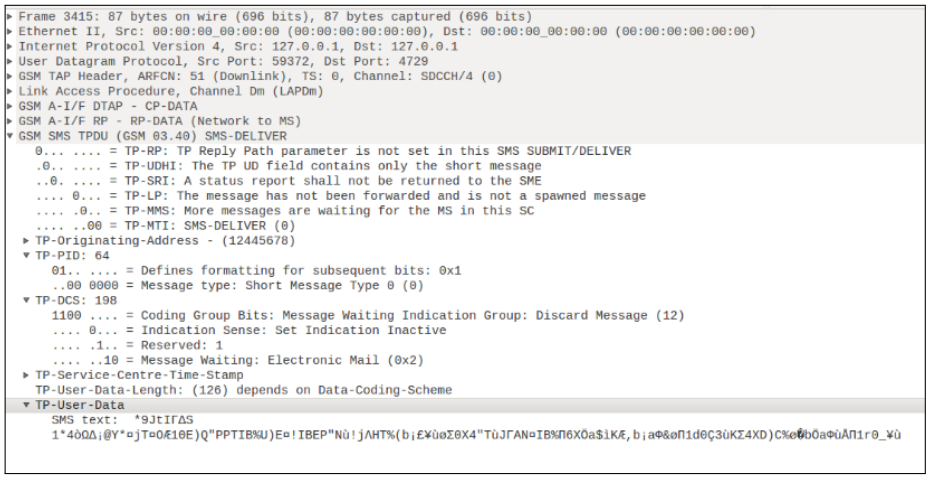


Figure 7.5: A screenshot of a fuzzed flash SMS captured in Wireshark.

7.5.2 Fuzzing the TP-UDH

As previously explained, the rationale for targeting the user data header in a fuzzing test is the broad range of additional features offered by this header. Many of the features are seldomly used and could trigger code paths not thoroughly tested.

Generic TP-UDH

First, we attempted to fuzz this header in a generic sense, not targeting any specific service delivered by it. Having the structure of a potentially valid `TP-UDH` in mind (see figure 7.3), the idea was to add up to 15 random header chunks in one SMS, which we based on previous work [Gol11]. The headers' chunks themselves had the following structure: the `IEI` identifiers took any random value between `0x00-0xFF`, and the `IED` data was constructed by adding between 2 and 8 random bytes, as portrayed in figure 7.7. A visualization of the final `TP-UDH` in our generic `TP-UDH` test case can be found in figure 7.6. For each header chunk, the `IEIDL` length indicator was set to the actual length of the data, as we do believe it is necessary to set it correctly for the receiver to parse the packet appropriately. Lastly, a random amount of bytes were appended as payload in the `TP-UD`. The code for this fuzzer can be found in appendix C.5.1, and a picture of an example packet in Wireshark can be seen in figure 7.8.

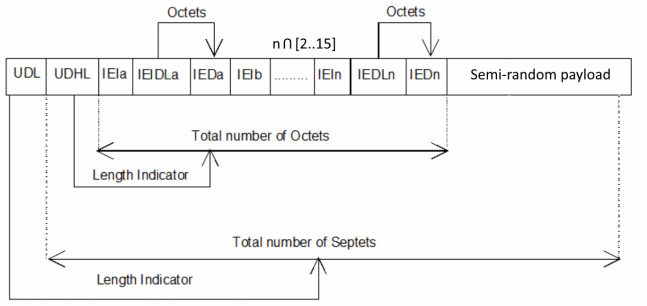


Figure 7.6: The structure of the `TP-UDH` that was created within our generic `TP-UDH` test case.

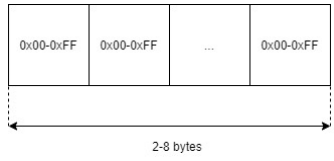


Figure 7.7: The IED field within each header chunk has a length of 2-8 bytes and takes any value.

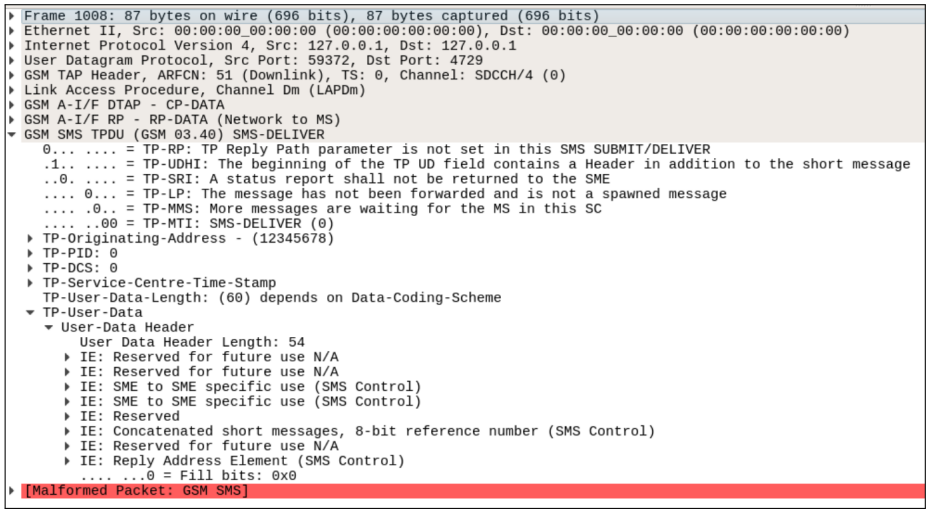


Figure 7.8: A screenshot of a fuzzed SMS in Wireshark, targeting random `TP-UDH` headers.

Concatenated SMS

Originally, SMSs were restricted to a maximum of 160 characters with 7-bit encoding and 140 characters with 8-bit encoding. Due to the multipart, or concatenated SMS feature, SMSs could be chained to send longer messages. We realized this through `TP-UDH` by splitting up longer payload to several messages and keeping track of transport information such as session identifiers and sequence numbers - not too different from how TCP works. At the receiver's end, the packets are reassembled. Concatenated SMS may consist of up to 255 unique SMSs to form longer messages. To illustrate, an example of a concatenated SMS session is highlighted in figure 7.9, presented in PDU format. A concatenated SMS is defined within the `TP-UDH`, conforming to the format below [3rd02].

1. `UDHL` equal to `0x05`.

2. `IEI` equal to `0x00`.

3. `IEIDL` equal to `0x03`.

4. `IED` data consisting of a one byte session identifier, one byte indicating the total number of SMSs in this session and one byte indicating the sequence number of the current SMS.

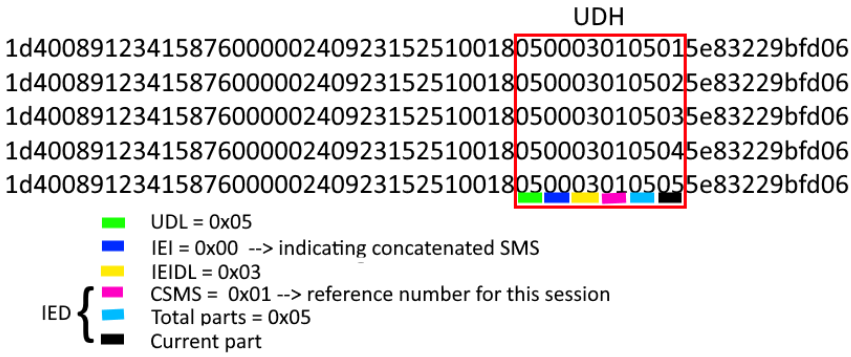
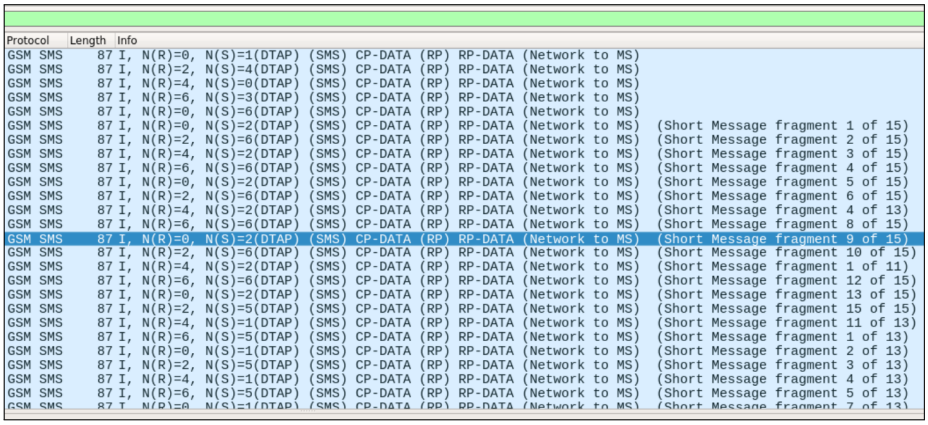


Figure 7.9: An example of a concatenated SMS session.

There exist multiple varieties of concatenated message sequences where the delivery and interpretation of the messages can cause odd behavior in the HMU. Our concatenated SMS test case aims to cover a wide range of problematic sequences and trigger edge cases. To realize that, this test case creates SMS sequences within the same multipart session, that consists of:

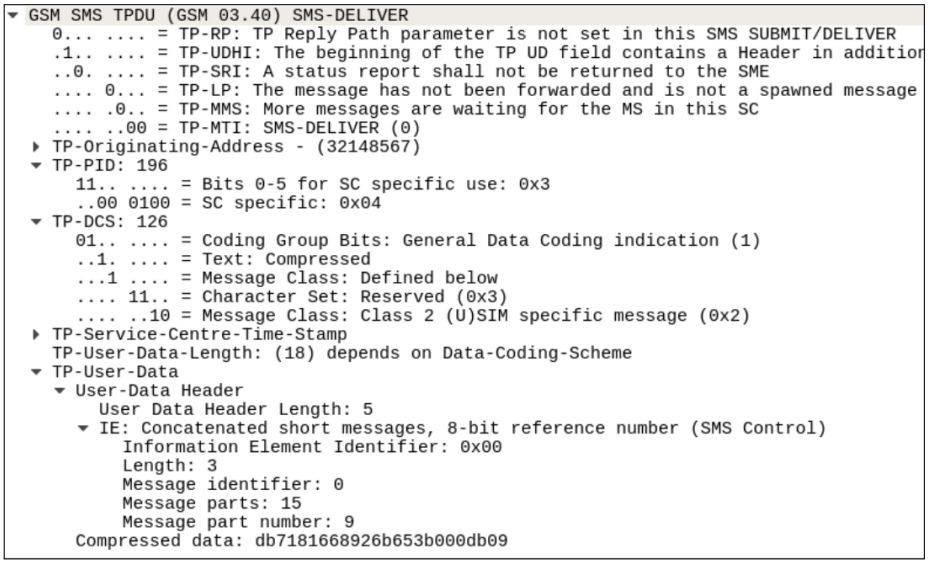
1. Different encodings (`TP-DCS`).
2. Different protocol identifiers (`TP-PID`).
3. Semi-random payload (`TP-UD`).
4. Missing sequence numbers.
5. Duplicate sequence number.
6. Sequence numbers exceeding the declared total number of multipart SMS.
7. Sequences of random size (up to 30 because of memory capacity at HMUs).
8. Out of order delivery.

The code for this test case can be found in appendix C.5.2. A screenshot from how it looks in Wireshark can be found in figure 7.10.



Protocol	Length	Info
GSM SMS	87	I, N(R)=0, N(S)=1(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS)
GSM SMS	87	I, N(R)=2, N(S)=4(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS)
GSM SMS	87	I, N(R)=4, N(S)=0(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS)
GSM SMS	87	I, N(R)=6, N(S)=3(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS)
GSM SMS	87	I, N(R)=0, N(S)=6(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS)
GSM SMS	87	I, N(R)=0, N(S)=2(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 1 of 15)
GSM SMS	87	I, N(R)=2, N(S)=6(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 2 of 15)
GSM SMS	87	I, N(R)=4, N(S)=2(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 3 of 15)
GSM SMS	87	I, N(R)=6, N(S)=6(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 4 of 15)
GSM SMS	87	I, N(R)=0, N(S)=2(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 5 of 15)
GSM SMS	87	I, N(R)=2, N(S)=6(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 6 of 15)
GSM SMS	87	I, N(R)=4, N(S)=2(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 4 of 13)
GSM SMS	87	I, N(R)=6, N(S)=6(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 8 of 15)
GSM SMS	87	I, N(R)=0, N(S)=2(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 9 of 15)
GSM SMS	87	I, N(R)=2, N(S)=6(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 10 of 15)
GSM SMS	87	I, N(R)=4, N(S)=2(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 1 of 11)
GSM SMS	87	I, N(R)=6, N(S)=6(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 12 of 15)
GSM SMS	87	I, N(R)=0, N(S)=2(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 13 of 15)
GSM SMS	87	I, N(R)=2, N(S)=5(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 15 of 15)
GSM SMS	87	I, N(R)=4, N(S)=1(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 11 of 13)
GSM SMS	87	I, N(R)=6, N(S)=5(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 1 of 13)
GSM SMS	87	I, N(R)=0, N(S)=1(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 2 of 13)
GSM SMS	87	I, N(R)=2, N(S)=5(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 3 of 13)
GSM SMS	87	I, N(R)=4, N(S)=1(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 4 of 13)
GSM SMS	87	I, N(R)=6, N(S)=5(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 5 of 13)
GSM SMS	87	I, N(R)=0, N(S)=1(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS) (Short Message fragment 7 of 13)

(a) SMS message fragments.



```
▾ GSM SMS TPDU (GSM 03.40) SMS-DELIVER
    0... .... = TP-RP: TP Reply Path parameter is not set in this SMS SUBMIT/DELIVER
    .1.. .... = TP-UDHI: The beginning of the TP UD field contains a Header in additior
    ..0. .... = TP-SRI: A status report shall not be returned to the SME
    .... 0... = TP-LP: The message has not been forwarded and is not a spawned message
    .... .0.. = TP-MMS: More messages are waiting for the MS in this SC
    .... ..00 = TP-MTI: SMS-DELIVER (0)
  ▸ TP-Originating-Address - (32148567)
  ▾ TP-PID: 196
      11.. .... = Bits 0-5 for SC specific use: 0x3
      ..00 0100 = SC specific: 0x04
  ▾ TP-DCS: 126
      01.. .... = Coding Group Bits: General Data Coding indication (1)
      ..1. .... = Text: Compressed
      ...1 .... = Message Class: Defined below
      .... 11.. = Character Set: Reserved (0x3)
      .... ..10 = Message Class: Class 2 (U)SIM specific message (0x2)
  ▸ TP-Service-Centre-Time-Stamp
    TP-User-Data-Length: (18) depends on Data-Coding-Scheme
  ▾ TP-User-Data
    ▾ User-Data Header
        User Data Header Length: 5
      ▾ IE: Concatenated short messages, 8-bit reference number (SMS Control)
          Information Element Identifier: 0x00
          Length: 3
          Message identifier: 0
          Message parts: 15
          Message part number: 9
      Compressed data: db7181668926b653b000db09
```

(b) The content of one message fragment.

Figure 7.10: An example of concatenated SMS captured in Wireshark.

Enhanced Messaging Service (EMS)

EMS was first defined in the GSM specification as an extension to SMS, offering the SMS suite new formatting [3rd19]. This extension *"may permit the message to contain animations, pictures, melodies, formatted text, and vCard and vCalendar objects"*. [3rd19]. The EMS extension is extensive and is implemented via the `TP-UDH` within the `TP-UD`. Since EMS itself offers vastly different formats, 31 different identifiers (`IEIs`) exists to accommodate EMS. The different headers (`IEIs`) follows separate formats which often must be combined with other headers, such as the concatenation

header. Considering the vastly different formats within EMS and the rapid growth of complexity when constructing correctly formatted packets, this thesis attempted to fuzz EMS in a more generic sense. An example of such a fuzzed packet can be seen in figure 7.11, captured in Wireshark. Within this test case, we create EMS packets in the following manner:

1. Randomly selects an `IEI` indicating EMS (`0x0a-0x1f`).
2. `IEIDL` between 1-7 bytes.
3. `IED` data consisting of random bytes. Number of bytes indicated by `IEIDL`.
4. Payload equal to 10-120 random bytes.
5. Random `TP-PID`.
6. Random `TP-DCS`.

The code for this test case can be seen in C.5.3.

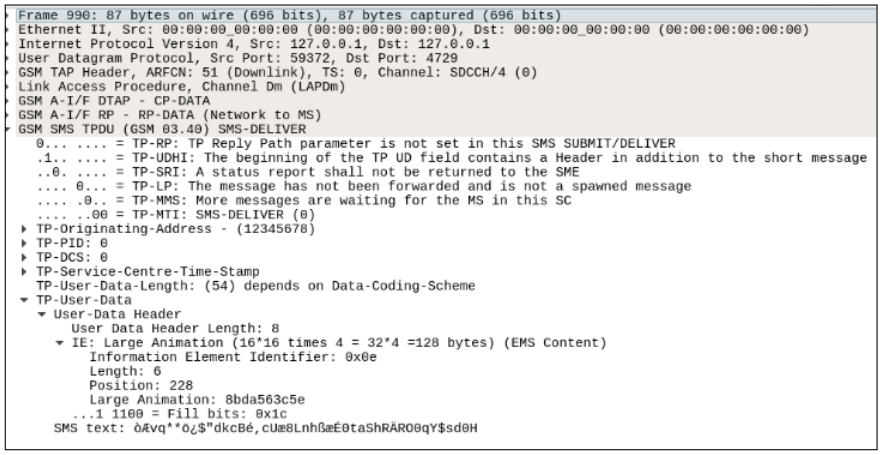


Figure 7.11: An example of a EMS SMS captured in Wireshark.

(U)sim data download

(U)SIM[8] Data Download is another feature of SMS that can be interesting from a fuzzing perspective as they interface directly with the SIM cards. According to the original specification, (U)SIM Data Download is a *"facility whereby the ME must pass the short message in its entirety including all SMS elements contained in*

[8]The character "U" is an abbreviation for universal.

the SMS deliver to the (U)SIM" [3rd02]. In other words, the SMS is passed to the (U)SIM-card without any processing of the SME itself. This feature of SMS is used to, for example, read or write data such as the IMSI of the SIM-card. The rationale of targeting this feature is thus to target another code base than the rest of the test cases.

The data in the request/responses from (U)SIM data downloads can be cryptographically secured, and follows the format of SIM Toolkit Security Headers [3rd20], which is present at the start of the `TP-UD` field. The details of the security header will not be explained in this thesis. To construct a correctly formatted SMS with such a header is not feasible, nor important for this thesis. Therefore the (U)SIM Data Download test case simply fills the `TP-UD` with random data, which is inspired by Golde [Gol11]. An example packet from Wireshark is shown in figure 7.12. In this project, the (U)SIM Data Download packets are created with the following:

1. `IEI` equal to `0x70`.
2. `IEIDL` equal to `0x00`.
3. `IED` is null.
4. `TP-PID` equal to `0x7F`, indicating (U)SIM Data Download.
5. `TP-DCS` equal to `0xF6`, indicating 8 bit encoding.
6. Random payload equal to 10-120 bytes.

The code for this test case can be found in appendix C.5.4.

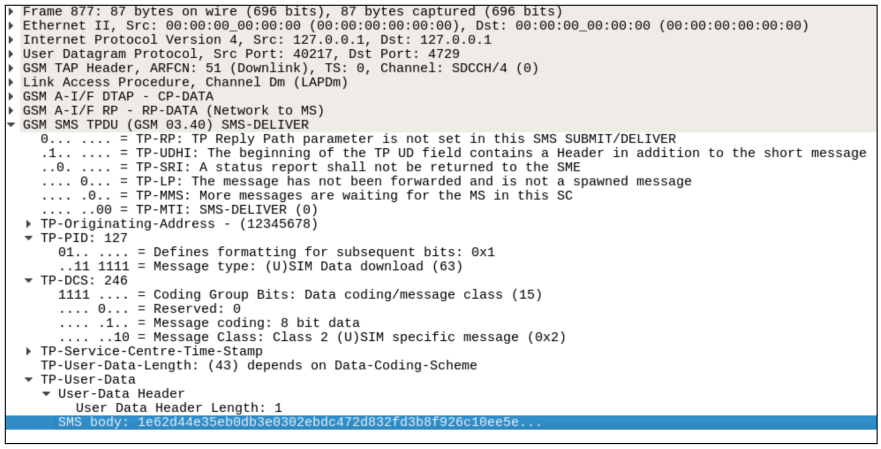


Figure 7.12: An example of a (U)SIM Data Download SMS packet in Wireshark.

7.6 Testing

7.6.1 Preliminary SMS testing

Relating back to RO_2, given that it states: *"To investigate potential vulnerabilities in the SMS interface of the Cardiomessenger 3G Smart through fuzzing"*, the natural first step in this security test is to look online for already published vulnerabilities. Since we found the exact version of the modem during our first scope of this thesis, this part was straightforward. If there indeed existed vulnerabilities in the firmware of the modem, we would attempt to exploit it on the HMU itself. However, there were no vulnerabilities related to this modem published on the internet.

Next, we conducted experiments where arbitrary SMSs were sent to the HMU to look at the responses it created. The tests indicated that the HMU acknowledges messages on the relay layer instead of the transport layer, within the SMS protocol stack. Moreover, the HMU appeared to conform to the response scheme defined within the original GSM-specification [3rd02]. An interesting finding done in the initial phases of the security test revealed that after 30 consecutive SMS-deliveries, the memory of the HMU is exhausted. Thus, the subsequent response codes generated by the HMU, were all `Memory Exceeded Error`. We found that in practice, this means that the modem does not parse the incoming message, but simply discards it. This constitutes a major hindrance to the effectiveness of the fuzzer framework. Since only 30 SMSs can be sent consecutively, the throughput is severely restricted.

We then conducted several tests on the stability and robustness of our pipeline. After establishing a radio channel between our base station and the HMU, our framework immediately started to send fuzzed messages. This continued for roughly one minute before the radio channel tore down. Seemingly, the stability issues are more related to the HMU than our setup and lab environment, as tests done on other devices continued for virtually unlimited time. In general, this behavior is symptomatic of problems beyond our control. It also drastically reduces the effectiveness of our testing. In practice, this opens a small one minute time frame where fuzzed messages can be delivered to the HMU. A physical reboot of the device is required for a consistent reconnection to our base station and is, therefore, a large bottleneck for the entire fuzzing project. There could be multiple reasons why the HMU tears down the radio channel after being connected for only one minute. We believe that it is battery conservative due to its limited capacity and consequently would not waste energy by staying connected more than strictly necessary.

As sending hundreds of thousands of fuzzed messages to the HMU is not realistically feasible in a consistent manner due to the short time window described above and due the memory capacity problem, we decided to order a copy of the modem directly from a modem supplier. Since this is the same chip as what is integrated on

the HMU, the same baseband stack/firmware will be scrutinized when fuzzed. Hence, if we find a vulnerability by testing the modem directly, the same vulnerability may exist on the HMU. Because the exact product number was retrieved during the first scope of our thesis, ordering the right modem was straightforward. When equipped on a specially designed developer board, which we also ordered, we are in complete control of both sides of the communication channel. The stability and robustness issues are, therefore, alleviated. However, the memory problem remains.

7.6.2 Setting up the modem

The ordered version of the modem is not technically identical to the one found on the HMU. The Telit HE910-D, which is integrated on the Cardiomessenger 3G Smart, was out of stock at the time. Due to time restrictions, we decided to order the Telit HE910-G, which is a very similar modem. They both belong to the same product family, and the difference in functionality is that the Telit HE910-G offers GPS functionality. It is common for manufacturers to reuse their code-bases on similar products, and we hypothesize that the entire SMS-stack is identical for the two products. If we are correct, testing on a slightly different product will have no implications for our project. The Telit HE910-G chip used in our experiments was running firmware version 12.00.006. Ideally, the firmware should be updated to that running on the HMU's modem, which was identified in section 6.2.3 as 12.00.024. However, we were unsuccessful in updating it.

Additionally, a suitable adapter was ordered, which enables a connection from the modem to a laptop using a standard USB connection. To interact with the connected modem, the USB-driver intended for the operating system on the laptop, has to be installed. Thereby, we could use any software which is capable of interacting with a serial connection. The baud rate of the modem is 115200. In the experiments in this thesis, we used Telit's own software to interact with their modem over a USB connection, namely the `Telit AT Controller`.

To realize a connection between the modem and our GSM-network, we had to insert a SIM-card. Given that the SIM-card of the Cardiomessenger 3G Smart did not fit, the SIM-card of the Cardiomessenger II-S GSM-version was inserted into our modem.

Interacting with the modem is virtually the same as issuing AT-commands directly to it and obtaining their output. AT-commands are modem-specific executable commands, which enable network connection, SMS, dialing, and hanging up. Typically, higher-level applications of embedded devices are responsible for issuing AT-commands to the modem, thereby commanding the modem to perform specific actions. Before conducting any fuzzing attempts, we instruct the modem directly to connect to our GSM network. We execute the following sequence of AT-commands

in order to realize the network connection. The AT-commands are taken from the AT-command reference guide for the specific Telit product family [Sol16].

```
AT+CREG?                #Check the current status of network connection
CREG: 0,0               #Networking not enabled, not connected
AT+CREG=1               #Enable networking
CREG: 1,0               #Networking enabled, not connected
AT+WS46=12              #Specify GSM as access technology
AT+COPS?                #Check current network selection method
COPS: 1                 #Manual selection enabled
AT+COPS=?               #List available networks, including ours
AT+COPS=1,2,90170       #Manually connect to our network, by specifying MCCMNC
AT+CREG?                #Check the current status of network connection
CREG: 1,2               #Networking enabled, attempting to connect
AT+CREG?                #Check the current status of network connection
CREG: 1,1               #Networking enabled, connected
```

Code Listing 7.1: AT-command sequence for registration to our network.

Initially, a challenge was that the AT-command sequence in listing 7.1 above did not result in a successful connection. The modem did detect our network but was repeatedly unsuccessful in connecting to it. As it turned out, the signal strength of our network did not align with the capacity of the antenna of the modem. This was remediated by connecting a peripheral antenna to the modem, increasing its receiving and transmission power. Figure 7.13 shows how the setup and how the antennas are connected onto the modem.

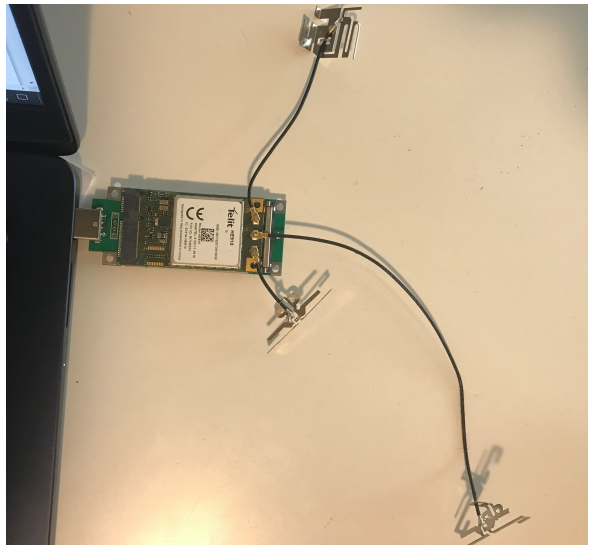


Figure 7.13: The setup of the modem in its developer board, with antennas connected onto it.

7.6.3 Implementing health checker and emptying SMS memory

Being in complete control of both sides of the communication channel brings additional benefits to the fuzzing framework. Since the current implementation of the fuzzer framework does not have any means of detecting unusual behavior before an entire fuzzing session is completed (which can span several hours), we implemented functionality to send health checks regularly to OpenBTS from the modem. Such health checks were not obtained through the debug/diagnostic ports of the modem, as we were not successful in connecting to those ports, implying that they might be disabled after production. Thus, we had to infer the condition of the modem otherwise. The health check is based on issuing an AT-command to the modem on the serial port and consequently checking whether it replies as expected or not.

The computer that has the modem attached using the development board sends TCP/IP packets containing the health status of the modem to the computer running OpenBTS. Hence OpenBTS can stop sending packets when it is notified of unusual behavior. This shifted the capabilities of OpenBTS from being insensitive to the recipient's status during a fuzzing session to a live health checker. The code for this can be found in Appendix C.3.1. Implementing a health checker of the modem within our fuzzing framework, effectively extends our network side monitoring capabilities, as mentioned in section 7.2.

Another advantage of controlling the recipient's modem was that SMSs could be deleted once they filled up the memory on the SIM-card. Like for the HMU, a maximum of 30 SMSs could be stored on the modem. Therefore, in the experiments conducted in this thesis, the memory was regularly emptied on the modem whenever it reached maximum capacity by issuing the AT-command `AT+CMGD=1,4`. This countered the problem of exhausting the memory. Figure 7.14, shows the final architecture of our fuzzing framework, now targeting the modem of the HMU in isolation. In order to complete the fuzzer framework, additional utilities and helping-functions have been implemented. These are found in appendix C.6.

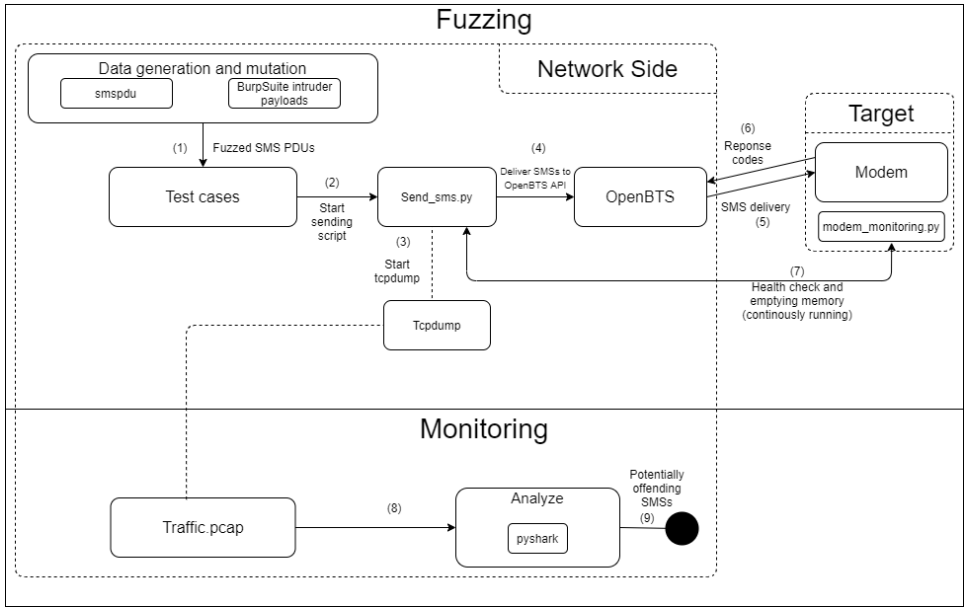


Figure 7.14: The complete architecture of our final fuzzing framework. The numbers indicate the logical execution order of our fuzzer framework.

7.6.4 Procedure for executing test cases

In this section, the exact procedure we followed when executing the different test cases is described. This procedure was the same for all test cases. The list below was followed for each case:

1. Run traffic capture with `tcpdump` with the following filter: `"sudo tcpdump -i lo udp port 'port' and not icmp -w 'testcase'.pcap"`.
2. Run the script to send all PDUs within a test case, which can be found in Appendix C.2. Simultaneously start health-checker at receiver side.
3. After the script is finished, run the analysis/monitoring script, which flags potentially offending SMSs based on the pcap file provided from step 1. This script can be found in Appendix C.3.2. It outputs a text file containing the flagged PDUs.
4. Reiterate steps 2-3, now with the output file containing possible offending PDUs as input to the send script.
5. If the results are consistent for both iterations, force the HMU to connect to your network, using the jammer. Send the flagged PDUs to the HMU while running a traffic capture and analyze the result.

Because of the general instability of the software used, some SMSs might never receive an acknowledgment. These SMSs are going to be flagged as potential offending packets in our analysis script, but in reality, they are more likely to be false-positives. Step 4 is thus of great importance and ensures the validity of our results. We will generate knowledge concerning the HMU itself in step 5.

7.7 Results and Analysis

This section provides an analysis of the experiments we conducted within scope 2 of this master thesis. Below, we present each test case, with their corresponding results. The analysis is largely based on the monitoring method that was established in section 7.2, and further extended in section 7.6.3. Due to unknown software bugs, OpenBTS seems to drop a few packets before they reach the air-interface. That is the reason why not all test cases sent an equal amount of SMSs. This problem is not a real issue, and it simply results in that some PDUs are never sent. Since our fuzzing framework send thousands of packets, losing a couple of them does not drastically reduce the probability of success.

In summary, we stumbled upon certain odd behaviors that turned out to have implications for our testing. We discovered that whether a SMS is stored on the SIM-card or not, could influence the way the recipient interprets it. Effectively, this meant that some PDUs only triggered bugs when the modem had available memory capacity. This behavior implies that when a SMS is stored on a SIM-card, it reaches other parts of the firmware - which might also contain bugs. This could also imply that the modem simply discards the SMS before parsing it all together when the memory is full. Hence, for the effectiveness of this project, emptying the memory buffer once it was full, turned out to be essential.

Even more surprisingly, for certain PDUs, the SIM-card itself could be a decisive factor. During preliminary testing, a Sysmocom SIM-card was used, which is a SIM-card that is not affiliated with any operator. A handful of bugs were triggered for many types of SMS PDUs when using this specific SIM-card. However, almost none of these bugs were present in the HMU. This forced us to rethink our experiments. Inserting one of the HMUs' SIM-card into the modem turned out to be key. In subsequent testing, bugs that were found in the modem had a much larger probability of being present in the HMUs as well. Apparently, the way the SIM-cards are programmed seems to have some implications for how the modem interprets each message.

In total, 14 different SMSs were found during our testing that interferes with the HMU's regular operation. All the produced crashes were tested multiple times using different SIM-cards - confirming that it is the firmware of the modem and not the

SIM-card, that suffers from lousy input validation.

7.7.1 Results of random TP-PID and semi-random TP-UD, regular 7-bit TP-DCS

In all, 13900 PDUs that belonged to this test case was sent to the modem. All the messages were either acknowledged or got an error. The errors seem to conform to the standard formats of the GSM-specification. Hence, none of the sent SMSs were flagged as potential offending SMS by our monitoring script. To this end, we believe that the SMS-interface of the HMU is capable of receiving such fuzzed messages without it interfering with its regular operation.

7.7.2 Results of random TP-PID, TP-DCS and semi-random TP-UD

Our monitoring script flagged three SMSs from this test case. After sending them to the HMU, we could confirm that they also crashed it. One of the offending SMSs-messages can be seen in PDU format below:

018603a10000009700089121 44465879c2a02500201841500864d6f7a696c6[REDACTED]

In figure 7.15, part of the network capture between the mobile network and the HMU is displayed. The offending SMS never receives an acknowledgment, and the subsequent frames simply show OpenBTS broadcasting different system information. After numerous system information broadcast messages, OpenBTS tries to issue several disconnection requests because the data channel is no longer active [3rd08], as seen in figure 7.16. However, this request never receives any acknowledgment, and after some time, OpenBTS stops sending these requests and assumes that the recipient went offline.

```
127.0.0.1    LAPDm     81 S, func=RR, N(R)=0
127.0.0.1    GSM SMS   87 I, N(R)=0, N(S)=0(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS)
127.0.0.1    LAPDm     81 U, func=UI(DTAP) (RR) Measurement Report
127.0.0.1    LAPDm     81 I, N(R)=1, N(S)=0(DTAP) (SMS) CP-ACK
127.0.0.1    LAPDm     87 S, func=RR, N(R)=1
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 6
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 5
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 6
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 5
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 6
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 5
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 6
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 5
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 6
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 5
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 6
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 5
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 6
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 5
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 6
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 5
127.0.0.1    LAPDm     87 U, func=UI(CCCH) (RR) System Information Type 6
```

Figure 7.15: Network traffic capture showing that the SMS never receives an acknowledgement from the HMU and that the HMU does not send any data afterwards.

127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	87	U P, func=DISC

Figure 7.16: Several disconnection requests from OpenBTS that are never ackowledged from the modem of the HMU.

After sending this particular SMS, the HMU is effectively forced off our network and thus denied in receiving or sending any further information. Figure 7.17 shows how an actual detach procedure from the network should occur, which looks nothing like the traces in figure 7.15 and 7.15.

Destination	Protocol	Length	Info
127.0.0.1	LAPDm	87	U, func=UI
127.0.0.1	LAPDm	87	U, func=UI(CCCH) (RR) System Information Type 6
127.0.0.1	LAPDm	87	U, func=UI(CCCH) (RR) System Information Type 5
127.0.0.1	LAPDm	81	U P, func=SABM(DTAP) (MM) IMSI Detach Indication
127.0.0.1	LAPDm	87	U F, func=UA(DTAP) (MM) IMSI Detach Indication
127.0.0.1	LAPDm	87	U P, func=DISC
127.0.0.1	LAPDm	81	I, N(R)=0, N(S)=0(DTAP) (RR) GPRS Suspension Request
127.0.0.1	LAPDm	81	U F, func=UA

Figure 7.17: A proper message sequence of the GSM detach procedure.

The dissected TPDU of the offending packet is shown in figure 7.18. It consists of a random protocol identifier, a random coding scheme, and semi-random payload as described in section 7.5.1. The payload is equal to the string: `"Mozilla/5.0 (Linux; U; Android 2.0; en-us; [Redacted]"`.

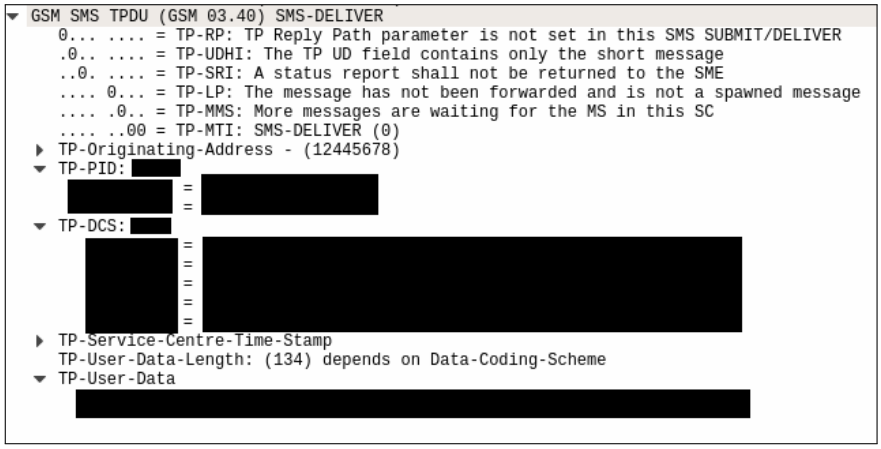


Figure 7.18: The dissected TPDU.

To check if the microcontroller of the HMU could get any response from the modem at all, we tried to simulate a similar testing scenario on the purchased modem. After sending the offending SMS, we launched a sequence of AT-commands to the modem over serial communication. By monitoring the USB interface of the port that the modem is connected to, we could infer how the modem behaved. Figure 7.19 shows this message sequence, captured in Wireshark. After issuing AT-commands, the modem never replies, indicating a potential full crash of the chip. Under normal conditions, we would expect to see `URB_BULK in` as responses to our requests. It was first when we reinserted the USB stick that the modem responded again.

25	69.598540	host	1.11.8	USB/AT	36 URB_BULK out, Sent AT Command: AT+COPS?
26	69.598671	1.11.8	host	USB	27 URB_BULK out
27	84.511901	host	1.11.8	USB/AT	36 URB_BULK out, Sent AT Command: AT+CREG?
28	84.512050	1.11.8	host	USB	27 URB_BULK out

Figure 7.19: USB monitoring. The modem does not respond to any AT-commands.

During our experiments, we observed that the HMU did reassociate itself again with the network after 15-20 minutes. This observation was slightly surprising as the modem we ordered had to be physically rebooted for it to read and execute AT-commands again. Following that rationale, the microcontroller might issue the modem to be active only for specific time intervals and shut down for the rest. This substantiates our assumptions regarding the HMU's conservative battery usage, which we described in detail as a severe bottleneck in the preliminary testing section (7.6.1). It is thus likely that the modem of the HMU is shut down for the most part and only maintains network connectivity for specific periods after the microcontroller issues a reboot.

7.7.3 Results of random TP-PID, semi-random TP-UD, and common TP-DCS values

In total, 13900 SMSs were sent within this test case. Our monitoring script flagged 23 SMSs as potentially offending test cases. After re-sending them to the modem, they were handled according to the GSM-specification. Hence, in reality, they were false-positives, and we can conclude that the modem is capable of handling this type of fuzzed SMSs.

7.7.4 Results flash SMS

In this test case, all SMS-messages triggered either an acknowledgment or an error. We sent a total of 11491 SMSs, and they all got responses. Considering that flash SMS-messages are used to attract the user's immediate attention by displaying them directly on the screen, they are perhaps not ideal fuzzing candidates on typical IoT-devices - as they typically do not have displays. Since neither the modem coupled on the development board nor the HMU have any way of displaying SMS-messages, we believe that the modem discards flash SMSs directly after parsed them.

7.7.5 Results generic TP-UDH

After running our analyze script on the network capture during this fuzzing session, there were several SMSs that seemingly caused trouble to the modem. Following step 4-5 in the procedure described in 7.6.4, it was evident that all of the flagged SMSs also caused crashed the HMU's modem, identical to how it is described in section 7.7.2. After sending 1000 SMSs, 11 crashes were detected. Conclusively, this test case is by far the most challenging for the modem of the HMU. We decided to not search for further vulnerabilities in this section, as the SMSs that crashed the modem of the HMU are symptomatic of fundamentally weak input validation when multiple `TP-UDHs` are chained. Figure 7.20 shows the corresponding traffic dump.

```
LAPDm      81 U, func=UI(DTAP) (RR) Measurement Report
LAPDm      81 S, func=RR, N(R)=3
GSM SMS    87 I, N(R)=0, N(S)=3(DTAP) (SMS) CP-DATA (RP) RP-DATA (Network to MS)
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 5
LAPDm      81 I, N(R)=4, N(S)=0(DTAP) (SMS) CP-ACK
LAPDm      87 S, func=RR, N(R)=1
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 6
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 5
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 6
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 5
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 6
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 5
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 6
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 5
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 6
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 5
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 6
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 5
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 6
LAPDm      87 U, func=UI(CCCH) (RR) System Information Type 5
```

Figure 7.20: Network traffic capture showing that the SMS never receives an acknowledgement from the HMU.

Below in figure 7.21, the one of the offending TPDUs is dissected in Wireshark. It consists of a random protocol identifier, a random coding scheme, and multiple random user data headers, as described in section 7.5.2. The payload itself is of zero length. In PDU format, it looks like the following:

014903a1000000354008912143658 70cfb02500201641000241ec002a637[REDACTED]

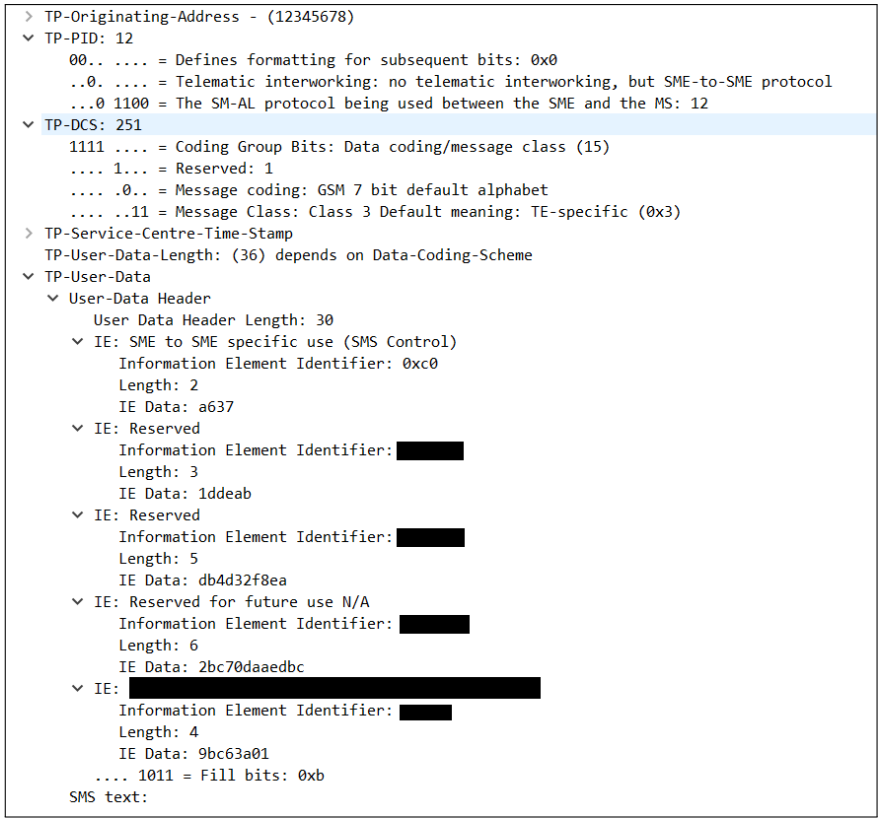


Figure 7.21: The dissected SMS in Wireshark.

7.7.6 Results concatenated SMS

In total, we sent 400 concatenated SMS sequences. Each message sequence consisted of between 2-30 SMS fragments, amounting to a total of 6300. A single fragment was not acknowledged by the modem, and the remaining SMSs fragments either got an acknowledgment or an error. After resending the unacknowledged SMS, it was acknowledged and did not result in any odd behavior. Consequently, based on the sample test cases, the firmware of the modem handled concatenated SMS without troubles.

7.7.7 Results EMS SMS

Within this test case, we sent 13238 SMSs, and two of them did not receive an acknowledgment. After reiterating the test, we found that they were false-positives as they properly got acknowledged in the second test. Since the formats of the different EMS features are vastly different from each other, we attempted to fuzz EMS in a generic sense that did not take any of its structures into account. That could be the reason why this test case did not uncover any vulnerabilities. Based on the random generation of packet payloads, the testing has probably not triggered typical edge cases that are more likely to contain bugs.

7.7.8 Results (U)SIM data download

A total of 13932 (U)SIM data download SMSs were sent to the modem within this experiment. All the messages were acknowledged. Thus no bugs were found nor any false-positives. This finding indicates that the SIM-Toolkit Security Header is a feature within the cellular firmware that is implemented with security in mind, properly handling malformed input. Considering this is a feature that can offer integrity verification and encryption, this is not very surprising. Likewise, for EMS, the structure of the SIM-Toolkit Security Header was not explicitly taken into account in the generation of the packets. Therefore, we can question how effective this test case was in reality.

7.7.9 Validation

Because we do not have access to diagnostic ports on the modem, it is essential to ensure the validity of our findings by other means. It is equally crucial to document our findings in a reproducible and transparent manner, such that the vendor has the opportunity to reproduce and confirm our findings. To ensure that the bugs found in this project are not contingent, we tested all the flagged SMSs on the three different Cardiomessenger 3G Smarts at our disposal. Similarly, we utilized four different SIM-cards within our testing. All the devices behaved similarly and did not acknowledge the offending SMSs. This removed the probability that specific configurations or settings within each HMU or SIM-card trigger the bug: consequently, the firmware of the modem is the real culprit.

7.7.10 Summary

In general, after an entire fuzzing session, it was frequently observed that the modem did not run as smooth as before the session - independent on whether a bug was triggered or not. Moreover, we can extrapolate a slight trend from the test runs. The modem tends to handle the simplest test cases adequately, whereas more intricate headers and features of the SMS-suite tend to cause more problems. By simple test

cases, we mean the features and coding schemes of the SMS-protocol that are popular and heavily used. These test cases never influenced the HMU's regular operation. Table 7.1 shows the number of sent SMSs within each test case and the corresponding number of offending SMSs. In total, our experiments entailed sending 87713 SMSs to the purchased modem and uncovered 14 SMSs that caused the modem of HMU to crashed.

On the other hand, intricate features and headers are more likely to cause irregularities. These features are typically offered by the `TP-UDH`, which rapidly increases the complexity of the SMS structure. Besides, these features are seldomly used. Especially combining different `TP-UDHs` that were typically not designed to be combined seems to be problematic. Thus, we hypothesize that features and coding schemes of the SMS-protocol that are less common are more likely to contain bugs.

Testcase	Number of sent SMS	Number of offending SMS
Random TP-PID, semi-random TP-UD, reuglar 7-bit TP-DCS	13900	0
Random TP-PID, random TP-DCS and semi-random TP-UD	13952	3
Random TP-PID, common TP-DCS and semi-random TP-UD	13900	0
Flash SMS	11491	0
Generic TP-UDH	1000	11
Concatenated SMS	6300	0
EMS SMS	13238	0
(U)SIM data download	13932	0

Table 7.1: Summary of results from the constructed test case in the second scope of this master thesis.

Chapter 8
Discussion and mitigation

This chapter aims to address and elaborate on the findings produced within chapters 6 and 7. Based on the results, we will describe which vulnerabilities these findings give rise to. Additionally, this chapter will provide a thorough discussion on what those vulnerabilities imply for patients and the vendor, and elaborate on which attacks are feasible when those vulnerabilities are exploited. The threat model, which is further outlined in section 2.2, will be used to contextualize the discussion.

8.1 Discussion on findings from hardware security testing

The initial research carried out was within RO_1, where we conducted a thorough hardware security test of the Cardiomessenger 3G Smart HMU. We knew that the hypotheses in table 1.1 were true for previous versions of HMUs from the same vendor. Consequently, by incorporating our black-box hardware testing methodology, which is further outlined in section 4.2.1, we constructed testing scenarios for all the hypotheses in table 1.1 that lie within our scope.

Our initial finding was that the device could be opened. Meaning that both the plastic cover and the metal plate partially concealing the PCB, could be removed without inflicting damage to the PCB itself. Although this may not seem like a groundbreaking finding within hardware security testing, this is where the bad security practices in the HMU start. Due to this finding, a magnitude of other testing possibilities opened up for us, which in turn, yielded findings of their own. All subsequent findings and misconfigurations would not have been obtained if the hardware of the HMU was tamper-proof. For a potential malicious attacker that has physical access to the HMU, the case is no different. A broader attack-surface opens up when the device can be easily opened, since the PCB is not damaged afterward.

Next, we identified all components and chips on the PCB, and by searching online for their serial numbers, which are written onto the components and chips themselves, we obtained the datasheet of all of them. The fact that all components and chips

on the PCB were identifiable, and that their respective datasheets were obtained, is quite reasonable. Biotronik is not a manufacturer of microcontrollers, modems, RAM or flash chips, so naturally, they will purchase such hardware from other companies. From a security perspective, this is not a bad practice in itself. However, this finding goes to show that a variety of information regarding the hardware of the HMU is easily obtainable. A malicious attacker can gather such information, and possibly utilize it to uncover further vulnerabilities, which in turn can be exploited in potential attacks which compromise the most critical assets within the pacemaker ecosystem.

In addition, we quickly located JTAG and UART interfaces, as they are labelled on the PCB. The labelling of JTAG and UART interfaces on the PCB, is a very bad practice from a security perspective. This completely rules out the need for a trial-and-error approach when locating the debugging interfaces, and we were quickly able to test if those interfaces were enabled on the HMU. Here, Biotronik seems to have taken a step backward, since the debugging interfaces are not labeled on the previous version of their HMUs. Only the JTAG debugging interface was enabled on the Cardiomessenger 3G Smart, in our version of the firmware which is not the latest, which we assume is 1.20. Nonetheless, disabling both debugging interfaces after production would have increased the hardware security of the device drastically.

The fact that the HMU can be easily opened, that its hardware components are identifiable, that its debugging interfaces are labeled and that its JTAG interface is enabled, are not findings which correspond to vulnerabilities in themselves. Nevertheless, these findings function as stepping stones that further enables attackers to compromise patients' safety or privacy potentially. Given that this is a medical device responsible for handling private patient data and interaction with the patients' pacemaker, we deem these findings as severe.

Using JTAG and Flashrom, we were able to extract the firmware of the HMU along with all other memory modules. Everything we extracted from the HMU turned out not to be obfuscated or encrypted in any way. The use of logging functionalities within the firmware also gave several hints regarding its structure. In addition to that, credentials were obtained in cleartext. Moreover, the lack of encryption, in combination with the use of logging functionalities in the firmware, dramatically eases the process of reverse engineering. This opens the possibility for an attacker to gain in-depth knowledge and to further disclose vulnerabilities in the protocols used.

Unencrypted firmware in a device which handles private patient data, and which communicates with patients pacemakers, is a critical finding. This finding directly corresponds to a vulnerability which is pivotal in the context of developing a sufficiently secure pacemaker ecosystem. From previous security research on HMUs, for example, the work by Muddy Waters and MedSec Inc [Blo16], multiple attacks

have been demonstrated against devices like this. In their research, they launched a battery-draining attack against a pacemaker, among other attacks, such as increasing the pacing rate of the pacemaker to dangerous levels by utilizing vulnerabilities in the HMU. Taking our findings on the Cardiomessenger 3G into consideration, whether Biotronik secures their products more extensively than other manufacturers, is unknown. Having access to the firmware and logging functionalities in their HMU would significantly reduce the amount of work required for an attacker to implement attacks like that.

8.1.1 Implications for patients

Within our threat model in section 2.2, we identified several assets and threat actors within the pacemaker ecosystem. Also, we outlined potential attacks, which a threat actor might wish to carry out, in order to compromise an asset. The most important assets within our threat model are arguably patient privacy and safety. After all, the entire pacemaker ecosystem is ultimately concerned with patient care. The most significant threat actor within our treat model is organized crime, as other threat actors lack the necessary motivation or knowledge to carry out attacks on the ecosystem. Consequently, the following discussion gives grounds for what the vulnerabilities might entail for patient privacy and safety when organized crime is the most significant threat actor.

First and foremost, to fully ensure that patient privacy and safety is protected when the HMU takes part in the pacemaker ecosystem, the vendor should guarantee the hardware security of the device. Our findings clearly show that this is not the case. If a potential malicious attacker were to acquire physical access to a patient's HMU, she could open the device, acquire a variety of information regarding its design, and connect to its enabled JTAG interface. Thereby, the attacker can easily dump the unencrypted firmware of the device and other memory modules, which contains debugging strings and cleartext credentials. Relating back to research question RQ_1, it is apparent that when our findings are used in combination, the Cardiomessenger 3G Smart not sufficiently secured.

Since it is not sufficiently secured, the confidentiality, integrity, and availability of patient data are at risk. Given that our findings encompass having access to the HMUs unencrypted memory, it is very likely that a potential attack breaches the confidentiality of patient data. When considering the unencrypted firmware in combination with our other findings, it is unlikely that proper encryption mechanisms are in place on all segments within the HMU, from the data arrives until it leaves. This leads to that data sent from a patient's pacemaker is most likely stored unencrypted in the HMU's memory at some point. An attacker who has acquired physical access to a HMU could thereby obtain such confidential patient data by making use of our

memory access finding.

Furthermore, the integrity of patient data is also at risk, given that our findings encompass an enabled JTAG interface. We deem it likely that a malicious attacker that can set up a JTAG connection to the device can also modify data that is to be sent to the vendor's backend servers, as JTAG has the functionality of writing individual bits to memory addresses and CPU registers at any given time. This would indeed interfere with the integrity of the data and could potentially inflict harm upon the patient if their doctor is led to believe that the patient is in good health when this is not the case.

In summary, our findings undoubtedly show that the hardware security of the Cardiomessenger 3G Smart is not ensured, and subsequently that the device is not sufficiently secured. Although we have not confirmed the scenarios described above, where potential attackers utilize our findings to breach confidentiality, integrity, or availability of patient data, there is reason to suspect that these properties of patient data are at risk. This is indeed an issue for the patients themselves. Consequently, it becomes increasingly important that patients are aware of the security risks involved within the pacemaker ecosystem and what those risks might entail for their data and treatment.

8.1.2 Implications for the vendor

Within our threat model in section 2.2, we also identified the vendor's backend servers as an asset. The vendor's backend servers constitute a vital part of the data flow from patients' pacemakers, via their HMUs, to practitioners and doctors. These servers stores private data from a large number of patients, and consequently, they should also be sufficiently secured. Arguably, the vendor's backend servers are not as critical an asset as patients' privacy and safety when considered in isolation. However, the vendor's backend servers and patient privacy are tightly linked, as a breach of those servers will most likely entail a breach of private patient information as well. Regarding the vendor's backend servers, the most significant threat actor is, again, organized crime.

One of our findings entailed obtaining cleartext credentials from debugging strings within the unencrypted firmware. The APN, IP address, port number and the username of the HMU are all obtainable credentials. For previous versions of HMUs, such credentials were used to authenticate with the APN and communicate with the backend server [WL19]. It is highly likely that similar authentication and communication is possible with the available credentials from the Cardiomessenger 3G Smart. An attacker that obtains physical access to a HMU can obtain these credentials without much effort. Once these are in the hands of an attacker, a range of attack vectors open up. Firstly, an attacker might abuse the credentials to

gain a foothold within their private network. From there, the attacker can launch reconnaissance attacks, mapping their servers and infrastructure. This could lead to a massive breach, compromising confidentiality, integrity, and availability of the servers. In turn, this could lead to a breach of the confidentiality, integrity, and availability for all patients associated with the pacemaker ecosystem of the vendor. Naturally, this would also impact the vendor's reputation.

Although we have not confirmed the scenario described above, our findings put the confidentiality, integrity, and availability of the vendor's backend servers slightly at risk. Consequently, since this also puts the confidentiality, integrity, and availability of private patient data at risk, the vendor's backend servers are an asset within the pacemaker ecosystem worth sufficiently securing. Specifically, security concerning how the backend servers can be accessed is of utmost importance.

As stated previously in section 1.2.2, medical devices are regulated by public authorities. The vendor is required by law to comply with the latest regulations. In this case, it falls under EU and Norwegian jurisdiction. The new EU regulations explicitly cover cybersecurity and will be in full effect as of May 2020. The regulation states the following regarding hardware security: *Manufacturers shall set out minimum requirements concerning hardware, IT networks characteristics and IT security measures, including protection against unauthorised access, necessary to run the software as intended* [Eur17b]. Considering the findings from this thesis, the question arises whether the vendor fails to comply with the newest regulation. Since the legal aspect is outside the scope of this thesis and that our findings might have been mitigated in newer firmware updates, we will not speculate whether the vendor complies with regulations or not. However, the vendor must ensure that in future firmware releases, they do comply with these regulations to not face legal ramifications for not securing their assets adequately.

8.2 Mitigation of findings from hardware security testing

All the produced findings from our research towards RO_1, and the vulnerabilities they give rise to, are possible to mitigate. Proper mitigation mechanisms could potentially reduce the risks that arise due to the HMU not being sufficiently secured, if not remove the risks entirely. In this section, we provide suggestions on what such mitigation mechanisms could entail, and how to implement them, to enhance the hardware security of the HMU. Our suggestions are largely based on the best security practices outlined by Joe Grand [Gra04].

8.2.1 Tamper-resistant hardware

Developers of embedded systems cannot neglect the fact that attackers can get their hands on these devices and compromise them at the physical level. Thus, for the next generation of HMU, we advocate the use of tamper-resistant hardware to harden the process of opening the device. In the best case, tamper-resistant hardware could entail that opening the device also meant inflicting damage to the PCB. However, we are unsure as to whether developing a HMU with such hardware is feasible. If not, we suggest that the plastic cover of the HMU is at least fastened using special screws, which can only be unscrewed using a proprietary tool.

8.2.2 PCB obfuscation

The lack of obfuscation mechanism on the PCB is apparent. Such obfuscation mechanisms can potentially function as a hindrance for attackers who are not very advanced and will thereby assist in securing the device. The JTAG and UART debugging interfaces should have their labels removed, and the serial numbers of hardware components on the PCB should also be removed after production. This would significantly harden the process of setting up a working JTAG connection and identifying the hardware components.

8.2.3 Disabling JTAG

The JTAG debugging interface should be disabled after production. This would entail adding some kind of protection on the TAP pins of JTAG, on the PCB itself. There are several ways to add protection to the TAP pins, ranging from quite straightforward to quite complex. One straightforward example of how to realize this is via adding fuses to the pins after production, leading to that nothing can be read of transmitted from those pins. A more complex example is to make use of a specific security feature in the STM32F4 microcontroller, namely Readout Protection (RDP). RDP is classified as a static protection mechanism, as it is enabled using option bytes. In the STM32F4, this mechanism offers protection against reading the main flash and SRAM memory through the JTAG interface [STM20]. However, RDP might not be enough to fully secure the debugging interface, as researches have recently proven that it offers no real protection[OT17]. Anyhow, enabling RDP will impede attempts from less advanced threats. Regardless of the complexity of the mechanism that is employed, we strongly recommend disabling the JTAG debugging interface using state of the art techniques.

8.2.4 Removing debugging strings and encrypting firmware

The use of logging functionalities within the firmware dramatically eases the process of reverse engineering for potential attackers, which naturally leads to that such

debugging strings should not be used. Despite this, it would still be possible to reverse engineer the firmware as it is unencrypted. To fully secure the HMU, we suggest that the firmware of the device is encrypted.

8.3 Discussion on findings from fuzzing

The final research we carried out was within RO_2, where we developed a framework capable of fuzzing the HMU's SMS interface. Fuzzing as a security testing technique is outlined in section 2.8, and by incorporating our black-box fuzzing methodology, which is further outlined in section 4.2.2, we were successful in the development of a working framework.

8.3.1 SMS applicability

Ideally, a security test should target all the networking interfaces, but realistically that scope turned out to be too wide for this thesis. Hence, for this thesis we chose to target the SMS interface, rather than the TCP/IP data-communication interface. The rationale for making such a choice is grounded in previous research and its applicability. The SMS interface can be attacked remotely and does not require physical access to the target HMU. Hence, the SMS interface is critical from a risk and impact perspective. Besides, SMS offers no real authentication mechanisms. In related work, researchers have developed SMS-fuzzers to disclose vulnerabilities with large success [Hon11][MGS11][MM09b][Gol11], showing its feasibility as a tool in security analysis. Furthermore, we knew that the HMUs could communicate with SMSs from previous thesis work [NB19][WL19], opening the SMS interface as a possible attack vector.

8.3.2 Strengths of fuzzing framework

Although our developed fuzzing framework is not exempt from limitations, the framework possesses three essential attributes that strengthen our contribution.

Firstly, the fuzzer framework generalizes well. The framework is not device-dependent, meaning that it can be used to fuzz all kinds of embedded systems, IoT devices or mobile phones, given that they have an SMS interface. Regardless of what kind of device, our framework is capable of fuzzing it, as long as it speaks the SMS protocol. In turn, this can potentially uncover vulnerabilities in any devices' SMS interface. Also, the framework is vendor-independent, meaning that it can be used to fuzz devices from any vendor. What this means is that the SMS-protocol is highly standardized, and all devices wishing to communicate via SMS must use the standardized protocol, not an altered proprietary version of it. It does not matter which manufacturer developed the product or if it is closed-source or not. The device can be fuzzed utilizing our software and the monitoring method relies on standardized responses from the network's perspective.

Secondly, the fuzzer framework offers great usability. Future researchers do not have to spend much time familiarizing themselves with our framework, as it can be

set up in a matter of hours. A USRP, installing and configuring the modified version of OpenBTS, and downloading and running the remaining code, is all it takes. This yields approximately 100,000 SMSs, within eight different test cases - which can be easily extended. Thus, our fuzzing framework provides future researchers with everything necessary for thorough testing of SMS interfaces, in a highly convenient manner. Since the fuzzer framework is open source, it is also easy for contributors to add additional functionality, as the foundation of the work is already implemented.

Lastly, and perhaps the most significant, the third strength of our contribution is related to the vulnerabilities our fuzzer framework uncovered. The fact that the framework uncovered several SMSs that triggered bugs in the Telit HE910 modem-series is indeed a strength. The vulnerabilities are not limited to the HMU itself, but rather to all devices that make use of the same kinds of modems, with the same firmware version. Thus, the vulnerabilities uncovered by our fuzzing of the HMU generalizes well, and are present on a variety of other devices possibly in use today.

8.3.3 Input from hardware testing to fuzzer development

As stated in section 1.3, the fact that we developed a fuzzing framework targeting the HMU's SMS interface, after we conducted a hardware security analysis of the HMU itself, is not a coincidence. The intention was that the hardware testing was to return valuable insight into how the HMU communicates with the backend servers of the vendor. This was indeed also the case, as the hardware testing yielded information regarding the HMU's modem. The manufacturer of the modem, as well as its model number, was consequently used to obtain further information. This information includes the datasheet and the AT-command reference guide of the modem. These documents were helpful throughout our work and turned out to be crucial when we realized that the fuzzer framework could not be utilized effectively on the HMU directly. In addition, the hardware security testing revealed JTAG and UART debug ports, which were both potential candidates for establishing monitoring capabilities at the receiver's side. With more effort - and a bit of luck, perhaps - it could have provided us with a more in-depth analysis of the crashes. Naturally, we were hoping that we could profit even more from the first scope when developing our fuzzing framework. However, we would not have been as successful in our second scope without completing the hardware security test.

Given that we quickly determined that the HMU would not stay connected to our fake base station for longer than approximately one minute, it became infeasible to utilize our fuzzing framework directly on the HMU. Consequently, we ordered a highly similar modem as the HMU uses, which brought several advantages. Having control over both sides of the communication channel, we could maintain a persistent, stable, and long-lasting connection to our network from the modem. Thereby, we

were able to run all test cases within our fuzzing framework, resulting in a thorough test of the security of the modem's SMS interface. As stated in section 7.1.1, there is no practical difference between testing the modem in isolation and targeting the HMU directly. Since the HMU utilizes the cellular baseband implemented on the modem chip when communicating over a cellular network, the same code base will be targeted. Therefore, once our fuzzing framework produced findings on the modem that was ordered, it was very plausible that the same finding could be observed on the HMU.

8.3.4 Answering research questions

Our main finding from fuzzing the HMU's SMS interface are several PDUs that denies the HMU's communication services. After sending this PDU to the HMU, it does not respond to subsequent messages and disconnects from our network. It requires that the microcontroller issues a reboot of the modem for it to return to normal operation. The modem of the HMU had seemingly crashed, which reflects the behavior we observed when we fuzzed the purchased modem directly. An interesting note is that when we sent the PDU to the HMU, nothing could be observed on its screen, leading to that the patient is left unaware of its network disconnection.

The fact that there exists a SMS PDU that can crash the modem of the HMU demonstrates a lack of appropriate security measures. The modem's firmware does not include proper edge case handling, and consequently crashed when it receives an input it is not capable of handling. This finding does directly correspond to a vulnerability that malicious adversaries can exploit. Therefore, relating to RQ_2, we can confidently state that there are indeed vulnerabilities in the modem's firmware that parses the incoming SMSs.

In this thesis, it has been clearly shown that it is indeed feasible to use the SMS interface as an attack vector since an attack on the SMS interface can leave the HMU incapable of fulfilling its function - effectively crashing the modem. The capabilities and resources of an adversary for conducting such an attack do not have to be at a government-funded level or organized crime. On the contrary, this attack can be launched using COTS-equipment and does not require highly competent adversaries. An adversary does not even have to possess in-depth knowledge of reverse engineering - proving the strength of fuzzing as a testing method. To answer RQ_3: it is indeed feasible to use the SMS interface as an attack vector. Moreover, we can assume that adversaries with limited resources can take advantage of this attack vector.

After sending any of the SMSs that causes the HMU to crash, it reconnects again after 15-20 minutes. We believe that this is related to that the HMU is trying to limit its battery usage by keeping the modem shut down for the majority of the time. Arguably, this is not an intended security mechanism in the firmware of the HMU

to protect against such vulnerabilities in the cellular modem. On the contrary, we believe that this was a design choice to reduce energy consumption. Nonetheless, it effectively resets the state of the modem and, thus, protects against the exploitation of bugs that renders the modem useless - to a certain degree. This mitigation strategy would offer no real security, however, since an attacker would still be able to maintain a crashed state by sending such offending SMSs every time it reconnected.

Based on that, we can partly validate and provide an answer to RQ_4. For the uncovered vulnerabilities within our research, which leads to a DoS-attack, there are no real security mechanisms in the firmware of the HMU that protects against any practical exploitation of this. Whether the same is true for more advanced attacks remains unknown.

8.3.5 Implications for patients

Similar to the hardware security testing of the HMU, the findings obtained from fuzzing the HMU's SMS interface also has implications for patients. Compared to previous findings, these findings can potentially be more severe, as they put patients' safety slightly more at risk. Patient safety is arguably the most important asset identified within our threat model in section 2.2. Also, within our threat model, we identified organized crime as the most significant threat actor.

A difference between the findings obtained from the hardware testing lies in how potential attackers would go about exploiting the vulnerabilities. The findings uncovered by fuzzing do not require having physical access to a HMU, but can be launched remotely. This does, in turn, lead to different implications for patients' safety. A remote attack is more likely to remain undetected since attackers do not need to acquire physical access. Additionally, a remote attack is generally easier to set up. Given the low complexity of the following hypothetical attack, it is reasonable to imagine that not only organized criminals can launch such an attack. We can assume that the other identified threat actors from within our threat model in section 2.2 might possess the appropriate funding and skill set necessary to complete the attack successfully. However, the success of the following hypothetical attack relies on a banal assumption. We must assume that the backend server has no monitoring method to detect that a HMU does not send its data.

If a potential attacker were to set up a fake base station close to a patient's home and subsequently utilize a jammer to make sure that the attacker's network offers the strongest signal in the area, the patient's HMU would connect to the attacker's network. Then, the attacker can send the offending SMS, which crashes the modem of the patient's HMU, causing it to disconnect from the attacker's network. The HMU will reconnect after 15-20 minutes after the modem is rebooted. However, an attacker can make sure to send the same offending SMS instantly - denying it

transmission and reception of messages. This effectively constitutes a DoS-attack, in which the HMU does not fulfill its function. To the best of our knowledge, the attack is unlikely to be detected by the patient. Since the HMU gives no indication of this, the patient would still believe that their data is relayed from the HMU to their doctor, when it is, in fact, not. If the patient's heart condition were to worsen after a successful attack, the patient's doctor would never receive data indicating a worsening in health condition. This hypothetical attack might prove critical for the patient's health, as their doctor remains unaware of their worsened condition, and the patient takes no medical countermeasures. Patient safety is consequently at risk given the successful completion of the DoS-attack, which is made possible due to the offending PDUs we uncovered.

Although we believe that the patient will remain unaware of the hypothetical attack described above, it is very likely that the attack is detected by the backend server. If the HMU does not send any messages over a certain time interval, the server could give an indication of a possibly malfunctioning HMU to the patient's doctor. Consequently, the doctor could contact the patient and the attack on patient safety fails. Also, a noteworthy point, since the HMU is not designed to alleviate sudden worsening of health condition for pacemaker patients, a compromise of the device will likely not lead to loss of human life.

8.3.6 Limitations of fuzzing framework

Although we were successful in developing a working fuzzing framework, targeting an SMS interface, our framework is not exempt from certain limitations. Firstly, a limitation is related to coverage of the test space within each test case. Since we decided to create fuzzed SMSs within eight different test cases, and given time restrictions, we had to limit the number of SMSs within each test case. Consequently, our framework does not cover all possible combinations within our test cases. For example, when fuzzing PID and DCS, which can take 256 different values each, the total number of SMSs we could craft with different combinations of PID and DCS becomes 65536. As this number is impractically large - considering the transmission rate of SMS from our network, we chose to limit the number of SMSs to approximately 14000, which amounts to roughly 20%. We made the same choice for all eight test cases, and for some of them, the total number of combinations of fuzzed fields is indefeasibly large nonetheless. In order to further develop our framework, we realize that the number of test cases could better cover the total test space within each test case. However, we made this trade-off since we wanted to fuzz a variety of different features of the SMS technology, thus, covering the entire test space for each test case was deemed infeasible.

Another limitation is not related to our development specifically, but rather to

fuzzing the SMS interface of a HMU itself. Given that the SMS interface exists in the modem, there is no guarantee that the sent SMS will ever reach the microcontroller of the HMU. We do not currently know how the firmware of the HMU manages incoming SMSs, that is, how the firmware extracts the payload from stored messages, and how often. Thereby, it becomes challenging to address the probability that a fuzzed SMS is capable of inflicting damage in the firmware of the HMU.

A third limitation is related to the fact that we chose to implement network side monitoring. Although network side monitoring has its benefits, it is far from perfect. Ideally, we would have preferred to implement some kind of receiver side monitoring mechanism, as it would yield a more detailed log of what happens inside the modem of the HMU when we trigger a bug. Since we did not implement such monitoring, we are unable to state why the bug was triggered when it receives our offending PDUs. Network side monitoring is indeed a limitation, as it leaves us unable to transform bugs to exploits by crafting specific SMS PDUs. With receiver side monitoring, we could potentially add such malicious content. For example, we could be able to craft SMSs that leverage buffer overflow bugs within the modem to gain code execution or read sections of memory. Exploiting such buffer overflows on mobile phones has been done in similar work conducted by Weinmann [Wei12], which completely compromised the integrity of the targets. We chose network side monitoring due to its low complexity. However, we realize that receiver side monitoring would be the most beneficial option.

8.4 Mitigation of findings from fuzzing

As mentioned earlier, due to the limitation of not having receiver side monitoring, we have not obtained logs which explicitly state why the modem crashes when it receives the offending PDU we uncovered. Consequently, it becomes challenging to be specific when proposing potential techniques to mitigate the vulnerability that exist in the modem.

Nonetheless, there is no doubt that the vulnerabilities should be mitigated somehow. Firstly, the bugs found in our experiments that caused the HMU's modem to crash should be identified in the firmware and patched with proper edge-case handling. An extensive review of the modem's firmware should then be carried out, as it might uncover pieces of code that contain vulnerabilities - as it is unlikely that the firmware only contains the bugs that were disclosed by us.

Also, as mentioned in section 8.3.5, we deem it likely that an attack exploiting the uncovered vulnerabilities in the modem will be detected by the backend server. If this is not the case as of now, a possible mitigation mechanism is to implement such functionality, which leads to a rapid detection of a compromised HMU.

8.5 Future work

Although we have performed extensive research on both the hardware security and the SMS interface of the Cardiomessenger 3G Smart HMU, several aspects of the device remain uninvestigated. These aspects might also be vulnerable to cyber-attacks. Therefore, we propose the following as future work within the field of pacemaker HMU security.

Enhancements on our SMS fuzzing framework is likely to produce even more findings, and possibly yield more knowledge about the security level of the SMS interface within the HMU. Enhancements can include new test cases, more granular monitoring, and smarter input generation. New test cases can uncover whether the SMS interface of the HMU is vulnerable to other features of the SMS technology. Improved monitoring, such as receiver side monitoring, would provide more knowledge regarding the security of the HMU's communication capabilities and could be used as a technique to develop exploits from the identified bugs. Exploiting bugs in a more advanced fashion could potentially demonstrate attacks on the pacemaker ecosystem with devastating consequences. Attackers can also use reverse engineering as a method to construct malicious input. By reverse engineering the HMU's firmware to understand better how different input can inflict damage not only in the modem but on the application layer, can be used to form smarter input.

Fuzzing of the data-communication capabilities of the HMU proved to be too extensive for this thesis project. Thus, future work can research the TCP/IP interface of the HMU, which might uncover other vulnerabilities. The architecture of such an experiment would be very similar to that of the SMS fuzzing. Future work can thus benefit from our simulated network and fuzzing generation. Hence, part of the foundation for this work is already laid by our work.

A validation of the untested hypotheses on the Cardiomessenger 3G Smart would provide the necessary information to affirm the complete evolution of security features from the previous versions of HMUs. This work could be carried out by setting up an illegitimate base station and spoof the vendor's backend server, which is possible since we retrieved the IP address and port number from the memory dumps.

USB fuzzing could also be a promising research area. The HMU has a USB-micro port, which it charges through. Attempting to fuzz that port could also uncover vulnerabilities, and interfere with the HMU's normal operation.

8.6 Ethical considerations

Finally, we need to address some ethical boundaries concerning our findings. As stated previously, we want to disclose our findings to the affected vendor responsibly.

Throughout this thesis project, there has been an ongoing coordinated vulnerability disclosure process with Biotronik as a consequence of the findings from last year's thesis projects [WL19] [NB19]. Because of this, SINTEF and NTNU have additional knowledge related to the vendor's systems and devices that we would not have without several meetings. Due to this new knowledge, we know that the vendor claims to have taken the findings and hypothetical attacks described in this thesis into account when developing their newest version of the firmware. Therefore, there is no reason for us to initiate a new coordinated vulnerability disclosure process with the vendor. The fact that these findings have been accounted for could imply that the vendor was not comfortable with the exposed attack vectors described in section 8.1. This substantiates our discussion on the hardware security testing of the device and the outlined implications for both vendor and patients.

For the modem manufacturer, we are going to initiate a coordinated vulnerability disclosure process such that they are aware of the vulnerabilities uncovered in this thesis. Hopefully, they will recognize that these findings constitute potential attack vectors against IoT-devices that have their modem integrated, with the same firmware version, and fix them accordingly.

Chapter 9
Conclusion

This master thesis defined two scopes regarding the security of the most recent HMU from Biotronik, namely the Cardiomessenger 3G Smart. A hypothesis, research questions and research objectives were formulated within each scope, and this chapter will address whether we achieved our research objectives and were successful in validating each of those hypotheses.

Within scope 1, we incorporated a black-box hardware testing methodology and thoroughly tested the hardware security of the HMU. We were indeed successful in achieving RO_1, as we successfully tested whether a set of security related hypotheses that had been proven true for older HMUs, also were true for the most recent version. We obtained a set of findings, some of which directly corresponded to vulnerabilities, while others were stepping stones that would lay the foundation for other attacks and vulnerabilities. Mitigation techniques were then proposed to counter the vulnerabilities. This leads to the confirmation of our main hypothesis within scope 1 of this thesis project, which was:

H_1 *The Cardiomessenger 3G Smart HMU contains vulnerabilities as a standalone embedded medical device that can be exploited by an attacker having physical access to the device to compromise a patient's privacy or safety.*

Within scope 2, we incorporated a black-box fuzzing methodology, and successfully developed a framework capable of fuzzing the SMS interface of the HMU. The framework is universally applicable and can target the SMS interface of any device. We achieved RO_2, as we thoroughly investigated potential vulnerabilities in the HMU's SMS interface through fuzzing. The fuzzing successfully uncovered several SMSs that crashes the HMU's modem, effectively resulting in a DoS-attack, which puts patient safety at risk. Then, countermeasures were identified for this vulnerability. The offending SMSs we uncovered are not unique to the HMU, but affects all IoT-devices with the same modem. To disclose our findings in a responsible manner to

the affected vendor, we chose to follow a coordinated vulnerability disclosure process. This leads to the confirmation of our hypothesis within scope 2 of this thesis project, which was:

H_2 *The implementation of SMS within the HMU is insecure and contains vulnerabilities that can be exploited by an attacker determined to compromise either the safety or the privacy of a patient.*

References

[3rd99] 3rd Generation Partnership Project. *Alphabets and language-specific information (GSM 03.38)*, June 1999.

[3rd00] 3rd Generation Partnership Project. *Point-to-Point (PP) Short Message Service (SMS) support on mobile radio interface (GSM 04.11)*, October 2000.

[3rd02] 3rd Generation Partnership Project. *Technical realization of the Short Message Service (SMS)Point-to-Point (PP) (GSM 03.40)*, January 2002.

[3rd08] 3rd Generation Partnership Project. *Mobile Station - Base Station System (MS - BSS) interface; Data Link (DL) layer specification (GSM 04.06)*, December 2008.

[3rd19] 3rd Generation Partnership Project. *3GPP TS 23.040; Technical realization of the Short Message Service (SMS)*, March 2019.

[3rd20] 3rd Generation Partnership Project. *3GPP TS 31.115; Secured packet structure for (Universal) Subscriber Identity Module (U)SIM Toolkit*, March 2020.

[A+17] US Food and Drug Administration et al. Cybersecurity vulnerabilities identified in St. Jude Medical's implantable cardiac devices and Merlin@ home transmitter: FDA safety communication. `https://www.fda.gov/medical-devices/safety-communications/cybersecurity-vulnerabilities-identified-st-jude-medicals-implantable-cardiac-devices-and-merlinhome`, January 2017.

[Blo16] Carson C. Block. Muddy waters capital is short st. jude medical, inc. (stj us). Research report, Muddy Waters Capital LLC, August 2016.

[DD12] Sebastien Dudek and Guillaume Delugré. Fuzzing the gsm protocol stack, 2012. `http://archive.hack.lu/2012/Fuzzing_The_GSM_Protocol_Stack_-_Sebastien_Dudek_Guillaume_Delugre.pdf`.

[Doe19a] Christian Doerr. *Network Security in Theory and Practice*, chapter Cryptographic Building Blocks, page 214. Christian Doerr, 2019.

[Doe19b] Christian Doerr. *Network Security in Theory and Practice*, chapter Introduction, page 38. Christian Doerr, 2019.

[Eur17a] European Commission - Press Release. New eu rules on medical devices to enhance patient safety and modernise public health. https://ec.europa.eu/commission/presscorner/detail/en/IP_17_847, April 2017.

[Eur17b] European Parliament, Council of the European Union. Regulation (eu) 2017/745 of the european parliament and of the council of 5 april 2017 on medical devices, amending directive 2001/83/ec, regulation (ec) no 178/2002 and regulation (ec) no 1223/2009 and repealing council directives 90/385/eec and 93/42/eec, May 2017.

[GM14] Brian Gorenc and Matt Molinyawe. Blowing up the celly! building your own sms/mms fuzzer. https://www.defcon.org/images/defcon-22/dc-22-presentations/Gorenc-Molinyawe/DEFCON-22-Brian-Gorenc-Matt-Molinyawe-Blowing-Up-The-Celly.pdf, 2014.

[Gol11] Nico Golde. Sms vulnerability analysis on feature phones. Master's thesis, Technische Universität Berlin, Germany, 2011.

[Gra04] Joe Grand. Practical secure hardware design for embedded systems. In *Proceedings of the 2004 Embedded Systems Conference, San Francisco, California*, 2004.

[HHR+08] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 129–142, 2008.

[Hon11] Brinio Hond. Fuzzing the gsm protocol. Master's thesis, Radbound University Nijmegen, The Netherlands, 2011.

[IEE13] IEEE Computer Society. Ieee standard for test access port and boundary-scan architecture. *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pages 1–444, 2013.

[ISO18] ISO Central Secretary. Information technology - security techniques - vulnerability disclosure. Standard ISO/IEC 29147:2018, International Organization for Standardization, Geneva, CH, 2018.

[KM19] Jakob Stenersen Kok and Bendik Aalmen Markussen. A security analysis of the pacemaker ecosystem. Project report in ttm4502, Department of Information Security and Communication Technology, NTNU – Norwegian University of Science and Technology, Nov 2019.

[KW18] Eivind Skjelmo Kristiansen and Anders Been Wilhelmsen. Security testing of the pacemaker ecosystem. Master's thesis, Norwegian University of Science and Technology, Norway, 2018.

[LC16] Robert Lipovsky and Anton Cherepanov. Blackenergy trojan strikes again: Attacks ukrainian electric power industry. https://www.welivesecurity.com/2016/01/04/blackenergy-trojan-strikes-again-attacks-ukrainian-electric-power-industry/, January 2016.

[Lov05] Lovdata.no. Forskrift om medisinsk utstyr, kap 1, paragraf 1-5. https://lovdata.no/forskrift/2005-12-15-1690/\T1\textsection1-5, December 2005.

[MGS11] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. Sms of death: From analyzing to attacking mobile phones on a large scale. In *USENIX Security Symposium*, volume 168, 2011.

[MM09a] Collin Mulliner and Charlie Miller. Fuzzing the phone in your phone. *Black Hat USA*, 25:31, 2009.

[MM09b] Collin Mulliner and Charlie Miller. Injecting sms messages into smart phones for security analysis. In *USENIX Workshop on Offensive Technologies (WOOT)*, volume 29, 2009.

[MSG+16] Eduard Marin, Dave Singelée, Flavio D Garcia, Tom Chothia, Rik Willems, and Bart Preneel. On the (in) security of the latest generation implantable cardiac defibrillators and how to secure them. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 226–236, 2016.

[MSP20] Dominik Maier, Lukas Seidel, and Shinjo Park. Basesafe: Baseband sanitized fuzzing through emulation, 05 2020.

[MT90] Colin M Maunder and Rodham E Tulloss. *The test access port and boundary scan architecture*. IEEE Computer Society Press Los Alamitos/Washington, DC, 1990.

[NB19] Guillaume Nicolas Bour. Security analysis of the pacemaker home monitoring unit: A blackbox approach. Master's thesis, Norwegian University of Science and Technology, Norway, 2019.

[OT17] Johannes Obermaier and Stefan Tatschner. Shedding too much light on a microcontroller's firmware protection. In *Proceedings of the 11th USENIX Conference on Offensive Technologies*, WOOT'17, page 17, USA, 2017. USENIX Association.

[OWAa] OWASP. Application threat modeling. https://owasp.org/www-community/Application_Threat_Modeling.

[OWAb] OWASP. Fuzzing. https://owasp.org/www-community/Fuzzing.

[Par18] The United Kingdom Parliament. Cyber-attack on the nhs. https://publications.parliament.uk/pa/cm201719/cmselec5/cmpubacc/787/78702.htm, March 2018.

[RB17] Billy Rios and Jonathan Butts. Security evaluation of the implantable cardiac device ecosystem architecture and implementation interdependencies. Technical report, Whitescope, May 2017.

[Reg15] Regjeringen. Digital sårbarhet – sikkert samfunn — Beskytte enkeltmennesker og samfunn i en digitalisert verden (Digital Vulnerabilities in Society). *Norsk Offentlig Utredning 2015: 13 (Official Norwegian Report 2015: 13)*, 2015.

[Sol16] Telit Wireless Solutions. He910/ue910/ul865/ue866 at commands reference guide. https://www.telit.com/wp-content/uploads/2017/09/Telit_3G_Modules_AT_Commands_Reference_Guide_r11.pdf, October 2016.

[SP18] Torkel Steen and Eivind S. Platou. Norsk pacemaker- og icd-statistikk for 2017. *Hjerteforum*, 31(2), 2018.

[STM20] STMicroelectronics. Introduction to stm32 microcontrollers security. https://www.st.com/resource/en/application_note/dm00493651-introduction-to-stm32-microcontrollers-security-stmicroelectronics.pdf, February 2020.

[The18] The Norewgian Medicines Agency. Our goals and tasks. https://legemiddelverket.no/english/about-us/our-goals-and-tasks, January 2018.

[The20] The Norwegian Medicines Agency. About medical devices - new eu regulations. https://legemiddelverket.no/english/medical-devices/about-medical-devices#new-eu-regulations, January 2020.

[UTHG17] Jon Ungoed-Thomas, Robin Henry, and Dipesh Gadher. Cyber-attack guides promoted on youtube. https://www.thetimes.co.uk/article/cyber-attack-guides-promoted-on-youtube-972s0hh2c, May 2017.

[Wei12] Ralf-Philipp Weinmann. Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks. In *WOOT*, pages 12–21, 2012.

[WL19] Anniken Wium Lie. Security analysis of wireless home monitoring units in the pacemaker ecosystem. Master's thesis, Norwegian University of Science and Technology, Norway, 2019.

Appendix A
OpenOCD Scripts

A.1 Simple connection using OpenOCD

```
1  # Script used to establish a simple connection to a CardioMessenger 3g Smart
2
3  # INTERFACE
4  interface bcm2835gpio
5  bcm2835gpio_peripheral_base 0x20000000
6  bcm2835gpio_speed_coeffs 113714 28
7  bcm2835gpio_jtag_nums 11 25 10 9
8  bcm2835gpio_srst_num 24
9  reset_config srst_only srst_push_pull
10 adapter_khz 500
11
12 # TRANSPORT
13 transport select jtag
14
15 # TARGET
16 set WORKAREASIZE 0
17 set CHIPNAME stm32f4x
18 source [find target/stm32f4x.cfg]
19 reset_config srst_only srst_nogate
20 adapter_nsrst_delay 100
21 adapter_nsrst_assert_width 100
22
23 # EXEC
24 init
25 targets
26 halt
```

Code Listing A.1: Successful JTAG connection using OpenOCD

A.2 Dump memory using OpenOCD

```
1  # Script used to dump the firmware and RAM of the Cardiomessenger 3G Smart
2
3  # INTERFACE
4  interface bcm2835gpio
5  bcm2835gpio_peripheral_base 0x20000000
6  bcm2835gpio_speed_coeffs 113714 28
7  bcm2835gpio_jtag_nums 11 25 10 9
8  bcm2835gpio_srst_num 24
9  reset_config srst_only srst_push_pull
10 adapter_khz 500
11
```

```
12 # TRANSPORT
13 transport select jtag
14
15 # TARGET
16 set WORKAREASIZE 0
17 set CHIPNAME stm32f4x
18 source [find target/stm32f4x.cfg]
19 reset_config srst_only srst_nogate
20 adapter_nsrst_delay 100
21 adapter_nsrst_assert_width 100
22
23 # EXEC
24 init
25 targets
26 halt
27
28
29 echo "Dumping flash..."
30 dump_image flash.img 0x08000000 1048575
31 echo "Done!"
32
33 echo "Dumping CCM RAM..."
34 dump_image ccm_ram.img 0x10000000 65535
35 echo "Done!"
36
37 echo "Dumping system memory OTP..."
38 dump_image system_memory_OTP.img 0x1FFF0000 31247
39 echo "Done!"
40
41 echo "Dumping SRAM..."
42 dump_image sram.img 0x20000000 131071
43 echo "Done!"
44
45 echo "Dumping RAM..."
46 dump_image ram.img 0x60000000 2097152
47 echo "Done!"
```

Code Listing A.2: Successful memory dumping using OpenOCD

Appendix B

Installing modified version of OpenBTS

Although OpenBTS is intended for use on a Ubuntu 12.04 distributions, we managed to install OpenBTS on a Ubuntu 16.04 distribution properly. This appendix entails a description on how to install OpenBTS version 5 with the extra feature of sending raw PDUs to a specified IMSI on a clean installation of Ubuntu 16.04. This guide is based on a mikrotechnia blog post[1].

First install necessary dependencies.
$sudo apt-get install software-properties-common python-software-properties
$sudo add-apt-repository ppa:git-core/ppa (press enter)

Then update and install git.
$sudo apt-get update
$sudo apt-get install git

Download OpenBTS, SMQueue, SIPauthserve, Asterisk and UHD drivers. First install convenient scripts from Range Networks.
$cd
$git clone https://github.com/RangeNetworks/dev.git
$cd dev
$./clone.sh

Before building the package, we need to copy our code containing the functionality of sending raw PDUs into the source code of OpenBTS. We implemented two varieties of this. The first function sends one SMS PDU to a specified IMSI, whereas the other function sends all PDUs from a file to a specified IMSI. This needs to be added as functions in the CLICommands.cpp file inside *dev/openbts/CLI* folder.

[1] `https://mikrotechnica.wordpress.com/2017/04/09/setting-up-openbts-on-national-instruments-usrp-2922/?fbclid=IwAR2lwn8dqgDqo1GKjrORheWk_z73_osYKm75-S93iIgILO43qXlMFwhpEUJU`, accessed on 20.03.2020.

```
/** Submit an SMS for delivery to an IMSI. */
static CLIStatus sendsmspdu(int argc, char** argv, ostream& os)
{
        if (argc<4) return BAD_NUM_ARGS;

        char *IMSI = argv[1];
        char *srcAddr = argv[2];
        string rest = "";
        for (int i=3; i<argc; i++) rest = rest + argv[i]; //+ " ";
        const char *txtBuf = rest.c_str();

        if (!isIMSI(IMSI)) {
                os << "Invalid IMSI. Enter 15 digits only.";
                return BAD_VALUE;
        }

        // We just use the IMSI, dont try to find a tmsi.
        FullMobileId msid(IMSI);
        Control::TranEntry *tran = Control::TranEntry::newMTSMS(
                                                NULL,    // No SIPDialog
                                                msid,
                                                GSM::L3CallingPartyBCDNumber(src
                                                        Addr),
                                                string("application/vnd.3gpp.
                                                sms"));
        Control::gMMLayer.mmAddMT(tran);
        os << "message submitted for delivery" << endl;
        return SUCCESS;
}


//send all SMSs from a file
static CLIStatus sendsmspdus_from_file(int argc, char** argv, ostream& os){
        if (argc<4) return BAD_NUM_ARGS;

        char *IMSI = argv[1];
        char *srcAddr = argv[2];
        string filename = "";
        for (int i=3; i<argc; i++) filename = filename + argv[i]; //+ " ";
        const char *txtBuf = filename.c_str();

        if (!isIMSI(IMSI)) {
                os << "Invalid IMSI. Enter 15 digits only.";
                return BAD_VALUE;
        }

        ifstream inFile;
        string path = "../../../Master-Project/src/3g/Fuzzer/TestCases/"+filename;
        inFile.open(path.c_str());
        string line;

        if(!inFile){
                os<<"Error, could not open file.";
                return BAD_VALUE;
        }
        while(getline(inFile,line)){
                // We just use the IMSI, dont try to find a tmsi.
                FullMobileId msid(IMSI);
                Control::TranEntry *tran = Control::TranEntry::newMTSMS(
                                                        NULL,    // No SIPDialog
                                                        msid,
                                                        GSM::L3CallingParty
                                                        BCDNumber(srcAddr),
                                                        line, // message body
                                                        string("application/
                                                        vnd.3gpp.sms"));
                Control::gMMLayer.mmAddMT(tran);
                os << "message submitted for delivery" << endl;
        }
        inFile.close();
```

```
71          return SUCCESS;
72  }
```

Code Listing B.1: Function containing PDU contruction.

Additionally, we need to add the new command in the command parser. This is done by adding the following line in the Parser function inside the same file.

```
1  addCommand("sendsmspdu", sendsmspdu, "IMSI src# PDU message... -- send PDU SMS
2                      to IMSI, addressed from source number src#.");
3
4  addCommand("sendsmspdus_from_file", sendsmspdus_from_file, "<IMSI> <src>
5          <filename> -- send PDUs in filename to specified IMSI from src");
```

Code Listing B.2: Adding command to parser.

Now in the *dev* folder run the build script with the wanted USRP. Since we used Ettus research N200 we provide this as argument to the build script.
$./build.sh N200

Then we need to install the correct Debian packages.
$cd BUILDS/<your_build_timestamp>
$sudo dpkg -i *.deb
$sudo apt-get install -f

A separate tutorial on how to install the USRP can be found in this post[2]. Since we are running Ubuntu 16.04 instead of 14.04, we must further perform some surgery on the build for it to work. It appears that this build cannot find the installed transceiver correctly, thus from inside *openbts/apps/* folder, run the following to create a symbolic link:
$sudo ln -s ../Transceiver52M/transceiver .

Next, Ubuntu changed its init daemon from Upstart to Systemd in the 16.04 version. Therefore the SMQueue, SIPauthserve and Asterisk processes need to be started directly.
$sudo /usr/local/sbin/smqueue
$sudo /usr/local/sbin/sipauthserve
$sudo /usr/sbin/asterisk -vvvv

Then start OpenBTS from *openbts/apps/* folder:
$sudo ./OpenBTS

For the HMU to connect to our network, certain configurations needed to be applied. The mobile country code (MCC) and operator name was changed to that of the

[2] `https://kb.ettus.com/USRP_N_Series_Quick_Start_(Daughterboard_Installation)`, accessed on 24.03.2020.

SIM-card operator on the HMU. Additionally, TMSIS were assigned once connected and GPRS was enabled. GSMTAP, which is used for network captures, was also enabled. Inside the OpenBTS, the following commands: **OpenBTS> config Control.LUR.SendTMSIs 1**
OpenBTS> config Control.GSMTAP.GPRS 1
OpenBTS> config Control.GSMTAP.GSM 1
OpenBTS> config GSM.Identity.ShortName Telekom
OpenBTS> config GSM.Identity.MCC 262

Appendix C
Fuzzer framework

C.1 Parent class for different fuzzing cases

```
import random
import smspdu
import Utils
import get_dcs_length_mapping
import get_fuzz_payload
import abc

class fuzzer:
    rpdu_base = "014803a1000000"
    test_cases = []
    user_data =""
    rpdu_counter = 0
    i=0

    def __init__(self):
        self.payloads = get_fuzz_payload.payloads
        self.dcs_length_mapping = get_dcs_length_mapping.dic

    def fuzz_pid(self, tpdu):
        value = random.randint(0, 256)
        tpdu.tp_pid = value

    @abc.abstractmethod
    def fuzz_dcs(self, tpdu):
        return

    def fuzz_ud(self,tpdu):
        if self.i < len(self.payloads):
            dcs = tpdu.tp_dcs
            max_payload_len = self.dcs_length_mapping[dcs]
            if len(self.payloads[self.i]) <= max_payload_len:
                tpdu.tp_ud = self.payloads[self.i]
                tpdu.tp_udl = len(self.payloads[self.i])
            self.i += 1
```

```
    @abc.abstractmethod
    def fuzz_tpdu(self,tpdu):
        return

    def create_fuzzed_tpdu(self):
        tpdu = smspdu.SMS_DELIVER.create('12445678','recipient',self.
                                                user_data)
        self.fuzz_tpdu(tpdu)
        tpdu_string = tpdu.toPDU()
        return tpdu_string

    def pack_fuzzed_tpdu_in_rpdu(self,tpdu):
        # Add length of the TPDU at the end of RPDU header
        # Divde by two because we group them in octets
        len_in_hex = Utils.to_hex(len(tpdu)/2)
        tmp = self.rpdu_base + len_in_hex

        # CreateMR-field of the RPDU, must be an octet
        if self.rpdu_counter==256:
            self.rpdu_counter=0
        rpdu_mr = str(Utils.to_hex(self.rpdu_counter))

        # Set MR-field of RPDU, increment the MR-field of RPDU by 1
        ret = tmp[0] + tmp[1] + rpdu_mr + tmp[4:]
        self.rpdu_counter += 1
        ret += tpdu

        #Return TPDU packed in RPDU with set MR-field
        return ret

    def create_fuzzed_pdus(self):
        # Create final fuzzed PDUs, append to test_cases
        for i in range(len(self.payloads)):
            tpdu = self.create_fuzzed_tpdu()
            rpdu_with_tpdu = self.pack_fuzzed_tpdu_in_rpdu(tpdu)
            self.test_cases.append(rpdu_with_tpdu)

    def write_to_files(self, subfolder, filename_base):
        # Write final fuzzed PDUs to file, 2 PDUs per file
        file_counter = 1
        start = 0
        step = 1000
        total_number_of_pdus = len(self.test_cases)
        while start < total_number_of_pdus:
            current = self.test_cases[start:step]
            with open('TestCases/' + subfolder + filename_base + str(
                                                file_counter) + '.txt',
                                                 'a') as file:
                for i in range(len(current)):
                    out = current[i]
                    file.write(out)
                    file.write("\n")
```

```
            start += 1000
            step += 1000
            file_counter += 1
```

C.2 Implementing the actual sending of PDUs via OpenBTS

```
import subprocess
import time
from os import walk

test_cases = []
files = []
path = "TestCases/"

def load_files(testcase):
    #needed to make this slighty weird because walk method does not
                                        give test cases in sorted order
    for (dirpath,dirname,filenames) in walk(path+testcase):
        for i in range(len(filenames)):
            try:
                filename = filenames[i]
                numb = filename[len(testcase):].split(".")[0]
                files.append((testcase + "/" + filenames[i],int(numb)))
            except Exception as e:
                print(e)

#sorts the tuples such that RP_MR are sent in increasing order
def sort_files():
    sorted_files = sorted(files, key=lambda filename: filename[1])
    return sorted_files

def send_sms(imsi,src):
    cmd_path = "/home/jakob/Master/dev/openbts/apps/OpenBTSCLI"
    filenames = sort_files()
    for file in filenames:
        cmd = "sendsmspdus_from_file" + " " + imsi + " " + src + " " +
                                                file[0]
        a = subprocess.Popen([cmd_path, "-c", cmd], stdout=subprocess.
                                                PIPE)
        (std_out, std_err) = a.communicate()
        print(std_out)
        time.sleep(1)

def main():
    load_files("NAME of TESTCASE")
    send_sms("IMSI","SOURCE")

main()
```

C.3 Monitoring

C.3.1 Health checker

```
import socket
import threading
import serial

#set serial communication variables
ser = serial.Serial()
ser.baudrate = 115200
ser.port = 'COM9'

#configure socket
HOST = '192.168.1.118'
PORT = 65432        # Port to listen on (non-privileged ports are >
                                    1023)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(5)
print("Binded to %s IP on port %d" %(HOST,PORT))

def communicate_with_modem():
    ret = ""
    cmd = "AT+CMGD=1,4\r\n"
    try:
        ser.open()
    except Exception:
        print("Can't establish connection to modem..")
        print("Stop sending")
        ret = b"stop"
    if ser.isOpen():
        ser.write(cmd.encode())
        reply = ser.read(17)  # Number of bytes, varies depending on
                                        command
        print("Reply from modem was: %s" %reply)
        if "OK" in reply.decode("utf-8"):
            ret = b"continue"
        else:
            ret = b"stop"
    ser.close()
    return ret

def on_new_client(clientsocket):
    while True:
        msg = clientsocket.recv(1024)
        if not msg:
            break
        response = communicate_with_modem()
        clientsocket.sendall(response)
    clientsocket.close()
```

```
def main():
    while True:
        conn, addr = s.accept()
        print("Connected to: " + addr[0] + ":" + str(addr[1]))
        t = threading.Thread(target=on_new_client,args=(conn,))
        t.start()
    s.close()
main()
```

C.3.2 Parsing resulting Pcap file

```
import pyshark

def compare_reference_number(prev_pkt, pkt):
    equal = False
    if prev_pkt["gsm_a.rp"].rp_message_reference == pkt["gsm_a.rp"].
                                        rp_message_reference:
        equal = True
    return equal

def write_pdu_to_file(pkt,testcase):
    filename = "../ResendPDUs/resend_" + testcase + ".txt"
    with open(filename, 'a') as file:
        rpdu = pkt["gsm_a.dtap"].rpdu
        # strip rpdu of : separators
        line = ""
        for character in rpdu:
            if character != ":":
                line += character
        file.write(line + "\n")


def main():
    testcase = "fuzz_pid_7bit_dcs_ud"
    target = "../Pcaps/fuzz_pid_7bit_dcs_ud/" + testcase + ".pcap"
    cap = pyshark.FileCapture(target)
    sent = 0
    responses = 0
    previous_sms = None  # Initialized to NONE, first SMS must arrive
                                        before first response
    previous_sms_got_response = False
    last_packet_was_sms = False
    for pkt in cap:
        # This means that there is an RPDU layer in this packet
        if "gsm_a.rp" in pkt:
            # Network to MS RP-DATA
            if pkt["gsm_a.rp"].msg_type == "0x00000001":
                if last_packet_was_sms and sent != 0:
                    print("Last SMS with msg-ref %s never got reply"
                    % previous_sms["gsm_a.rp"].rp_message_reference)
                    write_pdu_to_file(previous_sms,testcase)
                previous_sms = pkt
```

```
                previous_sms_got_response = False
                sent += 1
                last_packet_was_sms = True
                print("Parsed this many SMS: " + str(sent))
            else:
                last_packet_was_sms = False
                # MS to Network RP-ACK
                if pkt["gsm_a.rp"].msg_type == "0x00000002":
                    responses += 1
                    # Check if RP-ACK was for previous SMS
                    if compare_reference_number(previous_sms, pkt):
                        previous_sms_got_response = True
                        print("SMS with msg-ref %s received an ACK"
                        % previous_sms["gsm_a.rp"].rp_message_reference
                                                                )
                # MS to Network RP-ERROR
                if pkt["gsm_a.rp"].msg_type == "0x00000004":
                    responses += 1
                    # Check if RP-ERROR was for previous SMS
                    if compare_reference_number(previous_sms, pkt):
                        previous_sms_got_response = True
                        print("SMS with msg-ref %s received an ERROR"
                        % previous_sms["gsm_a.rp"].rp_message_reference
                                                                )
                # If RP-ACK or RP-ERROR does not belong to previous SMS
                                                    , write PDU to file
                if not previous_sms_got_response:
                    print("SMS with msg-ref %s did not receive response
                                                                    "
                    % previous_sms["gsm_a.rp"].rp_message_reference)
                    write_pdu_to_file(previous_sms,testcase)

    print("Sent %d" % sent)
    print("Responses %d" % responses)


main()
```

C.4 Various fuzzing of PID, DCS and UD

C.4.1 Fuzzing PID, DCS and UD

```
from create_sms_cases import fuzzer
import random

class testcase1_fuzz_pid_dcs_ud(fuzzer):

    def fuzz_dcs(self, tpdu):
        value = random.randint(0, 256)
        tpdu.tp_dcs = value
```

```
    def fuzz_tpdu(self, tpdu):
        self.fuzz_pid(tpdu)
        self.fuzz_dcs(tpdu)
        self.fuzz_ud(tpdu)

def main():
    fuzz = testcase1_fuzz_pid_dcs_ud()
    fuzz.create_fuzzed_pdus()
    fuzz.write_to_files('fuzz_pid_dcs_ud/', 'fuzz_pid_dcs_ud')

main()
```

C.4.2 Fuzzing PID and UD, with 7 bit DCS

```
from create_sms_cases import fuzzer

class testcase2_fuzz_pid__7bit_dcs_ud(fuzzer):

    def fuzz_dcs(self,tpdu):
        value = 0
        tpdu.tp_dcs=value

    def fuzz_tpdu(self, tpdu):
        self.fuzz_pid(tpdu)
        self.fuzz_dcs(tpdu)
        self.fuzz_ud(tpdu)

def main():
    fuzz = testcase2_fuzz_pid__7bit_dcs_ud()
    fuzz.create_fuzzed_pdus()
    fuzz.write_to_files('fuzz_pid_7bit_dcs_ud/', 'fuzz_pid_7bit_dcs_ud'
                                            )

main()
```

C.4.3 Fuzzing PID and UD, with common DCS

```
from create_sms_cases import fuzzer
import random

class fuzz_pid_common_dcs_ud(fuzzer):

    i = 0
    # 7 bit, 8 bit, 7 bit class 1, 8 bit class 1
    common_dcs = [0,4,241,245]

    def fuzz_dcs(self, tpdu):
        value = self.common_dcs[random.randint(0,len(self.common_dcs)-1
                                                )]
        tpdu.tp_dcs = value
```

```
    def fuzz_tpdu(self, tpdu):
        self.fuzz_pid(tpdu)
        self.fuzz_dcs(tpdu)
        self.fuzz_ud(tpdu)

def main():
    fuzz = fuzz_pid_common_dcs_ud()
    fuzz.create_fuzzed_pdus()
    fuzz.write_to_files('fuzz_pid_common_dcs_ud/', '
                                        fuzz_pid_common_dcs_ud')

main()
```

C.4.4 Flash SMS

```
from create_sms_cases import fuzzer
import random


class fuzz_pid_ud(fuzzer):

    dcs_flash_values=[240,244]

    def fuzz_dcs_pid(self,tpdu):
        a = random.randint(0,1)
        if a == 0:
            tpdu.tp_pid=64
            tpdu.tp_dcs=random.randint(0,255)
        else:
            tpdu.tp_pid=random.randint(0,255)
            b = random.randint(0,1)
            tpdu.tp_dcs=self.dcs_flash_values[b]

    def fuzz_tpdu(self, tpdu):
        self.fuzz_dcs_pid(tpdu)
        self.fuzz_ud(tpdu)


def main():
    fuzz = fuzz_pid_ud()
    fuzz.create_fuzzed_pdus()
    fuzz.write_to_files('flash/', 'flash')


main()
```

C.5 Various fuzzing of UDH features

C.5.1 Fuzzing generic UDH

```
from create_sms_cases import fuzzer
import random
import Utils
import smspdu

class fuzz_generic_udh(fuzzer):

    alphabet = "0123456789ABCDEF"

    def fuzz_udh(self,tpdu):
        tpdu.tp_udhi = 1
        #create up to 15 tag value triplets
        number_of_headers = random.randint(1,15)
        udh = ""
        for i in range(number_of_headers):
            header = ""
            iei = Utils.to_hex(random.randint(0,255))
            ied = ""
            ied_length = random.randint(2,8)
            for j in range(ied_length):
                ied_byte = Utils.to_hex(random.randint(0,255))
                ied+=ied_byte
            ied_length = Utils.to_hex(ied_length)
            header += iei + ied_length + ied
            udh += header
        payload = ""
        for i in range(10):
            payload += self.alphabet[random.randint(0, len(self.
                                                    alphabet) - 1)]
        udh_len = Utils.to_hex(len(udh)/2)
        udh = udh_len + udh
        tpdu.tp_ud = ""
        tpdu.tp_udl = len(udh+payload)/2
        tpdu_string = tpdu.toPDU() + udh + payload
        return tpdu_string

    def fuzz_pid(self, tpdu):
        value = random.randint(0, 255)
        tpdu.tp_pid = value

    def fuzz_dcs(self, tpdu):
        value = random.randint(0, 255)
        tpdu.tp_dcs = value

    def fuzz_tpdu(self, tpdu):
        self.fuzz_pid(tpdu)
        self.fuzz_dcs(tpdu)
        pdu = self.fuzz_udh(tpdu)
        return pdu

    def create_fuzzed_tpdu(self):
```

```
        tpdu = smspdu.SMS_DELIVER.create('12345678','recipient',self.
                                                user_data)
        pdu = self.fuzz_tpdu(tpdu)
        return pdu

def main():
    fuzz = fuzz_generic_udh()
    fuzz.create_fuzzed_pdus()
    fuzz.write_to_files('generic_udh/', 'generic_udh')

main()
```

C.5.2 Fuzzing concatenated SMS

```
import random
import Utils
import smspdu
import get_dcs_length_mapping

class fuzz_udh_concat:

    alphabet = "0123456789ABCDEF"
    i = 0
    imsi = ""
    src = ""
    rpdu_base = "014803a1000000"
    test_cases = []
    user_data = ""
    rpdu_counter = 0

    padding_map = {6: 1, 5: 2, 4: 3, 3: 4, 2: 5, 1: 6}

    def __init__(self):
        #self.payloads = get_fuzz_payload.payloads
        self.number_of_sequences = 70
        self.dcs_length_mapping = get_dcs_length_mapping.dic

    def fuzz_dcs(self, tpdu):
        value = random.randint(0,255)
        tpdu.tp_dcs = value

    def fuzz_pid(self, tpdu):
        value = random.randint(0,255)
        tpdu.tp_pid = value

    def fuzz_tpdu(self, tpdu):
        self.fuzz_pid(tpdu)
        self.fuzz_dcs(tpdu)

    def insert_padding(self,udh,payload):
        #insert up to 6 bits of 0's at the start of payload, depending
                                                on length of UDH
```

```
        bits_in_udh = (len(udh) * 8)/2
        mod = bits_in_udh % 7
        numb_of_bits_to_add = self.padding_map[mod]
        payload_in_binary = bin(int(payload,16))[2:]
        for i in range(numb_of_bits_to_add):
            payload_in_binary = "0"+payload_in_binary
        decimal = int(payload_in_binary,2)
        payload_in_hex = str(hex(decimal).rstrip("L").lstrip("0x"))
        return payload_in_hex

    def create_udh(self, tpdu, tot, seq, csms):
        tpdu.tp_udhi = 1
        udh_len = "05"

        #Header triplet
        #Indicates concat SMS
        iei_a = "00"
        iei_a_len = "03"
        csms_ref_nr = Utils.to_hex(csms)
        total_sms_in_csms = Utils.to_hex(tot)
        sequence_numb = Utils.to_hex(seq)
        udh = udh_len + iei_a + iei_a_len + csms_ref_nr +
                                            total_sms_in_csms +
                                            sequence_numb
        payload = ""
        if seq < tot:
            for i in range(25):
                payload += self.alphabet[random.randint(0,len(self.
                                                    alphabet)-1)]
        else:
            #Just random string
            payload += "5E83229BFD06"
        padded_payload = self.insert_padding(udh,payload)
        ud = udh + padded_payload
        tpdu.tp_ud = ""
        tpdu.tp_udl = len(ud)/2
        tpdu_string = tpdu.toPDU() + ud
        return tpdu_string
        #ud/message itself can only be 134 bytes with this header

    def create_tpdu(self, tot, seq,csms_ref):
        tpdu = smspdu.SMS_DELIVER.create('32148567','recipient',self.
                                            user_data)
        self.fuzz_tpdu(tpdu)
        tpdu_string = self.create_udh(tpdu,tot,seq,csms_ref)

        return tpdu_string

    def pack_tpdu_in_rpdu(self,tpdu):
        len_in_hex = Utils.to_hex(len(tpdu)/2)
        tmp = self.rpdu_base + len_in_hex
        if self.rpdu_counter == 256:
```

```
            self.rpdu_counter = 0
        rpdu_mr = Utils.to_hex(self.rpdu_counter)
        ret = tmp[0]+tmp[1] + rpdu_mr + tmp[4:]
        self.rpdu_counter += 1
        ret += tpdu
        return ret

    def create_pdus(self):
        current_index = 0
        counter = 0
        concat_counter = 1
        csms_refnr = 0
        total_sms_in_concat = random.randint(2, 255)
        while counter < self.number_of_sequences:
            if concat_counter > total_sms_in_concat:
                total_sms_in_concat = random.randint(2,255)
                concat_counter = 1
                csms_refnr += 1
                counter += 1
            tpdu = self.create_tpdu(total_sms_in_concat,concat_counter,
                                                    csms_refnr)
            rpdu_with_tpdu = self.pack_tpdu_in_rpdu(tpdu)
            #drop random sms (4% chance)
            a = random.randint(0,100)
            b = random.randint(0,100)
            if a > 5:
                self.test_cases.append(rpdu_with_tpdu)

                # double sequences (5 % chance of happening)
                if (1 <= b <= 5):
                    self.test_cases.append(rpdu_with_tpdu)

                #scramble delivery order (5 % chance of happening)
                if (5 < b <= 10):
                    len_test_cases = len(self.test_cases)
                    if len_test_cases > 5:
                        c = random.randint(0,len_test_cases-2)
                        tmp = self.test_cases[current_index]
                        self.test_cases[current_index] = self.
                                                            test_cases[
                                                            c]
                        self.test_cases[c] = tmp
            else:
                current_index-=1
            concat_counter += 1
            current_index += 1

    def write_to_files(self, subfolder, filename_base):
        file_counter = 1
        start = 0
        step = 50
        total_number_of_pdus = len(self.test_cases)
```

```
        while start < total_number_of_pdus:
            current = self.test_cases[start:step]
            with open('TestCases/' + subfolder + filename_base + str(
                                                    file_counter) + '.txt',
                                                     'a') as file:
                for i in range(len(current)):
                    out = current[i]
                    file.write(out)
                    file.write("\n")
            start += 50
            step += 50
            file_counter += 1

def main():
    fuzz = fuzz_udh_concat()
    fuzz.create_pdus()
    fuzz.write_to_files('concat_sms/', 'concat_sms')

main()
```

C.5.3 Fuzzing EMS

```
from create_sms_cases import fuzzer
import random
import Utils
import smspdu

class fuzz_pid_ud(fuzzer):

    alphabet = "0123456789ABCDEF"

    def fuzz_udh_ems(self,tpdu):
        tpdu.tp_udhi = 1
        udh = ""
        iei = Utils.to_hex(random.randint(10,31))
        ied_length = random.randint(1,7)
        ied = ""
        for j in range(ied_length):
            ied_byte = Utils.to_hex(random.randint(0,255))
            ied += ied_byte
        ied_length = Utils.to_hex(ied_length)
        udh += iei + ied_length + ied

        payload = ""
        for i in range(random.randint(10,120)):
            payload += self.alphabet[random.randint(0, len(self.
                                                    alphabet) - 1)]
        udh_len = Utils.to_hex(len(udh)/2)
        udh = udh_len + udh
        tpdu.tp_ud = ""
        tpdu.tp_udl = len(udh+payload)/2
        tpdu_string = tpdu.toPDU() + udh + payload
```

```
        return tpdu_string

    # Not fuzzing of pid, change later ?
    def fuzz_pid(self, tpdu):
        value = 0
        tpdu.tp_pid = value

    # Not fuzzing of pid, change later ?
    def fuzz_dcs(self, tpdu):
        value = 0
        tpdu.tp_dcs = value

    def fuzz_tpdu(self, tpdu):
        self.fuzz_pid(tpdu)
        self.fuzz_dcs(tpdu)
        pdu = self.fuzz_udh_ems(tpdu)
        return pdu

    def create_fuzzed_tpdu(self):
        tpdu = smspdu.SMS_DELIVER.create('12345678','recipient',self.
                                                    user_data)
        pdu = self.fuzz_tpdu(tpdu)
        return pdu

def main():
    fuzz = fuzz_pid_ud()
    fuzz.create_fuzzed_pdus()
    fuzz.write_to_files('ems/', 'ems')

main()
```

C.5.4 Fuzzing (U)SIM Data Download

```
from create_sms_cases import fuzzer
import random
import Utils
import smspdu

class fuzz_usim(fuzzer):

    alphabet = "0123456789ABCDEF"

    def fuzz_udh_usim(self,tpdu):
        tpdu.tp_udhi = 1
        udh = ""
        iei = Utils.to_hex(112)
        udh += iei
        #USIM does not require any IED data
        payload = ""
        for i in range(random.randint(10,120)):
            payload += self.alphabet[random.randint(0, len(self.
                                                    alphabet) - 1)]
```

```
        udh_len = Utils.to_hex(len(udh)/2)
        udh = udh_len + udh
        tpdu.tp_ud = ""
        tpdu.tp_udl = len(udh+payload)/2
        tpdu_string = tpdu.toPDU() + udh + payload
        return tpdu_string

    # pid must be 127 for usim
    def fuzz_pid(self, tpdu):
        value = 127
        tpdu.tp_pid = value

    # Dcs must be 246 for usim
    def fuzz_dcs(self, tpdu):
        value = 246
        tpdu.tp_dcs = value

    def fuzz_tpdu(self, tpdu):
        self.fuzz_pid(tpdu)
        self.fuzz_dcs(tpdu)
        pdu = self.fuzz_udh_usim(tpdu)
        return pdu

    def create_fuzzed_tpdu(self):
        tpdu = smspdu.SMS_DELIVER.create('12345678','recipient',self.
                                                    user_data)
        pdu = self.fuzz_tpdu(tpdu)
        return pdu

def main():
    fuzz = fuzz_usim()
    fuzz.create_fuzzed_pdus()
    fuzz.write_to_files('USIM/', 'USIM')

main()
```

C.6 Utilities

C.6.1 Import fuzz payload

```
from os import walk
import random

files = []
payloads = []
path = "FuzzLists"

def read_files():
    for (dirpath,dirname,filenames) in walk(path):
        for i in range(len(filenames)):
            files.append(path+"/"+filenames[i])
```

```
def get_payload():
    #The entire FuzzLists have 153 795 fuzzing payload, which is
                                        perhaps a bit much.
    #Since we want our payload to only be of 130 characters, we
                                        exctract only those who have
                                        130 of length or less
    #As well, we take only take each 10'th element
    j=10
    random_index = random.randint(0,10)
    for file in files:
        with open(file,'r') as f:
            lines = f.readlines()
            for i in range(len(lines)):
                #add random payload (from (0,10) + i) to list every 10'
                                                    th entry.
                if len(lines[i]) < 160 and j>=10:
                    payloads.append(lines[i-random_index])
                j += 1
                if j == 11:
                    j=0
                    random_index = random.randint(0,10)
        f.close()

read_files()
get_payload()
```

C.6.2 DCS length mapping

```
import smspdu

#dic holds maximum length allowed for user data when we use the
                                    different encodings
dic = {}
default = 160
for j in range(257):
    try:
        tpdu = smspdu.SMS_DELIVER.create('sender','recipient',20*'Hello
                                            World!',tp_dcs=j)
        #program gets here if dcs encoding can handle 160 chars
        dic[j] = default
        tpdu.tp_dcs = j
        tpdu_string = tpdu.toPDU()
    except Exception as inst:
        stacktrace = str(inst.args)
        if "140" in stacktrace:
            dic[j] = 140
        else:
            #Now we don't know the required max length because the
                                                encoding is reserved,
                                                thus for safe measures
                                                just set 120
```

```
            dic[j]=120
```

C.6.3 Hex converter

```
def to_hex(decimal):
    hex_numb = hex(decimal)
    ret = ""
    for i in range(2,len(hex_numb)):
        ret+=str(hex_numb[i])
    if len(ret) != 2:
        ret = "0"+ret
    return ret
```