Andreas Nilsen

Reverse Engineering the Home Monitoring Unit in the Pacemaker Ecosystem

Master's thesis in Communication Technology
Supervisor: Marie Elisabeth Gaup Moe
Co-supervisor: Guillaume Bour
July 2022

Andreas Nilsen

Reverse Engineering the Home Monitoring Unit in the Pacemaker Ecosystem

Master's thesis in Communication Technology
Supervisor: Marie Elisabeth Gaup Moe
Co-supervisor: Guillaume Bour
July 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Title:	Reverse Engineering the Home Monitoring Unit in the Pacemaker Ecosystem
Student:	Andreas Nilsen

Problem description:

Today, we are surrounded by smart devices that count our steps, measure our heart rate or track our location at all times. The trend of the last decade is to interconnect these devices to give them more functionality and smart capabilities. In this process, where time to market is a crucial factor, security features often seem to be overlooked or deemed secondary. When it comes to medical devices this trend can have severe consequences. So the question is, how secure are these medical devices?

Most pacemakers and ICDs are made by a handful of private companies worldwide, such as Biotronik, Medtronic, and Abbott (former St. Jude Medical). Even though these companies have to conform to government laws and regulations their devices and software are not accessible for review, making these medical devices black boxes from a security perspective. As they are getting interconnected and make use of wireless technologies, the devices are susceptible to new attack vectors. This leads to questions about undiscovered security flaws in the devices.

The lack of peer review and limited security research on medical devices add additional layers of uncertainty. Because of the uncertainty of adequate security implementations and having the ability to extract binaries from the home monitoring units, I will follow a reverse engineering approach to better understand the security measures implemented in such devices.

Responsible professor:	Marie Elisabeth Gaup Moe, NTNU
Supervisor:	Marie Elisabeth Gaup Moe, NTNU
Co-Supervisor:	Guillaume Bour, SINTEF

Abstract

Research on medical devices in the past has uncovered security vulnerabilities that threaten the safety and privacy of patients. Over the past decade, Internet of Things (IoT) devices, as well as medical devices, have become connected. The implementation of wireless communication has enabled remote monitoring of patients and simultaneously opened an additional attack surface. The lack of insight into the security practices of medical device manufacturers has led research organizations to investigate and perform security analysis to assess their security implementations. Recent research on medical devices has shown that they contain security vulnerabilities that an adversary could exploit to threaten patient safety and privacy. This Master's thesis is part of a collaborative research effort between NTNU and SINTEF.

In this thesis we follow a black box approach to reverse engineering the cryptographic code sections of the Biotronik CardioMessenger Smart 3G Home Monitoring Unit's (HMU) firmware. To perform the analysis, we utilize free and open-source software with commercial off-the-shelves (COTS) tools on the devices that are available in the SINTEF lab. The results show that physical access to the HMU and its on-board debugging interface, makes physical and remote attack scenarios that threaten the patient's safety plausible. Our analysis show that the HMU might be susceptible to a Denial-of-Service attack based on our new knowledge of its communication protocol. My thesis also proposes the necessary mitigations that are needed to secure the HMU.

The reverse engineering methodology and process enable further research on the Biotronik HMU's remaining interfaces, as well as HMUs of other manufacturers. The reverse engineering methodology developed in this thesis can therefore be used as a guideline for future reverse engineering projects.

Sammendrag

Tidligere forskning på medisinske enheter har avdekket sårbarheter som truer pasientens sikkerhet og personvern. I løpet av det siste tiåret har medisinske enheter fulgt i fotsporene til IoT og blitt tilkoblede. Implementeringen av trådløs kommunikasjon har muliggjort telemonitorering av pasienter og samtidig åpnet for nye angrepsoverflater. Mangelen på innsikt i sikkerhetspraksisen hos produsenter av medisinsk utstyr, har ført til at forskningsorganisasjoner gransker og gjennomfører sikkerhetsanalyser for å vurdere deres sikkerhetsmekanismer. Nylig forskning på medisinske enheter har vist at de inneholder sikkerhetssvakheter som en trusselaktør kan utnytte for å true pasientens sikkerhet og personvern. Denne masteroppgaven er del av et forskningssamarbeid mellom NTNU og SINTEF.

I denne masteroppgaven følger vi en black box-fremgang for å dekonstruere de kryptografiske kodeseksjonene i programvaren til Biotronik sin CardioMessenger Smart 3G hjemmemonitoreringsenhet (HMU). For å utføre analysen har vi brukt gratis og åpen-kildekode programvare med hyllevareutstyr på enhetene som er tilgjengelig i SINTEFs lab. Resultatene viser at fysisk tilgang på HMU og kretskortets feilsøkingsgrensesnitt, gjør det rimelig å anta at både fysiske og trådløse angrepsscenarioer som truer pasientens sikkerhet er mulige. Vår analyse at HMU-en muligens er sårbar for et tjenestenektangrep basert på våre nye funn i kommunikasjonsprotokollen. Denne masteroppgaven foreslår også de nødvendige mottiltakene som trengs for å sikre HMU-en.

Reverse engineering-metoden og prosessen beskrevet i oppgaven muliggjør videre forskning på Biotronik HMU-ens gjenværende kodeseksjoner, i tillegg til forskning på HMU-er fra andre produsenter. Reverse engineering-metoden utviklet i denne oppgaven kan derfor bli brukt som en fremgangsmåte for fremtidige reverse engineering-prosjekter.

Preface

This Master's Thesis is the final deliverable of a 5-year MSc programme in a Master of Science Degree in Communication Technology with a specialization in Information Security, at the Department of Information Security and Communication Technology at the Norwegian University of Science and Technology (NTNU).

The thesis research is performed by Andreas Nilsen in a collaborating project between NTNU and SINTEF. The research is part of a joint project effort on medical device security over the past five years. The work is supervised by Marie Elisabeth Gaup Moe and co-supervised by Guillaume Bour from SINTEF. Marie Elisabeth Gaup Moe is also the responsible professor from NTNU.

Acknowledgements

I would like to thank my supervisors Marie Elisabeth Gaup Moe and Guillaume Bour for their time, feedback, and guidance throughout this project. Working on the field of medical device security and reverse engineering has been both exciting and challenging - and occasionally really frustrating. Having such experienced supervisors has been a great help and enabled me to learn a lot about medical device security. I am truly grateful for their continuous support and patience throughout this process.

I would also like to thank NTNU and SINTEF for collaborating and proposing such an exciting project. The research on medical devices with real-world impact is inspiring, rewarding, and with the purpose of making a difference for people in a vulnerable position, all around the globe. SINTEF has also aided my project with a lab and all the needed resources, as well as financing one of my trips to Trondheim to perform lab work during the COVID pandemic. I would also like to thank the advisors at the Department of Information Security and at the Communication Technology and at Faculty of Information Technology and Electrical Engineering for their help and facilitation during the process of finishing my degree and Master's Thesis.

Lastly, I would like to give a special thanks to my girlfriend Linn, who has been encouraging and supporting me. I would definitely not have been able to finish my thesis without her support.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

AAMI Association for the Advancement of Medical Instrumentation.

AIMD Active Implantable Medical Device Directive.

API Application Programming Interface.

APN Access Point Name.

CIA Confidentiality Integrity Availability.

CM CardioMessenger.

COTS Commercial off-the-shelf.

CRC Cyclic Redundancy Check.

CSRF Cross-Site Request Forgery.

CVE Common Vulnerabilities and Exposures.

DoS Denial-of-Service.

EoL End-of-Life.

ETSI European Telecommunications Standards Institute.

EU European Union.

EUDAMED European Database on Medical Devices.

FDA Food and Drug Administration.

Ghidra Generic Hexidecimal Integrated Decompiling Reverse-Engineering Architecture.

HMSC Host Mass Storage Class.

HMU Home Monitoring Unit.

ICD Implantable Cardioverter-Defibrillator.

ICSCCC International Conference on Secure Cyber Computing and Communications.

IDS Intrusion Detection System.

IMD Implanted Medical Device.

IoT Internet of Things.

IVDD In vitro Diagnostic Directive.

IVDR In vitro Diagnostic Medical Devices Regulation.

JTAG Joint Test Action Group.

MDD Medical Device Directive.

MDR Medical Devices Regulation.

MICS Medical Implant Communication System.

MitM Man-in-the-Middle.

MMI Multi Media Interface.

NSA National Security Agency.

NSM Norwegian National Security Authority.

NTNU Norwegian University of Science and Technology.

OSSTMM Open Source Security Testing Methodology Manual.

OWASP The Open Web Application Security Project.

PCB Printed Circuit Board.

RF Radio Frequency.

SHARP SINTEF Hardware Hacking Raspberry Pi.

SRE Software Reverse Engineering.

SSID Service Set Identifier.

SVD	System View Description.
SWD	Serial Wire Debug.
TPM	Trusted Platform Module.
UMP-AMI	Ultra Low Power Active Medical Implant.
UMTS	Universal Mobile Telecommunications System.
USRP	Universal Software Radio Peripheral.
VM	Virtual Machine.
W-CDMA	Wideband Code-Division Multiple Access.

Chapter 1 Introduction

1.1 Context and Motivation

In the past few decades technological advancement has seen a rapid growth after the invention of the Internet. The Internet has revolutionized the way we interact both personally with social media and technically between machines. Today, most household electronics can be bought as a smart device, which implies inter-connectivity to a local network or remotely to the Internet. Usually the smart device can be controlled through an application on a computer or smartphone residing in the same local network. These devices are part of the umbrella term, Internet of Things, or IoT devices for short.

Connecting common household appliances are deemed both practical and innovative, and therefore found in numerous homes nowadays. Appliances such as cameras, automated lighting systems, alarm systems, and cameras. Devices and systems that used to be standalone devices and today are used to improve security or to optimize energy consumption in a modern home. Taking part in the IoT revolution has enabled these devices to collect data and communicate with other devices on a the network. This connectivity opens the possibility of device vulnerabilities being exposed on a network-level.

Although cyber attacks and hacking might sound frightening in the context of common IoT devices, there are other devices that also used to be standalone and disconnected, which have become connected in the past decades. Devices that are life-dependent for many people worldwide - that is implanted medical devices (IMD). Medical devices have followed in the same footsteps and have become a part of the IoT revolution. With the ability to be connected, a medical device can be used actively in the treatment and monitoring of patients. It can also be used to notify medical professionals about events concerning the patient's well-being. This opens the possibility of adjustable and customized treatment on a patient-level. It also enables home care without the need of physical appointments at the hospital. Telecardiology consequently saves large amounts of resources for doctors, companies, and the society as a whole. It also increases the safety of patients while saving them from frequent clinic visits, and allows for a closer follow-up of their treatment.

The Pacemaker Ecosystem

The pacemaker or ICD is dependent on several other devices function and relay information. Most pacemaker ecosystems consists of the same devices, the pacemaker, the home monitoring unit, the programmer(i.e configuration device), and the vendor servers. Between these devices are various communication channels to relay the information.

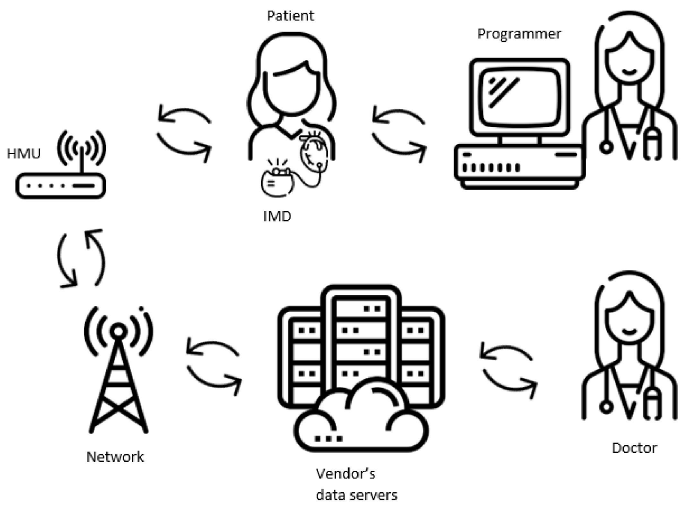


Figure 1.1: A figure of the Pacemaker Ecosystem

Figure 1.1 shows all the devices in the pacemaker ecosystem[1]. The communication between the devices are indicated with arrows. Each of the devices in the pacemaker ecosystem is explained further below.

The pacemaker or ICD, the implanted medical device, is used to monitor and support the heart by generating electrical impulses. The pacemaker logs are stored in its internal memory and periodically transmitted wirelessly to the HMU. The pacemaker has a wireless interface which communicates with the HMU and the programmer through the Medical Device Radiocommunications Service (MICS) communication spectrum, also referred to as MedRadio [8]. Modern pacemakers are programmable and can be adjusted to fit the individual needs of each patient. The

[1]Figure 1.1 was designed using resources from Flaticon

pacemaker is configured and programmable through the Programmer.

The Home Monitoring Unit or HMU, is a stationary device kept in the patients home. It is in many ways comparable to a home router, and receives the short range wireless data from the pacemaker and then transmits the data using various communication mediums depending on the manufacturer and model. Older HMU models were connected by using telephone or Ethernet cables, while the latest generations of HMU devices use either 2nd/3rd generation telecommunications services or a connection through the patient's home Wi-Fi. The latest generations of HMUs are also smaller and as mobile as a common smart phone, which also enables patients to transmit their health- and monitoring data outside of their homes.

Vendor's Servers receive the logs from the HMU after they are transmitted through the appropriate communication medium. These servers are beneficial to store large amounts of patient data because of the limited size and processing power in the pacemaker and HMU. The cloud servers also makes patient data available on-demand for the medical professionals, reducing the amount of hospital visits for patients.

Programmer is a stationary configuration device that communicates with the implanted medical device to configure the IMD's settings, to retrieve operation logs from the IMD, or to update the IMD's firmware [9]. The programmer head is a device at the end of a cord connected to the programmer, that is placed on the patients chest for the programmer to wirelessly communicate with the IMD. The programmer device is located in a hospital and is operated by a medical professional.

1.2 Scope

Our project will focus on the cybersecurity of a HMU device from Biotronik. The research will focus on the CardioMessenger Smart 3G and how security is implemented in the firmware. Specifically, we will perform a reverse engineering process with suitable tools and scripts to find certain code sections of interest - mainly those related to its cryptography functionality. The end goal is to find and document the functionality of one of its communication protocols. The HMU's protocol for sending and receiving data to and from the data servers. We will also document the reverse engineering process of a medical device, which can be used as a guideline for future reverse engineering projects.

The manufacturer and devices are chosen based on devices available in the lab at

SINTEF. The devices we have available have been bought at online marketplaces (e.g eBay) or been donated. The code we will be analyzing is the result of extracting (i.e dumping) the memory of the Biotronik CardioMessenger Smart 3G. This procedure was first performed by my supervisor Guillaume Bour, and we have replicated this procedure in the same lab and with the same equipment. We also modified a couple of the scripts to test and extract through another debugging protocol. The process is thoroughly explained in Appendix A. The research performed in this project replicates some of the previous research by reusing some of the previous used tools and methodology, it extends the previous research by focusing on the latest generation of the HMU. Since the CardioMessenger Smart 3G is the latest model from Biotronik, my findings will be highly relevant because patients worldwide use this device today.

1.3 Hypothesis and research questions

The HMU device we will research is the latest model of home monitoring units from Biotronik. We expect it to be more robust and secure in both its hardware and software compared to its predecessors. Its security implementations should be up to today's standards and good practices relating to both security- and legal perspectives [10, 11]. Previous work on medical devices have found various vulnerabilities, and it is our intention to confirm whether these have been addressed and mitigated in the latest model.

It is also our intention to identify and analyze the proprietary communication protocol of the CardioMessenger Smart 3G. Previous work from the Pacemaker Hacking Project at SINTEF have analyzed the communication protocol of an older HMU device from the same manufacturer Biotronik, an approach from the outside, by establishing a connection to a custom base station. However, our approach will be from the inside of a newer HMU model from the same manufacturer by extracting its memory and analyze its internal code.

With these expectations and intentions, I have formulated the following research questions and research objectives to guide my project:

Research Questions (RQ):
RQ1: To what extent is the HMU implementing security features to preserve the patients safety?
RQ2: To what extent is the latest HMU protecting the patients privacy?
RQ3: To what extent is the latest HMU protecting the patients personal data in its communications?

Research Objectives (RO):
RO1: Analyze the security implementations in the extracted binaries of the CardioMessenger Smart 3G
RO2: Describe the vulnerabilities and improvements in the latest CardioMessenger Smart 3G, and compare to findings of the former CardioMessenger II-S or T-Line
RO3: Propose improvements and/or solutions to findings in the CardioMessenger Smart 3G

1.4 Outline of the Thesis

Chapter 2: Background introduces and lays the background information for this thesis. Firstly, a review of work related to implanted medical devices - IMDs. Going through their functionality and their security vulnerabilities. Then some technical background about the memory dumping procedure and the used debugging interfaces. Lastly, some technical information about reverse engineering tools and practices.
Chapter 3: Methodology explains different methodologies I have used to approach the work throughout this thesis. Each method is described in detail along with my reasoning for its use case.
Chapter 4: Results goes through all the reverse engineering process and results from the research. This chapter also gives explanations on how these results were achieved and references to detailed descriptions in one of the appendices. It starts with an exploration of the function hierarchy, and ends with identifying the communication protocol.
Chapter 5: Mitigation goes through the countermeasures related to our findings from the Results chapter.
Chapter 6: Discussion is about the impact of our work and findings on the HMU. This chapter also illustrate attack scenarios that the HMU might be susceptible to. This chapter also highlights need for future work, on the HMU and other devices in the pacemaker ecosystem.
Chapter 7: Conclusion Lastly, this chapter answers some of the research questions and concludes the thesis.

Chapter 2
Background

2.1 Cybersecurity Terminology

In this thesis we use terminology that is common to find in the field of cybersecurity. Since we use these terms through our thesis, it is useful to define them early in this chapter. The following terms are formulated from the NIST glossery [12].

An **asset** can be an application, a system, equipment, or personnel - anything with value to the company. An asset can be either tangible such as hardware, software, or a device, or it can be intangible such as information, intellectual property, or reputation.

A **threat** is a circumstance or event that potentially can adversely impact or harm an asset.

A **vulnerability** is a flaw or a weakness in a system which can be exploited or triggered by a threat. A vulnerability can take the form of a misconfiguration in a system, poor system design, or human error. When identifying vulnerabilities it is common to look for the vulnerabilities that are the easiest to take advantage of - the low-hanging fruits.

An **exploit** is a tool that takes advantage of a vulnerability or a flaw in a system.

A **risk** is a measure of the extent an asset is threatened by an event or circumstance.

A **zero-day**, alternatively an n-day, refers to a vulnerability that is previously unknown and exploits software or hardware.

2.2 Related Work

In this section we present research and the findings that are related to the pacemaker ecosystem. Some of the papers are from large research organizations, international conferences, medical associations, and also included are previous papers and research from the NTNU and SINTEF colleborative pacemaker project.

In 2008, Halperin et al. studied an ICD and its privacy and security properties [13]. They managed to reverse the ICD's wireless communication protocol to the programmer using an oscilloscope and a software-defined radio. This enabled them to develop different radio-based attacks that could compromise a patients safety and privacy.
In 2010, Færestrand published a journal through the Norwegian Medical Association emphasizing the positive use cases of telecardiology [14]. The use of IMDs are practical for both the patient and the medical professionals, as it enables everyone to save time and money. Færestrand also writes that the remote monitoring devices' ability to detect anomalies technically and physically, improve the patients safety. Each scheduled clinic visit can be up to six months, which means that an event is detected immediately instead of weeks or months after the event occurred.

There has been issues related to the technical wiring and the functionality of the IMD if it is somehow manipulated or malfunctioning after being implanted. Even though these devices occasionally have technical errors in wiring or function, they are from a medical point of view deemed safe and effective for diagnostics and monitoring a patients diagnosis. They believe that remote monitoring is forward-thinking and that patient and medical personnel are satisfied with the practicality of telecardiology and remote monitoring.
In the paper written by Fu in 2015, they go through the history of some medical devices and their security issues [15]. Issues related to software on the devices and the possibility of wireless attacks to modify the functionality of the IMD. A proven example of modifying functionality is to disable the ICDs ability to send electrical pulses to correct the heart beat - an attack that can cause a deadly or abnormal heart rhythm.

They also mention the pre-market guidance for management of cybersecurity in medical devices from the Food and Drug Administration (FDA), on purchases of certain medical devices that pose a cybersecurity risk, risk analysis of medical device applications before market clearance, and their continuous surveillance of of emerging cybersecurity risks in this area [16].

They also reference several papers finding vulnerabilities in various medical devices. Such as a vulnerability in an insulin pump, demonstration of a computer worm spreading from a computer to an automated external defibrillator, a compromised website belonging to a ventilator manufacturer that modified the softwares to include a malicious payload in their software updates, a malware infected drug compounder running on Windows XP embedded operation system, and more.

The paper then proceeds to claim that security problems often arises where abstractions meet, as in the digital to analog. This is especially the case for medical devices. They also urge manufacturers of medical devices to address the cybersecurity risks in the initial requirements engineering and design time, and that they continue support and perform post-market surveillance during the device lifecycle.

In 2016, Muddy Waters released a report performed by the cybersecurity research firm MedSec, on the security issues of the St. Jude Medical's devices [17]. Their findings were so severe that they urged St. Jude Medical to recall and remediate their devices. They also estimated that St. Jude Medical would lose close to half their revenue because these devices made up 46% of St. Jude Medicals revenue in 2015.

They found that they could perform a crash attack which could set a dangerous pacing rate and a battery drain attack on the implantable cardiac device. They also found that these attacks can be performed roughly within a few meters radius, and could theoretically be performed on a large scale. The report also makes a point of how many hundreds of thousands of these devices that are distributed for use or sold secondhand on eBay. Since they found that these Merlin@Home devices lack the basic forms of security, such as encryption and authentication, an attacker could easily buy one of these devices online cheap, then impersonate and communicate with a St. Jude Medical cardiac device.

In 2016, Marin et al. found that IMDs often use proprietary protocols with limited security when communicating wirelessly to a programmer [18]. They were able to fully reverse engineer a proprietary communication protocol between a programmer and the latest generation of a widely used ICD. They showed that reversing this protocol was possible for an adversary with limited resources and cabapilities, and by only using inexpensive commercial off-the-shelf (COTS) equipment.

They also found functionality attempting to obfuscate the transmitted data, compared to previous research on medical devices which did not find any security functionality. Additionally, they conducted attacks that imposed on the patients privacy, caused a denial of service (DoS) and compromised the patients safety. They also claim that their findings apply to at least 10 types of ICDs that were on the market at the time. Lastly, they proposed several short- and long-term countermeasures for the discovered vulnerabilities.

In 2017, Whitescope performed a review on the monitoring devices of four unnamed major IMD vendors [19]. They found that their inherent architecture and implementation interdependencies are susceptible to security risks that might impact of the overall confidentiality, integrity and availability (CIA) of the ecosystems.

Their list of findings are related to embedded hardware security and software implementation issues. They were able to obtain all the devices from public sources, open and accessible debugging interfaces on the PCBs, extract the file systems in its entirety of the medical devices, they found credentials and infrastructure data in cleartext, use of third-party libraries, lack of digital firmware signatures, lack of file system encryption, lack of authentication on the programmer device that enables physicians to configure the IMDs, and much more.

The fact that their findings were more or less consistent between the four different vendors show the lack of security implementations in medical devices in general. They reported that these findings highlight for all vendors to perform an in-depth evaluation of their security.

In 2021, on the second international conference on secure cyber computing and communications (ICSCCC), a threat model of the entire pacemaker ecosystem was presented by Manikandan and Shiju [20]. They explain MICS, which is the mobile radio frequency service used between the IMD to and from either the programmer or HMU.

They use attack trees to visually represent threats on the different devices of the pacemaker ecosystem, and how it imposes on their CIA triad. They also claim there is a lack of authentication between the devices using MICS and that an attacker within close proximity of 5m can trigger the RF circuitry of the IMD. They also have suggestions on how this threat to medical devices using MICS can be mitigated.

2.2.1 Results from the Pacemaker Hacking Projects at SINTEF

In 2018, Kristiansen and Wilhelmsen worked mainly on the programmer or controller device in the pacemaker ecosystem [21]. They were able to extract the file system of a programmer and create a running virtual machine (VM) with it. They found several security issues such as an unencrypted hard drive and no authentication mechanism to access and use the device. Their work also opens the possibility to decrypt patient data exported from the programmers, and they claim it is possible for any Biotronik programmer at the time.

In 2019, Bour, one of my supervisors, worked extensively on multiple HMUs from different vendors. They performed a series of embedded hardware security analysis which included Merlin@Home from Abbott, a HMU from Medtronic, and multiple versions of Biotroniks CardioMessenger HMU series. They were able to identify the components on the PCBs and access their debugging interfaces by soldering a connection to the PCBs, which gave full access to the microprocessor and unencrypted memory.

Bour also worked on identifying the proprietary communication protocol of the CardioMessenger II-S TLine, an older HMU model from Biotronik, between the HMU and the backend data servers. By emulating a modem on a computer and connecting to the HMU, they were able to analyze the HMU's transmissions and identify the byte structure of the protocol.

Bour also showed that a Man-in-the-Middle (MitM) attack was feasible against the HMU and the backend private data servers due to the lack of mutual authentication.

In 2019, Lie demonstrated that using only COTS equipment and open source software connected to a Universal Software Radio Peripheral (USRP), it is possible to connect to an HMU over GSM mobile communications [22]. By establishing a connection to the HMU and eavesdropping on the wireless transmission, they found that the data was encrypted and that Biotronik were using a communication channel within a private APN. They were also able to report on multiple attacks including spoofing the data server, establishing a wireless connection to the HMU without authenticating, and if they had physical access to the device they could decrypt the encrypted data sent from the HMU intended towards the data servers.

In 2020, Kok and Markussen did preliminary hardware security testing of Biotroniks CardioMessenger II-S T-Line and GSM version, and hardware analysis of the CardioMessenger Smart 3G [23]. They proceeded to develop a fuzzing framework for the modem on the CM Smart 3G HMU based on COTS equipment such as a USRP connected to a computer running on a modified OpenBTS installation. The fuzzer is applicable to any modem, not just the modems on the HMUs made by Biotronik. They also showed that the fuzzing framework was able to both interrogate and cause its modem to crash, indirectly denying the communication service on the HMU. Potentially risking the safety of the patient.

In 2022, Bour et al. published an article at the Biodevices 2022 conference [24]. The paper was a continuation of Bour's work from 2019, and presented their hardware security findings from 2019 and network analysis of the Biotronik CardioMessenger II-S TLine HMU's communication protocol. This was achieved by setting up a custom software-defined base station, forcing a downgraded GSM connection, for the transmissions intended for the backend server and listening on its communications. This paper was published after having completed a coordinated vulnerability disclosure process, resulting in a medical advisory from CISA related to their earlier findings [25].

The article is a new representation of Bour's findings from 2019. They also represent two new and potential attack scenarios that are feasible on the pacemaker ecosystem in the light of the findings. They also state that these findings enable an adversary to weaponize the HMU, and to act as a Man-in-the-Middle since the CardioMessenger II-S TLine's wireless transmissions were analyzed, and parts of the protocol was identified.

2.3 European Legislation & Regulations

This section contains the latest European legislation on medical devices and in vitro diagnostic medical devices, best practice guideline documents from ENISA, and the implications of the latest Norwegian security act.

2.3.1 MDR & IVDR

In December of 2019, the Medical Coordination Group published a guidance document to provide manufacturers with guidelines on how to comply with the two new EU regulations 45/2017 (MDR) and 746/2017 (IVDR) on the cybersecurity of medical devices [10, 26, 27].

The MDR and IVDR were entered into force on the 25th of May 2017, and their dates of application were on 26th of May 2021 and 26th of May 2022, respectively. The objective of the new regulations were to replace three older directives, Medical Device Directive (MDD), the Active Implantable Medical Device Directive (AIMD), and In vitro Diagnostic Directive (IVDD). The new regulations create a regulatory framework that improves the safety, quality, and reliability of the medical devices in the EU.

The MDR contains regulation on all medical devices and their accessories for sale and service in the EU [26]. It is an extensive document and it contains several noteworthy points that relates to the pacemaker ecosystem. The MDR has a list of general obligations for the manufacturers of medical devices, and one of the items one the list is to establish and maintain a system for risk management. The risk management systems needs to identify and analyze threats for each device. Additionally, they need to evaluate each threat and eliminate or control the threat by secure design, adequate protection measures, or provide information and training.

The manufacturers also need to implement a quality management system to address various aspect such as identifying general safety and performance requirements, resource management and control of suppliers, implement and maintain a post-market surveillance system, implement a process for reporting serious incidents and take corrective actions based on safety, a management of corrective and preventive actions, and processes to monitor, measure and analyze data for product improvement.

To improve traceability and available information, the MDR also state that every patient shall have an implant card with information on the device, information about the manufacturer, warnings or precautions the patient or medical professionals need to be aware of in certain environments or conditions, information about the expected lifetime of the device, and information to ensure the safety of the device.

To improve identification and traceability within the supply chain of medical devices, the MDR states that a manufacturers and authorized representatives shall cooperate, and that a Unique Device Identification System (UDI) shall be created. The information within the UDI database system shall be publically available.

To ensure the safety and performance of the medical devices, the manufacturer needs to create a summery of the device's safety and clinical performance. The summary also needs to be written in such a way that it is clear for the user. The summary needs to contain identification of the device and manufacturer, device's intended purpose, a description of the device, a reference to older devices and their differences, a description of the accessories of the medical device, and information on the residual risks and precautions. The summary also needs to be publically available through the new European database on medical devices, EUDAMED.

Each manufacturer also needs to have a certificate of conformity to the regulations and necessary safety markings, such as CE, to sell their medical devices in the EU. The certificate is intended to guarantee the safety and performance of the devices, and show that the manufacturer are conforming to all legal and regulations.

The MDR also states that devices with electronic programmable systems, including software, shall be designed to ensure performance, reliability and repeatability. In the case of any faulty condition, it needs to take appropriate actions to reduce or eliminate the risks or impairment on performance. Devices with software shall also be developed with *state of the art* principles in its development life cycle, risk management, validation and verification, and information security.

After the manufacturer has gained a certification and sold medical devices on the European market, they shall plan, implement and maintain a post-market surveillance system. The surveillance system is intended to be part of the manufacturers quality management system, and shall systematically gather and analyze relevant data from the devices on safety, quality and performance throughout the device's lifetime. The surveillance plan is intended to be used to improve the benefit-risk assessment, improve manufacturing and design, improve safety performance, and to detect trends of incidents that impose on the patients health or safety.

The MDR also establishes a program for market surveillance that is enforced by appointed authorities to perform the necessary checks on the conformity and performance of devices on the market. These checks include documentation from the manufacturer, physical, and laboratory checks of device samples. If the sample or device is shown to present an unacceptable risk or non-complience with the regulations, they will be forced to perform justified corrective actions to comply with the regulations. Based on the presented risk, devices may be recalled or withdrawn from market for a reasonable time period.

Similarly to the MDR, the IVDR contain many similar articles to be enforced [27]. The IVDR specifies a similar list of obligations for the manufacturer related to compliance and deliver information about their products. The in vitro devices also need the CE marking on the devices to show conformity to the regulation.

The UDI system is also applied to in vitro medical devices, similarly to all medical devices defined by the MDR. The in vitro devices shall also be included in the EUDAMED system to enhance their traceability. The IVDR share the same safety and performance requirements as defined in the MDR. A few key differences between the IVDR and MDR, is the applicability of all medical devices and in-vitro medical devices. Additionally, the IVDR require a methodically sound performance evaluation procedure to demonstrate scientific validity, analytical performance, and clinical performance. They shall implement and maintain a performance evaluation plan that includes general safety and performance requirements, a specification of methods and tools used in analytical and clinical performance, identification of relevant standards, guidance and best practice documents. For the software on the devices, the manufacturer needs to reference and specify the sources of their data used in the decision making process. To demonstrate the scientific validity and the analytical and clinical performance, the manufacturer needs to perform a systematic literature review of the scientific relevant data to the device, its intended purpose, and remaining issues that need to be addressed. They also need to review peer-reviewed scientific literature, perform proof of concept studies, and analyze results from clinical performance studies.

The IVDR requires a post-market surveillance plan to secure quality, performance and safety, similarly to the MDR. The IVDR specifies that manufacturers of devices classified as C and D, in general term devices that directly interacts with the biological, shall prepare a periodic safety update report(PSUR) to summarize the results and gathered data of post-market analyses.

The IVDR has a section on vigilance and reporting of serious incidents. In case of an incident, they shall report to competent authorities, perform trend reporting, and analysis of performed corrective actions and safety implications.

Competent authorities shall perform checks similarly to those specified in the MDR, on in vitro diagnostic medical devices. In the case of unacceptable risk or non-complience, where the risk to health or safety of patient is deemed critical, devices and their accessories may be recalled or withdrawn from the market.

2.3.2 ENISA best practice

In the ENISA section for critical infrastructures and services, there are published documents on handling medical devices and their inherent cybersecurity [11]. These documents can be used as guidelines by manufacturers to manage their business and their devices in a secure manner. They have best practice documents for configuration and management to handle malicious actions, testing components and setting up a segregated network, implementing plans for vulnerability identifications and incident response, how to handle and encrypt sensitive data, how to enhance security controls in wireless communication, and much more. ENISA provides good practice guidelines to the manufacturers of medical devices, which is in accordance with best-practice and regulation.

2.3.3 Norwegian Security Act

The Norwegian Security Act went into force in 2019 and applies to the government, county, and municipal bodies as well as suppliers of good and services. It also applies to relations with other states and international companies, fundamental societal functions, and the basic security of the population [28].

The National Security Authority has the responsibility for protective security work and to supervise and ensure compliance to related security work. The NSM have the authority to perform basic criteria inspections, obtain and assess information on protective security work, develop and advise on security work, and they are granted unhindered access to critical national information systems, infrastructure, and information.

In response to a risk to national security interests or on infrastructure and services, the supervisory authority has the authority to access undertakings' information systems and infrastructure. They also have the authority to process personal data in accordance to their intended tasks as long as it does not pose a disproportionate interference with the right to privacy.

Chapter 4 in the Norwegian security act states that that the responsibility of protective security work rests upon the undertaking, and that the undertaking shall implement a management system for protective security work. Additionally, the undertaking should perform regular risk assessments. The undertakings are also obligated to implement protective security measures to ensure an appropriate level of security, to reduce the risk of associated threats to security. The undertaking is also obligated to document the identified threats and document their planned and implemented security measures.

The undertaking shall also give notice to the NSM immediately if they are affected by present threats to security, they have a reasonable suspicion of activites that have affected or may affect their or other undertakings, or of serious breaches of security requirements on national information, infrastructure, systems, communications, or personal security.

2.4 Hardware - Memory Extraction

In this section, we will explain the technical aspects of the devices and protocols used in the memory dumping procedure. Firstly, we will go through some the technical aspects of the microprocessor on the CardioMessenger Smart 3G, the STMicroelectronics STM32F417. Followed by a technical explanation of the two debugging protocols that were used in the dumping procedure, namely JTAG and SWD.

2.4.1 STMicroelectronics Microprocessor - STM32

The STM32F417 microprocessor is part of the STM32 32-bit microcontrollers based on the Arm Cortex-M processor, and is developed by STMicroelectronics [29]. It offers functionality such as high performance, signal processing, low-power operation, connectivity, and a fully integrated platform with tools and software online for ease of development. The STM32 is an industry-standard microcontroller that is used in anything from small projects to large end-to-end platforms.

The STM32F417 comes with an integrated cryptographic- and hashing processor that provides hardware acceleration for AES (128, 192, 256), DES (&TDES), and hashing by MD5 or SHA-1 [30].

The STM32 also has multiple built-in features for safety [31]. Some of the safety features of the STM32F4xx is a watchdog timer that can detect and recover from malfunctions, a hardware CRC unit to identify errors in digital data, a memory protection unit that manages the access of stack and memory in-use, and power supply monitoring.

2.4.2 JTAG & SWD

JTAG is an interface for debugging an embedded device or circuit board with a microprocessor [32]. JTAG stands for Joint Test Action Group and is an industry standard for testing manufactured circuit boards. JTAG defines a dedicated debug port with multiple connection pins implementing a serial communication with the circuit board.

The JTAG standard consists of five specific connector pins. Test Data In (TDI), Test Data Out (TDO), Test Clock (TCK), Test Mode Select (TMS), and the optional Test Reset (TRST). TDI is the pin feeding input data to the chip. JTAG does not define a protocol for the data stream on this channel and leaves it open for any sequence of 1's and 0's. It leaves the microprocessor to deal with the input. Manufacturers can however define protocols to run over the JTAG standard. TDO is specific for the data stream out of the chip. The TCK sets the speed of the Test Access Port controller, it is set from the outside device controlling the JTAG connection. TMS is controlled with different voltages to define different modes for JTAG. The TRST pin is optionally used when it is needed to reset the TAP controller to a correct state.

SWD, Serial Wire Debug, is an ARM alternative to JTAG with a 2-pin interface [32]. Hence, less wires are needed to connect onto an electrical circuit board. SWD is defined in ARM debug interface version 5 and uses a bi-directional protocol. This enables access to the chips system memory, peripherals and registers. The TMS and TCK pins in a standard JTAG connection are defined as SWDIO and SWCLK for a SWD connection.

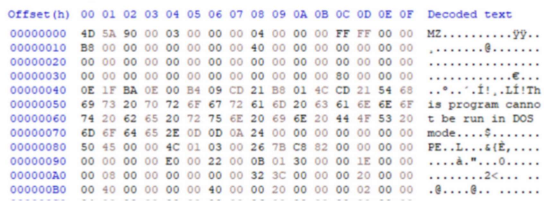
2.5 Software: Reverse Engineering

This section gives a technical overview of binary analysis, what assembly/disassembly and compile/decompile is in the context of reverse engineering, and it gives an overview of reverse engineering programs such as Binwalk and Ghidra.

2.5.1 Binary Analysis

In this subsection, we will go through technical details, techniques, and programs for our binary analysis process. As previously stated, our intention in this thesis is to reverse parts of the code running on the HMU. However, by dumping the memory of the HMU we do not get source code, we get binary files that represent parts of the executable code running on the device. The software running on the HMU were code in a high-level programming language, which was compiled, i.e converted into a lower-level code such as machine code, and turned into an executable binary file which is installed on the device. Our analysis will therefore not be on source code but rather a set of compiled binaries from the HMUs memory. This is a great distinction and the reason it makes the reverse engineering process complex. It is not always possible to fully reverse or convert the binary back to a similar state as the source code, the goal of the reverse engineering is therefore to understand what the code does, to the extent this is possible. To perform this task we need to use sophisticated open-source reverse engineering software.

A challenging aspect of binary analysis is the loss of information when the source code is compiled. An example of this is naming conventions used in high-level programming languages. This type of symbolic information is lost in the compilation process. Other examples of lost information after compilation can be variable types, abstractions and compartmentalization of functions, and classes. With these limitations, our reverse engineering software will decompile machine code in the binaries from the HMU memory and generate an obfuscated imitation of high-level code, to the best of its ability. Because of the loss of information and imperfect nature of the tools decompilation techniques, the result of a binary analysis may be imprecise, especially when we are limited by somewhat obfuscated decompiled code that cannot be executed during analysis - i.e static analysis.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ,.......@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ............€...
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  50 45 00 00 4C 01 03 00 26 7B C8 82 00 00 00 00  PE..L...&{È,....
00000090  00 00 00 00 E0 00 22 00 0B 01 30 00 00 1E 00 00  ....à."...0.....
000000A0  00 08 00 00 00 00 00 00 32 3C 00 00 00 20 00 00  ........2<... ..
000000B0  00 40 00 00 00 00 40 00 00 20 00 00 00 02 00 00  .@....@.. ......
```

Figure 2.1: Example of static analysis of .exe file

Figure 2.1 shows a typical example of a static binary analysis. Every .exe file has the identifying *MZ* ASCII string at the beginning, or *4D 5A* represented in hexadecimal notation. Being able to identify file types is important in an reverse engineering process and here are many tools available with this functionality. Two programs that are able to detect, even *hidden*, file types are Binwalk and Ghidra, and they are technically explained below and their use cases are shown under the Binary Analysis section in the Results chapter.

2.5.2 Binwalk

Binwalk is a tool used for analyzing, reverse engineering, and extracting data from firmware images [33]. It is used to search and find file signatures and code embedded within a binary file. Binwalk is able to detect these files based on its system of improved signatures that are commonly found in compressed and archived files, firmware headers, the Linux kernel, and file systems etc [34]. Binwalk is also equipped with several binary analysis tools such as an entropy scan.

2.5.3 Ghidra

Ghidra is an open-source reverse engineering framework developed by the National Security Agency (NSA) [35]. It is an open source alternative to the well-known SRE IDA by Hex-Rays. Ghidra was released at the RSA conference in March 2019, and it features such as disassembly tools, decompilation, debugging, platform support for Windows, MacOS and Linux, graphing and scripting. Ghidra also has a public and extensive API (Application Programming Interface) and supports script plugins written in Python and Java [36].

Among Ghidras many features, there is a decompiler and a disassembler. Which means that Ghidra is capable of loading and parsing machine code in a binary format and converting it back to assembly code or a higher-level readable code which will resemble the source code.

Ghidra also has a multiple different pre-installed auto analyzers. There is an ARM Analyzer that looks for 32-bit ARM instructions, an ASCII Strings Analyzer which searches for valid ASCII strings and defines them, a function signature analyzer, stack analyzers, an embedded image analyzer and a few more. Hence, with installed analyzers, script support and much more functionality, Ghidra well suited for most reverse engineering purposes [37].

Chapter 3
Methodology

3.1 Scientific Method

The scientific method describes the process of discerning facts from a set of information or data. The process is characterized by steps such as observation, experimentation, inductive and deductive reasoning, forming a hypothesis, and testing [38]. In order to apply the scientific method correctly, each test needs to be used on a falsifiable phenomenon. Which means the phenomenon needs to possess the capacity to be proven wrong. In the scientific method proving or disproving is equally valid, hence if a phenomenon lacks falsifiability, there can not be draw any conclusions from its tests.

Most methods used in any scientific field, whether they are related to technology, medicine, economics etc, all use similar steps as defined in the scientific method. Their commonality is the basis of obtaining knowledge based on proving or disproving a hypothesis. The scientific method forms the foundation of all scientific methodologies.

The Importance of Reproducibility

The scientific method suggests that every finding should be retested and reproduced in an attempt to be falsified. The scientific method does not blindly accept nor trust results that do not possess the ability to be falsified or have not been replicated.

The Standford Enclyclopedia of Philosophy defines reproducibility as the redoing of computations or whole experiments [39]. The ability to replicate processes and scientific findings is an essential part of any scientific method. Replication can confirm and validate findings, or disprove them. If a result drawn from a scientific process can not be replicated, it can not be stated as true nor assumed as valid. Failing to make a process and its results reproducible, undermines the credibility of the experiment and makes the derived results unreliable.

In 2016, Baker published an article about the *reproducibility crisis* [40]. In their study, they found that over 70% of 1500 researchers had tried and failed to reproduce at least one experiment from another scientist. Over 50% of the same researchers agreed that there is a reproducibility crisis, however 31% of the researchers did not think the results were wrongful even when they could not reproduce the published results. Baker also states that statistical data on how much of the scientific literature is limited, and that some studies from the fields of psychology and cancer biology show rates of 40% and 10% being reproducible.

Another example of the reproducibility problem was revealed by a study performed by the Center for Open Science when they released the findings of a eight year long project trying to replicate 193 experiments from 53 of the top papers published between 2010 and 2012 related to cancer [41]. The findings stated that only a quarter of the experiments were able to be reproduced. In addition to that, 50 of the experiments that were reproduced showed an effect size 85 percent lower than reported in the original experiments.

The reproducibility issue highlights the need to thoroughly document a scientific process. Even though the examples above is related to psychology and cancer research, the same responsibility of reproducibility applies to our project. Documenting the processes in this thesis will be a priority to ensure reproducibility. This thesis builds on the works of others and reproducing their work is also essential to conduct this thesis as well as to verify our findings.

3.2 Threat Model

The Open Web Application Security Project (OWASP) defines Threat Model as a method to *identify, communicate, and understand threats and mitigations within the context of protecting something of value* [42]. It is a structured representation of an application and its information affecting its security. Threat modelling can be used on systems, applications, IoT, networks, business processes and much more.

A threat model usually consists of a detailed description of the asset, potential threats to the asset, a set of actions to mitigate the defined threats, and an assessment to validate the threat models and verify the success of mitigating actions. The threat model is not limited to these steps and may also include the motivation or who the adversary might be.

By organizing and analyzing all the information about the asset, the purpose of threat modelling is to improve the security of the asset. Threat modelling is a continuous process which should be reevaluated throughout the assets life-cycle. It is also crucial to detect security flaws early, hence threat modelling should be implemented from the planning phase of asset development.

When a threat model is applied to the Pacemaker Ecosystem the obvious assets are patient information and their devices. An attacker may attempt to exploit inherent insecure functionality for either financial gain or other malicious reasons. These days all kinds of personal information is sold on the Internet. Multiple high profile individuals such as a former United States president, other well-known politicians, a former pope, famous athletes, and actors have been fitted with pacemakers through the years [43]. This kind of patient information is highly sensitive and can be taken advantage of, especially in the hands of an adversary with malicious intent.

In this thesis we are examining the various interactions an active attacker can perform with the HMU and its software. Mainly by having physical access to a HMU initially and being able to connect to it. There are different ways to interact with an HMU. The HMU has its physical interfaces on the circuit board and a wireless communication interface to the patients pacemaker. Connecting to the debugging ports on the HMU circuit board is an example of physical interaction. Wireless eavesdropping, and modifying messages being sent to or from the HMU are examples of non-physical interactions.

Some of the devices in the Pacemaker Ecosystem have a higher likelihood of being targeted by attackers, such as the HMU. Of all the devices and assets in the ecosystem, except the HMU, it is a rarity to find devices for purchase. Programmers are a special device only located in hospitals or treatment facilities, and they cost a small fortune. Used pacemakers are naturally not accessible because of their location within the body, hence they are not commonly sold anywhere. HMUs however, are not difficult to find sold on online auction websites such as eBay. In this sense, the HMU is the *lowest hanging fruit*.

Risk Assessment Criteria

A specific set of terminology is used when performing a risk assessment in a security context. Terminology such as confidentiality, integrity and availability, the CIA triad, are used in every security assessment. Additionally, there are a few more terms which can be used when necessary. These terms are used to describe the asset and its inherent state. They are also used to evaluate the severity of a finding, specifically in the context of security, privacy, and safety of the patients in the pacemaker ecosystem. The following terminology is defined by the Cybersecurity and Infrastructure Security Agency (NICCS) [44].

Confidentiality is the property of only disclosing information to authorized users, devices, and processes. In the context of the pacemaker ecosystem, the confidentiality property avoids adversaries to access patients devices and obtain information which can have severe consequences when exposed.

Integrity is the property where information or an entity has not been modified or changed without authorization. Modified or destroyed devices could pose serious health risks for a patient. In a worst case scenario, a damaged pacemaker could be lethal for the patient.

Availability is defined as the property of being usable when needed or queried. Medical devices need a high level of availability. Especially the IMDs need to have a high availability since they are the implanted device monitoring and maintaining the patients heart rhythm. If the IMD were unavailable while the patient is experiencing a type of incident, this could also be severe or lethal for the patient. Whereas the data servers or HMU can be offline occasionally, the IMD cannot be afforded the same availability standard.

Authentication is the property or the ability to verify the identity of a user, process or device.

Privacy is an assurance property that defines the access to information about a protected asset.

Safety is defined as the condition of being free from harm, risk of injury, loss, and danger.

Authenticity is defined as the property that data originated from the alleged source. It means that the item in question has the property to be verified and can be trusted.

Non-repudiation is the property to prove the origin and integrity of data. It is an assurance of the validity of an item which is hard to deny.

Authorization is the property where a right or access is given to an approved or privileged user to a system or on of its resources. If an unauthorized user is granted access to a system it is considered compromised.

Now that we have defined the different risk assessment criteria, we can model threats for the HMU in the pacemaker ecosystem. To identify threats we are using the STRIDE model. STRIDE is an acronym for Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of privilege [5].

STRIDE		
Threat	**Desired Property**	**Attack on HMU**
Spoofing	Authenticity	Establishing a connection to the HMU by one of its interfaces, physical or wireless, to gain access or authenticate while masquerading as a legitimate actor. By either hijacking the ongoing communication session or performing a Cross-Site Request Forgery (CSRF).
Tampering	Integrity	Modifying the memory or firmware of the HMU to alter its functionality
Repudiation	Non-Repudiation	Craft customized messages sent to the HMU from a device that imitates the pacemaker or the data servers, by exploiting weak authentication/authenticity, lack of signatures, or lack of event logs.
Information disclosure	Confidentiality	Setting up an illegitimate base station to eavesdrop on communication from the HMU, imitating as a legitimate base station
Denial of Service	Availability	Halting the HMUs microprocessor by sending a large amount of instructions such that the microprocessor becomes unable to perform its tasks
Elevation of Privilege	Authorization	Gaining access to the private data servers by obtaining valid user credentials from the HMU

Table 3.1: A STRIDE threat model of the HMU [5]

To assess these threats we are using the DREAD system developed by Microsoft [6]. DREAD is an acronym for Damage, Reproducibility, Exploitability, Affected users, and Discoverability. It is an assessment system that gives a numeric score to each identified threat. The score is linearly correlated to the severity of the threat. A

higher score equals a more severe threat. Each property is given a score between 0 and 10. The threat is assessed by the impact on the different properties. Damage potential's score is based on how much damage that will be caused if the threat occurred. 0 indicated no damage and 10 equals total destruction or a compromised system. The reproducibility score is based on how easy the threat exploit is to reproduce. A score of 0 in reproducibility means that the exploit is near impossible to reproduce, and a score of 10 means that only a simple tool is needed to perform the exploit [45]. The exploitability score is based on what resources are needed to exploit the threat, a higher score is given if the amount of needed resources are low. If the exploit requires advanced programming skills or knowledge the score is 0. The affected users score is based on how many users that would be affected by the threat exploit, it ranges from no users with a 0 score to all users and a score of 10. The discoverability score is based on how easy it is to discover this threat. A score of 0 indicates that it is near impossible to discover the threat and that special access or source code is necessary. A score of 10 means that the threat is discoverable by simple tools or very limited knowledge.
The table 3.2 below gives a DREAD assessment example of an attack based on the integrity property. We chose this example because it is closely related to knowledge of the internal structure of the HMU, one of our main objectives in this thesis.

DREAD	
Example: Integrity - modifying the memory of the HMU	
Damage Potential	2-10
Reproducibility	0-6
Exploitability	6
Affected users	1
Discoverability	0-2

Table 3.2: A DREAD threat assessment of the HMU [6]

The damage potential of modifying the memory on the HMU can be quite severe. The damage of the exploit depends on the section of memory being modified. The HMU has memory sections that are accessible and other sections that are not. However if an important system memory section is modified, the threat exploit can cause malfunctioning of the device in various ways. The types of malfunctions range from a device resets, restarts or crash due to illegal state handling, forcing a downgrade on a security functionality such as encryption, or disabling an interface or peripheral on the microcontroller. In those cases the damage score range from 1-5, where a reset is a 1 and a downgrade attack can be a 5. In a worst case scenario, the HMU can be flashed, i.e overwritten, with a modified firmware which can attack the IMD over their ULP-AMI interface. Attacks on this interface, by RF activation, have previously been shown to successfully perform replay attacks and battery drainage attacks on four brands of IMDs [19]. The damage potential could be critical for the patient. The HMU could in this scenario act as a Man-in-the-Middle and control the most sensitive parts of the pacemaker ecosystem. If we score this threat by the modified firmware scenario, it is a solid 10.

Reproducing a memory modification on a HMU device requires an HMU device which is easily available online, it requires a few simple lab tools to connect to the PCB, and it requires an individual with knowledge of embedded security. Since this exploit needs different steps and someone with a specific skill-set, 6 might be a reasonable score. In the worst case example of the HMU attacking the IMD, the score would be near impossible even for a skilled individual and deserves a score of 0.

The exploitability of the threat is somewhere in the middle of the scale. The threat requires some tools and programming abilities. However there are a lot of information available online with guides to connect and query a microprocessor, and by extension modify memory. A reasonable score might be 6.

Our threat exploit is performed on a single HMU and therefore only affects one patient. Hence, we give the affected users score 1. Discovering the possibility of modifying the memory of a HMU unit, and also discovering that the HMU is susceptible to the threat would require a highly skilled individual. The adversary would need to be either a security researcher or someone with a similar technical capacity. Hence, it would be very hard and rated at a score of 2. In the example of modifying the firmware to attack the patient's IMD, the score would be near closer to 0. To calculate the threat ranking we use the DREAD formula.

$$AverageThreatRanking = (D + R + E + A + D)/5$$

The best case of the threat is calculated with the lowest ratings for each property. It is the scenario of modifying the memory to the extent that the firmware is replaced and used to attack a patients IMD.

$$ModifiedFirmwareThreatRanking = (2 + 0 + 6 + 1 + 0)/5 = 1.8$$

The worst case of the threat is calculated with the highest ratings for each property. This is the scenario of modifying the memory to cause some kind of malfunction.

$$MalfunctionThreatRanking = (10 + 6 + 6 + 1 + 2)/5 = 5$$

The output from the DREAD threat assessment formula gives a numerical value that can be used to prioritize threat cases. In our case the threat ranking for malfunction was 5 and the threat ranking for modified firmware was 1.8. In a general sense the threat of malfunction by modification should be the first priority. The scale for the formula goes from 0 to 10, and the malfunction threat scores at the middle of the scale. If the manufacturer has an implemented risk identification and assessment process, the malfunction by memory modification threat should be an obvious threat to mitigate. The manufacturer has to prioritize threats according to their threat assessment scores. Depending on the context of the devices, some manufacturers need to handle threats with lower DREAD scores than other manufacturers. In the context of medical devices, the standard for threat assessment scores that needs to be mitigated should be lower than the standard used for other devices. Thus, the firmware modification score of 1.8 should also be addressed by the vendor. Since these devices pose a threat to the health of a patient, it require the manufacturers to handle most threats with low scores if they additionally pose large health or safety impacts for the patient.

To mitigate these threats it is beneficial for the manufacturer to also apply certain cybersecurity mitigation strategies. There exists different frameworks or strategies developed by security authorities. Two such companies or authorities are Microsoft and the NSA. They have their own defined list of top ten mitigation strategies to reduce impact of commonly identified threats [46]. Their strategies are quite similar and relate to the mostly the same categories of cybersecurity. In figure 3.3 we can see the categories defined by the NSA and how they can be handled [7].

NSA'S Top Ten Cybersecurity Mitigation Strategies	
Update & upgrade software	Applying software updates by an automated process to avoid zero-day/n-day exploits from threat actors.
Defend Privileges and Accounts	Access control and management of credentials should be automated by a access privileged access management system. Implement procedures to securely reset credentials such as passwords, tokens etc.
Enforce Signed Software Execution Policies	There should be a list of trusted certificates and an enforced software policy for signed firmware, drivers, and other executables. To maintain integrity devices should use secure boot options and whitelisting of applications.
Exercise a System Recovery Plan	To protect critical operational data, configurations, and logs a comprehensive recovery strategy should be implemented. There should be encrypted backups and support for recovery of systems and devices. A recovery plan can be a necessary mitigation plan against malicious attacks such as ransomware or other integrity attacks.
Actively Manage Systems and Configurations	Actively manage devices, applications, applications interacting on the network. Systems need to adapt to a dynamic threat environment. Unknown, unnecessary or unexpected devices should be removed from the network.
Continuously Hunt for Network Intrusions	Proactive measures to detect, contain, and remove malicious devices or services in the network. Both passive and active forms of detection mechanisms should be deployed.
Leverage Modern Hardware Security Features	Implement hardware security such as secure boot, TPM, application containment, schedule replacement of old hardware/devices, and hardware virtualization.
Segregate Networks Using Application-Aware Defenses	Segregation of critical networks and services. Implement IDS and application-aware defences in the network to block malicious traffic.
Integrate Threat Reputation Services	Implement multi-source reputation and information-sharing services to prevent and detect malicious attacks/events.
Transition to Multi-Factor Authentication	Transition away from single-factor authentication systems such as password-based systems, and replace with physical token-based systems supplemented with knowledge-based systems such as passwords or PINs. Authentication of both devices are also critical, i.e mutual authentication.

Table 3.3: Cybersecurity mitigation strategies by NSA [7]

3.3 Black Box Testing

Black Box Testing is defined as *"a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software"* [47].

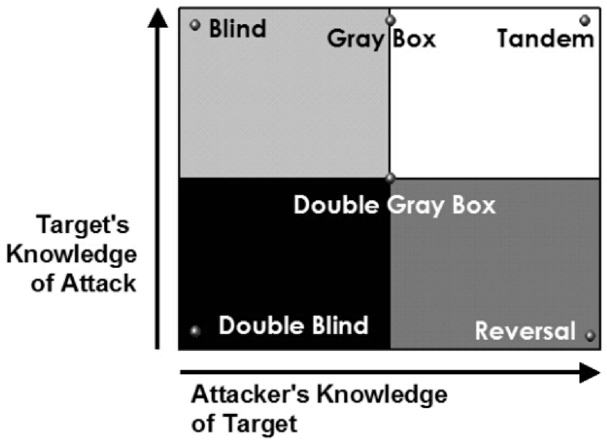


Figure 3.1: *Common test types* defined by OSSTMM [1]

Finding a suitable method is dependant on the available information. OSSTMM defines six different types of common test types based on the target and the information available to the attacker [1]. The available information, or knowledge, of the attacker is plotted along the x-axis and the targets knowledge is plotted along the y-axis. The figure can then be divided into four squares, where the top right square defines a *white box test* and the bottom left corner defines a *double blind test.* White box is a test where both the analyst and target are aware of the audit, and where the assets and all knowledge is available for both parties. This can be the case if the analyst is hired by a company. In that scenario, the analyst is the "attacker".

In the opposite end of the scale, there is the *Double Blind* test. In the double blind test, the attacker has no prior knowledge of the targets system, assets or defenses. This audit can be performed by both an internal analyst or a real attacker, depending on the context. The purpose of an analyst to do a double blind audit is to simulate the scenario of a real attacker. Similarly, in this project we will be the attacker - an *ethical* attacker.

Our role in reverse engineering the HMU devices will be similar to the attacker with limited to no knowledge. The only prior knowledge we have of the HMUs internal system is findings from other similar projects also working on the devices in the pacemaker ecosystem.

Black Box Testing is a methodology for testing software without access or knowledge of the internal system. The method consists of modifying input data, observing behavior and then analyzing the corresponding output data. Based on this, we can try to find the internal functionality of the system based on input and output.

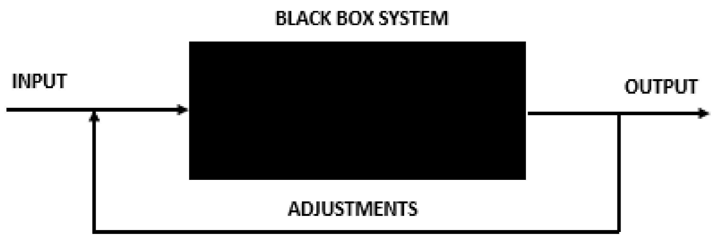


Figure 3.2: An illustration of the Black Box Testing Method

Black Box Testing method stages

Below are the stages stages of the black box testing method [47].

1. **Information gathering**
   Initially, we try to obtain all the available information about the system. All the specifications, interfaces and technical information in the components datasheet or manual, or even readable on the device itself. All these details about the system is documented and used to make an outline of the device. This is used to form a basis and hypothesis for our following steps.

2. **Hypotheses**
   With the information from the previous step, we construct hypotheses about how the system will behave based on our current knowledge.

3. **Test Cases**
   We make positive and negative test cases, predicting the expected outputs of the system for each of our constructed supposed valid or invalid inputs. We base these test cases on each of the hypotheses defined in the previous step.

4. **Execution**
   The test cases are executed, where the constructed inputs from each test case are inserted into the system in order.

5. **Comparison and analysis**
   For each input we compare the observed output with our hypothesized expected output. If the observed output is coinciding with the predicted output, we have confirm functionality and obtained knowledge of the system. The goal is to learn more information about the systems internal behavior through a series of constructed inputs and their corresponding outputs- a trial-and-error approach based on the information available. If the output does not match our expected test case hypothesis, we have falsified our hypotheses, and can form a new hypothesis based on the system output. Outputs can occasionally give unintended internal knowledge, e.g debugging strings, which can give clues as to functionality.

6. **Retest**
   The information obtained during the previous iterations are added to our known information, and then used to construct a new series of hypotheses in the next iteration with new test cases. It is an iterative process that examines constructed inputs based on hypotheses and compares to actual system output.

Each generated input and resulting system output will impact the next iteration of inputs. My reverse engineering process will therefore be a series of *trial-and-error* iterations.

The benefit of black box testing is that fact that we are analyzing a real system. Its output, whether it is output corresponding to valid input or error messages, describes the internal behavior and might give us information we otherwise would not know. Except from error messages, there may be debugging strings, configuration information or even as severe as internal system information. The drawback of black box testing is the lack of internal knowledge. White Box Testing is the opposite, it is defined by its access to system internals and source code. Ideally, we would like to do white box testing but the medical device companies do not give away their internal source code. Their medical device source code is highly sensitive and may be taken advantage of in the hands of an adversary with malicious intent.

Since our pacemaker project's centre of attention is Biotronik's proprietary pacemaker ecosystem, our knowledge is limited to previous research on HMU devices and vaguely-worded device user manuals.

Finally, findings that show critical vulnerabilities are reported to the appropriate authority in a coordinated vulnerability disclosure process. Along with the findings it can be useful to report mitigating measures. The test cases may uncover unintended functionality or insecure practises, hence reporting means of mitigating risk and exposed systems are useful for the authority and the by extension the people affected by the service.

3.4 Limitations

The first limitation of this reverse engineering process is the lack of source code. As described in the Software: Reverse Engineering section, a lot of useful information is lost in the compilation process. Our project is therefore limited to the effect of the available tools, and their functionality to reverse and decompile code from the binary memory dumps.

The analysis tools may not me accurate at reversing the memory dumps because certain information is lost from the compilation process, the software makes an educated guess and optimizations when analyzing the files. However, given enough time and the required skills and knowledge, it is feasible to reverse parts of the code to a stage where we are able to determine the structural code design and functionality.

One of the largest limitations is my own knowledge and experience. My experience going into this project rest on the general knowledge gained as a student of cyber security, and the specific knowledge of an electrical computer engineering course during my studies. The process of reverse engineering in this project might therefore be challenging and even more time-consuming based on the lack of specific training in this method of research.

3.5 Ethical & Legal Considerations

When starting a project on medical devices, it is important to recognize the implications of our work and findings. These medical devices are a life-necessity for many people around the globe. Therefore, it is important that we disclose critical findings in a safe manner to the proper authorities and agencies before the paper is disclosed publicly. This is done with a coordinated vulnerability disclosure (CVD) process with the manufacturer. They will then have the opportunity to mitigate any of the disclosed critical findings and vulnerabilities. We have also signed a non-disclosure agreement with SINTEF.

There is a possibility of finding personal information on medical devices. However, in the case of HMUs, previous work on the CardioMessenger Smart 3G and CardioMessenger II-S have shown no sign of personal information like the information found on pacemakers. However, if we were to find personal information such as identification numbers, phone numbers or such, it will be redacted from publication because of privacy and security considerations.

Chapter 4 Results

This chapter contains a preliminary information gathering phase about the CardioMessenger Smart 3G devices available in the SINTEF lab, by identifying the components and debugging interfaces on the boards, and then dumping certain memory locations.

We then perform a series of security analysis tests. We perform a preliminary strings analysis of the memory files. Section 4.2 gives an overview of the binary dumping procedure. A detailed walk-through is added to the Appendices A. We use Binwalk to perform an entropy analysis to show how data is stored within the files. Lastly, the flash memory file is analyzed in the SRE framework Ghidra to reverse engineer the code sections that make up the communication protocol from the HMU to the data servers.

4.1 Information Gathering

In gathering information about the HMU and the pacemaker ecosystem we are limited by publicly available information. It would be unethical and even illegal for us to utilize different information gathering tools that impose on devices and networks owned by a private company. Scanning or trying to establish a connection to their internal network can be classified as an attack on their property, hence we need conduct our research cautiously and within the bounds legal and ethical considerations.

Limited documentation of the HMU is available online. The manufacturer Biotronik has a technical manual for the CardioMessenger Smart 3G [9]. It contains information on the basic functionality of the HMU. How to setup the HMU, some basic maintenance and handling, and some limited telemetry- and technical data. The manual mentions the micro USB interface for charging, the error and self-test functions that are displayed on the HMU display, and information on the frequency bands for the different HMU model's telecommunication services and short range

MICS.

Previous research on the pacemaker ecosystem and specifically the HMUs have also uncovered additional information. Previous Master's Theses from the Pacemaker Project at SINTEF have performed hardware analysis on most of the different HMU models from Biotroniks, and some models from other vendors. The hardware components on the CardioMessenger Smart 3G's PCB have been identified by Bour and Kok&Stenersen [23, 48]. The HMU has a microprocessor from STMicroelectronics called STM32F417, a micro-USB interface, a 2MB SRAM EM7164SP from Jeju semiconductor, a LC4064ZE programmable logic device from Lattice semiconductor, a Telit HE910-D modem, and a GD25Q32C 4MB external flashfrom Giga Device.

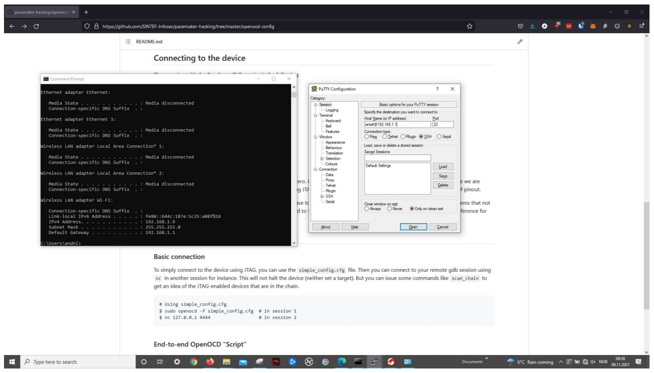
The STM32F417 microprocessor has publically available documents - the reference manual and the datasheet [4, 30]. These documents define all the technical details of the microprocessor, its internal components and detailed description of its functionality. Information in the reference manual and datasheet can be used to identify the microprocessor's internal components, their supported features, and their defined procedures. These features and procedure sets the basis of features the HMU is capable of. For example the hashing processor of the STM32F4xx supports the secure hash algorithm (SHA-1, SHA-224, SHA-256) and the message-digest algorithm (MD5). Hence, we would expect that one or more version of these hashing algorithms are used in the firmware. Similarly, the same can be said for the cryptographic processor. The STM32F4xx cryptoprocessors support DES, TDES, and AES encryption standards. we would expect to find at least one of these encryption standards implemented in the firmware.

4.2 Reproducing HMU Binary Dumps

Before we can start analyzing the binary memory files from the HMU we need to reproduce the memory dumping procedure on the HMU. With the assistance of my supervisor and the equipment at the SINTEF lab, we were able to dump the memory images from the HMU. The memory dumping have been performed by my supervisor on older HMU models and on one of the CardioMessenger Smart 3G devices. They used the JTAG interface to connect to the HMU's PCB. In our case, we first reproduced the dumping over the JTAG interface, and then we additionally performed the dumping procedure over the SWD interface. We modified the JTAG scripts to support SWD and connected to the microprocessor. We were able to reproduce the JTAG dumping of the CardioMessenger Smart 3G, as well as reproducing the same memory dumping through the SWD debugging interface. We performed the dumping procedure on the three CardioMessenger Smart 3G HMUs that are available in the SINTEF lab.

Finding 1. *SWD debugging interface is available*

The laptop is connected to a local wireless private network, an access point (AP) running on the Raspberry Pi Zero with authentication, using the SSH protocol through the PuTTY application on my Windows laptop. An image of the connection setup in PuTTY can be seen below.



https://github.com/SINTEF-Infosec/pacemaker-hacking/tree/master/openocd-config

README.md

Connecting to the device

```
Ethernet adapter Ethernet:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Ethernet adapter Ethernet 3:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Local Area Connection* 1:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Local Area Connection* 2:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::644c:187e:5c35:a08f%16
   IPv4 Address. . . . . . . . . . . : 192.168.1.9
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1

C:\Users\andni>
```

PuTTY Configuration

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): sintef@192.168.1.1 Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Saved Sessions: Default Settings

Close window on exit: Always Never [✓ Only on clean exit]

Basic connection

To simply connect to the device using JTAG, you can use the `simple_config.cfg` file. Then you can connect to your remote gdb session using `nc` in another session for instance. This will not halt the device (neither set a target). But you can issue some commands like `scan_chain` to get an idea of the JTAG-enabled devices that are in the chain.

```
# Using simple_config.cfg
$ sudo openocd -f simple_config.cfg  # in session 1
$ nc 127.0.0.1 4444                  # in session 2
```

End-to-end OpenOCD "Script"

Figure 4.1: Establishing an SSH connection through PuTTY

We connected the laptop to the wireless AP on the Raspberry Pi Zero. The raspberryPi is then connected with wires through either the JTAG or SWD debugging interface. This enabled us to remotely execute commands on the CardioMessenger Smart 3G's STM32F417 microprocessor from the laptop.

Executing the scripts for JTAG or SWD in A.4 dumps the memory from the HMU. The memory dumps goes through the selected debugging interface, to the Raspberry Pi Zero and wirelessly to our laptop over the SSH session in PuTTy by running the following command:

```
# ssh from windows laptop to linux raspberry pi zero:
pscp.exe -r sintef@192.168.1.1:/home/sintef/tools/hardware-hacking/3gTesting
/first3gdumps
```

A detailed description of process, the lab setup, and the connection between the HMU's PCB and raspberryPi to our laptop is included in Appendix A, Binary Extraction.

4.3 Binary Analysis

In this section we start analyzing the binary memory files extracted from the Biotronik CardioMessenger Smart 3G HMU device. Firstly, we conducted a string and entropy analysis of each file in order to gather information. This will help us to see what kind of information is available initially from the strings in the files, and it sets expectations from which we can form hypotheses. To accomplish these tasks we will be using the built-in *strings* command in Linux and the functionalities of Binwalk. The following section in this chapter is related to reverse engineering in Ghidra, and the tools used and developed in that process.

4.3.1 String Analysis

Firstly, we wanted to check whether there were any interesting information available straight from the memory files. So we boot up a virtual machine with Ubuntu Linux and run the strings command on each of the memory files we extracted from the HMU. In the boot configuration section of the STM32 reference manual, we know that the boot space is available in the flash section of memory [30]. The boot configuration also mentions that the embedded SRAM code area might be used as a boot space as well. Hence, we start by searching for text strings in those files. A full overview of the memory section in the STM32 microcontroller is added to appendix, A.6 [4].

```
1200
AT+CPIN="8[REDACTED]8"
bio[REDACTED]mobile
internet
 64[REDACTED]29@cmsmart-homemonitoring.de
 Z1[REDACTED]bM
 172.[REDACTED].14.1
 2323
```

These findings are also similar to those of Lie, Kok&Markussen, and Bour [22, 23, 48]. They also found usernames, passwords (i.e credentials), pin codes and network structure stored in cleartext in the SRAM and flash, on both older CardioMessenger HMUs and of the same Smart 3G model. Additionally, the memory gave away information about Biotroniks internal network structure. The IP address starting with 172 is defined to be used in a local network. 2323 would likely be related to the port number the service is communicating on. The same finding was true for each of the three CM Smart 3G HMUs we have at our disposal at SINTEF. They all had their own unique username and password stored in the memory. The credentials of the other two devices can be seen below in redacted form for security considerations.

```
64[REDACTED]03@cmsmart-homemonitoring.de
D*[REDACTED]ON

64[REDACTED]74@cmsmart-homemonitoring.de
*n[REDACTED]rR
```

Most of the extracted memory files do not contain any debugging strings. However, in addition to the SRAM, the flash memory file contains a large amount of debugging strings. Strings in the form of error messages that reveal naming conventions and functionality of the code inside the HMU. Below is a selection of the most interesting strings that were found.

Finding 2. *Credentials (i.e username and password), pin code, and infrastructure information is unencrypted in SRAM*

Firstly, we ran the strings command on the flash file and found a lot of debugging strings. These strings had certain terms such as compression, copyright, header, layer, encryption, data, hash, download, source, GSM etc. Running the strings command with the grep option enables us find all the strings that contain each of the selected terms.

```
Command: $ strings flash.img | grep Copyright
deflate 1.2.1 Copyright 1995-2003 Jean-loup Gailly
inflate 1.2.1 Copyright 1995-2003 Mark Adler
incorrect header check
unknown compression method
```

One of the first searches we performed was to check whether the binary had any strings that referenced to third party software. In the extracted text above we found a (de)compression reference for inflate/deflate which is part of the Zlib library written by Jean-loup Gailly and Mark Adler [49]. The string *1.2.1* is a reference to the version of the Zlib library. This version contains two known CVEs, CVE-2005-2096 and CVE-2004-0797 [50]. We also found the *incorrect header check* and *unknown compression* strings next to the copyright strings when looking through the flash file.

These strings show that the code has a defined set of allowed compression methods, and that there likely is a header check to define the selected method of compression. The CVEs we identified have the potential to cause denial of service (DoS), i.e a crash. CVE-2004-0797 contains an issue with the error handling in the inflate and inflateBack functions which allow local users to cause a crash by denial of service. CVE-2005-2096 shows that a remote attacker can cause a DoS attack with a crafted stream with an incomplete code description with length greater than 1, leading to a buffer overflow. In the CVE description, a crash could be demonstrated by a modified PNG file [50]. However, at this point it is unknown whether any of these CVEs would affect the HMU because the restrictions on the compressed payload in the communication protocol is unknown. We also do not know the full extent of how data is processed by the HMU. A proof of concept is therefore required to verify if the HMU is vulnerable to the CVEs.

Finding 3. *The firmware contains the third-party compression library zlib version 1.2.1, which has two known CVEs*

```
Command: $ strings flash.img | grep Source
$Source: src/variables.c $ $Revision: 1.1 $
$Source: src/variables.h $ $Revision: 1.2 $
$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: UlpamiLow1.c $ $Revision: 1.41 $
$Source: UlpamiLow.h $ $Revision: 1.13 $
$Source: UlpamiLow_H3.h $ $Revision: 1.2 $
$Source: UlpamiLowInterface.h $ $Revision: 1.21 $
$Source: src/UlpamiLow_nonsh_debug.h $ $Revision: 1.42 $
$Source: src/UlpamiLow_nonsh_Hw.h $ $Revision: 1.39 $
$Source: UlpamiLowHAL.c $ $Revision: 1.6 $
$Source: UlpamiLowHAL.h $ $Revision: 1.2 $
$Source: bfs.c $ $Revision: 1.37 $
$Source: bfs.h $ $Revision: 1.14 $
```

Searching for strings containing *Source* showed that they reveal the directory, file names, and their revision number. Although we do not have the files, we can see the revision number of the files, their names, and a *src* folder structure. We also notice that many of strings mention the name *ULPAMI*, which is an acronym for Ultra Low Power Active Medical Implant - an IMD with a medical implant communication system (MICS).

Finding 4. *Revision numbers of the ULPAMI scripts are available in Flash memory*

```
Command: $ strings flash.img | grep UsbInterface
Message_UsbInterfaceHandler: configuration not found
Message_UsbInterfaceHandler: unable to open virtual pdhm port
Message_UsbInterfaceHandler: unsupported interface
Message_Init: unable to init Message_UsbInterfaceHandler
pGpGMessage_UsbInterfaceHandler: com port open event received and debug is disabled
Message_UsbInterfaceHandler: unable to request rx and/or tx buffer from port
Message_UsbInterfaceHandler: com port open event received and debug is disabled
Message_UsbInterfaceHandler: com port close event received - closing virtual pdhm port
```

The HMU has a micro USB interface that is used for charging [9]. When searching through the flash memory for USB and UsbInterface we find several debugging strings. We find mentions of port, interface, and PDHM. PDHM is an acronym or name we find in a multitude of strings related to network and communication. Hence, we believe it is likely a name related to Biotroniks proprietary communication protocol. Finding a reference to PDHM related to USB was unexpected. Maybe the USB interface has more functionality than the mere charging capabilities that were described in the HMU manual.

```
Command: $ strings flash.img | grep Hmsc
Message_HmscInterfaceHandler: unable to initialize semaphore
Message_HmscInterfaceHandler: unable to set rx/tx buffer
Message_Init: unable to init Message_HmscInterfaceHandler
Message_HmscInterfaceHandler: unable to open virtual hmsc sms pdhm port
Message_HmscInterfaceHandler: triggered by regular check timeout
Message_HmscInterfaceHandler: unable to create gsm state handler process
Message_HmscInterfaceHandler: unable to start gsm modem e.g. due to
                            low battery voltage, waiting for usb plug/unplug
Message_HmscInterfaceHandler: unable to export out message list
Message_HmscInterfaceHandler: sms fallback timeout occured while waiting
                            for disconnect event
Message_HmscInterfaceHandler: unable to open virtual hmsc pdhm port
Message_HmscInterfaceHandler: sms fallback timeout occured while waiting
                            for connect event
Message_HmscInterfaceHandler: unable to change in/out direction
Message_HmscInterfaceHandler: unable to set sms connection wanted event
```

Hmsc is an acronym for Host Mass Storage Class and it is a driver for USB. When searching for strings containing Hmsc we find debugging strings that mention opening of virtual ports, PDHM, and SMS. The strings also mention that the GSM modem is dependent on sufficient battery voltage to start, and that the SMS interface is waiting for an event to perform connection and disconnection to the other device.

```
Command: $ strings flash.img | grep Layer
PDHM: ProcessDataFromMessageLayer - unknown layer [%d] error
PDHM: pdhm_decode_rx_handler - transport layer expected
PDHM: pdhm_network_rx_handler: communication timeout during network layer selection
```

Since we are looking for a communication protocol we expect to find structures similar to the OSI model and its defined layers. Searching for the term "layer" we found a few interesting strings. The first line refers to ProcessDataFromMessageLayer and clearly throws an error if the layer is not set correctly, likely the message layer. It means that the program expects a specific value to define the layer, otherwise an error is thrown. The other two strings mention the transport layer and the network layer, in PDHM__DECODE__RX__HANDLER and PDHM__NETWORK__RX__HANDLER respectively. It is clear that the software has specific functionality to check if the transport-, network- and message layer is defined correctly, and that a layered structure similar to OSI is used.

```
Command: $ strings flash.img | grep pdhm_network
PDHM: pdhm_network_rx_handler - malloc error
PDHM: pdhm_network_rx_handler: communication timeout
PDHM: pdhm_network_tx_handler: communication timeout
pdhm_open: unable to initialize pdhm_network_rx_handler
PDHM: pdhm_network_rx_handler: communication timeout during network layer selection
pdhm_open: unable to initialize pdhm_network/decode_tx_handler
```

```
Command: $ strings flash.img | grep auth
PDHM: ProcessDataFromMessageLayer - MSG auth CRC error
PDHM: pdhm_cmd_auth: response
PDHM: pdhm_cmd_auth - error: adding message to out list
PDHM: pdhm_cmd_auth - error: invalid passkey
PDHM: pdhm_message_handler - pdhm_cmd_auth: invalid frame len
PDHM: pdhm_message_handler - pdhm_cmd_auth: error during authentification
PDHM: ProcessDataFromMessageLayer - MSG auth CRC error
authentication failure
GSM_HandlePdpContext: authentication failure or wrong apn
```

Searching for the term auth, as in authentication, we find interesting strings containing CRC, passkey, authentication, and APN. In the first line there is an error about CRC authentication on data that is processed in the message layer, and that there is an authentication response being either logged or sent. There is also an error message if a passkey is invalid which can be interesting to analyze later in the reverse engineering process. It might be the same password we identified from the SRAM strings or possibly another credential. The last line above mentions an APN which is the gateway name a telecommunications network and a computer network. An APN is made up of a network identifier and a operator identifier. It might be referring to the APN we found previously from the SRAM string analysis.

Finding 5. *The HMU has functionality authenticate and detect a wrong APN*

```
Command: $ strings flash.img | grep ProcessData
PDHM: ProcessDataFromMessageLayer - unknown layer [%d] error
PDHM: ProcessDataFromMessageLayer - MSG Hash error
PDHM: ProcessDataFromMessageLayer - frame len [%d/%d] error
PDHM: ProcessDataFromMessageLayer - invalid direction
PDHM: ProcessDataFromMessageLayer - unspecific error
PDHM: ProcessDataFromMessageLayer - pdhm alloc error
PDHM: ProcessDataFromMessageLayer - MSG auth CRC error
PDHM: ProcessDataFromMessageLayer - invalid container crc
PDHM: ProcessDataFromMessageLayer - invalid target [%d] error
PDHM: ProcessDataFromMessageLayer - invalid source [%d] error
PDHM: ProcessDataFromMessageLayer - invalid session id in frame
PDHM: ProcessDataFromMessageLayer - invalid sequence number in frame
```

From the previous string findings we saw the term *ProcessData*, searching for this term might be useful to uncover more functionality on how data is handled in the code. Searching for ProcessData gave many different error messages, which can be seen above. There is another CRC error check in the container in the data from message layer, there are checks on the validity of source and target, and mentions of session, frame, and sequence numbers. Frame is the unit for the data link layer. However, the session id in frame does not relate properly to the session layer in the OSI model. The session layer is multiple layers above the data link layer for frames, therefore session id might be a local identifier unrelated to the OSI model.

Finding 6. *The HMU has functionality to detect an invalid target and source*

Finding 7. *The HMU has functionality to calculate a message hash and CRC*

```
Command: $ strings flash.img | grep Download
PDHM: pdhm_message_handler - pdhm_cmd_download_file: complete
PDHM: pdhm_message_handler - pdhm_cmd_download_file - rx: %x/%x
PDHM: pdhm_message_handler - pdhm_cmd_download_file: invalid offset detected
PDHM: pdhm_message_handler - pdhm_cmd_download_file: invalid file id detected
PDHM: pdhm_message_handler - pdhm_cmd_download_file: parsing error - response too short
```

Since the HMU has a network connection, we suspect that it has functionality to download firmware updates. Searching the flash file for *Download* reveals that the HMU indeed does have functionality to download a file. The strings also show a function named PDHM_MESSAGE_HANDLER, a file id check, and offset check. The strings do not indicate the file type being downloaded, but we assume it is related to either a firmware- or configuration file.

Finding 8. *The HMU has functionality to download a file*

```
Command: $ strings flash.img | grep Install
Message_InstallFile: pdhm_file_list_processing_failed
Message_InstallFile: invalid signature.
Message_InstallFile: invalid magic number.
```

The strings also reveal that the HMU installs a file - likely the downloaded file. It also shows that the file has a signature check and a magic number check.

Finding 9. *The HMU has functionality to install a file*

4.3.2 Entropy Analysis in Binwalk

We already know that the flash memory section is supposed to contain code and the string findings from the previous section show error messages related to various functionality. Thus, we want to analyze the contents of the flash file and see how the data is distributed within. For this purpose an entropy analysis is suitable. Entropy measures the *uncertainty* or *randomness* of data. In our case, the uncertainty is related to the change of data or information within a file. Entropy show where data is located within a file. For this purpose, Binwalk can help us perform an entropy analysis with a text and visual output as seen below in figure 4.2. The entropy of all the memory files is attached in appendix E, Entropy of Memory Files.

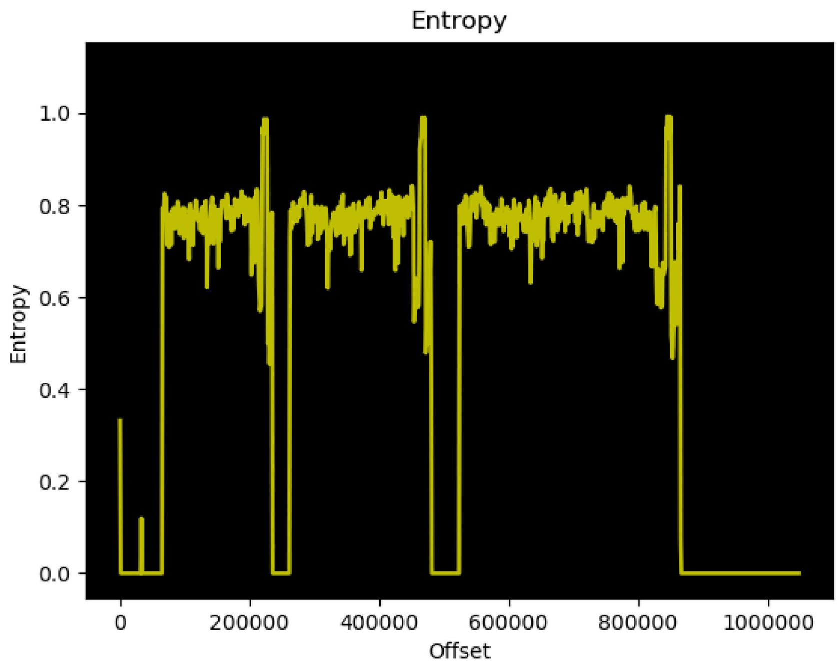
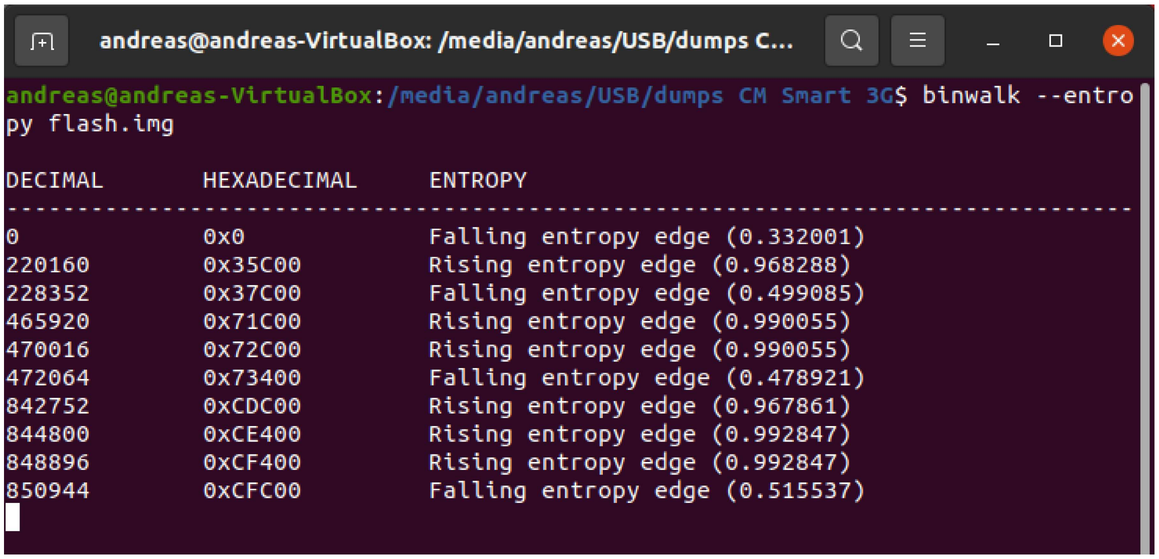


Figure 4.2: Entropy of the flash memory file



```
andreas@andreas-VirtualBox:/media/andreas/USB/dumps CM Smart 3G$ binwalk --entro
py flash.img

DECIMAL       HEXADECIMAL     ENTROPY
--------------------------------------------------------------------------------
0             0x0             Falling entropy edge (0.332001)
220160        0x35C00         Rising entropy edge (0.968288)
228352        0x37C00         Falling entropy edge (0.499085)
465920        0x71C00         Rising entropy edge (0.990055)
470016        0x72C00         Rising entropy edge (0.990055)
472064        0x73400         Falling entropy edge (0.478921)
842752        0xCDC00         Rising entropy edge (0.967861)
844800        0xCE400         Rising entropy edge (0.992847)
848896        0xCF400         Rising entropy edge (0.992847)
850944        0xCFC00         Falling entropy edge (0.515537)
```

Figure 4.3: Entropy of flash with detailed hexadecimal information

As shown in figure 4.2 above, we notice that the flash memory file contains three larger blocks of code. Aside form the size difference, each of the blocks have similar peaks and troughs in the entropy graph. Another key similarity is the large spike at the end of each code block. These similarities implicates that these blocks have a similar structure.

4.4 Ghidra Analysis

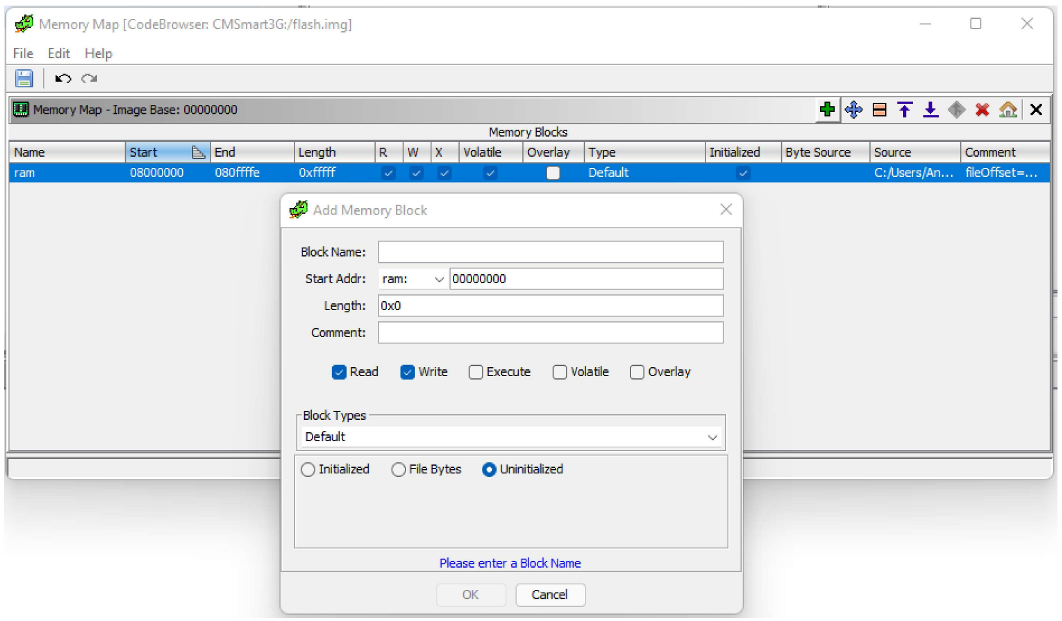
In this section we advance to the reverse engineering process in Ghidra. The prerequisite for this section is adding the binary into Ghidra, which is explained in Ghidra Installation & Setup in appendix C. This section describes the analysis tools that are used in Ghidra, modification of the binary input file, the development of necessary tools & scripts, the discovery of third party libraries, the exploration of a hierarchical code structure, and lastly we identify the layers and fields of the PDHM communication protocol.

4.4.1 Defining peripherals from base addresses

One of the first issues we experienced in Ghidra was the lack of readability. Ghidra does not automatically recognize the microprocessor from which the binary was obtained from, and therefore Ghidra does not know the names of the peripherals and registers the base addresses refer to. Most sections of code were therefore only referenced by hexadecimal base address or a DAT identifier - Ghidras functionality to specify data content of some unknown kind. Our first task was then to identify and define each of the peripherals and their registers names.

Script: Reading .svd file & defining peripherals

To make the code more readable, we decided to make a script plugin for Ghidra that defines memory sections by name accordingly to the peripherals and their registers based on how they are defined in the STM32 reference manual [30]. However, doing this process manually is time-consuming and not very practical. Below in figure 4.4 is an image of how this task were performed for the binary flash file, named ram in the image, and the process would be exactly the same for defining peripherals.


Memory Map [CodeBrowser: CMSmart3G:/flash.img]
File Edit Help
Memory Map - Image Base: 00000000
Memory Blocks
Add Memory Block
OK
Cancel

Name	Start	End	Length	R	W	X	Volatile	Overlay	Type	Initialized	Byte Source	Source	Comment
ram	08000000	080ffffe	0xfffff	✓	✓	✓	✓	☐	Default	✓		C:/Users/An...	fileOffset=...

Block Name:
Start Addr: ram: 00000000
Length: 0x0
Comment:

[✓ Read] [✓ Write] ☐ Execute ☐ Volatile ☐ Overlay

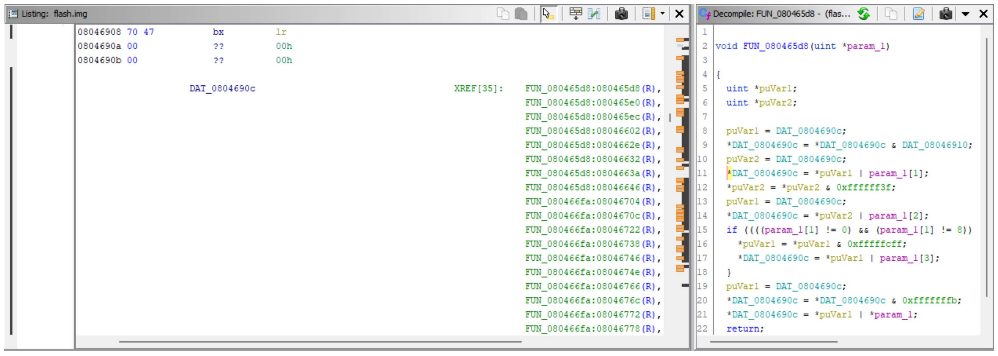
Block Types: Default

○ Initialized ○ File Bytes [✓ Uninitialized]

Please enter a Block Name

Figure 4.4: Manually adding memory blocks

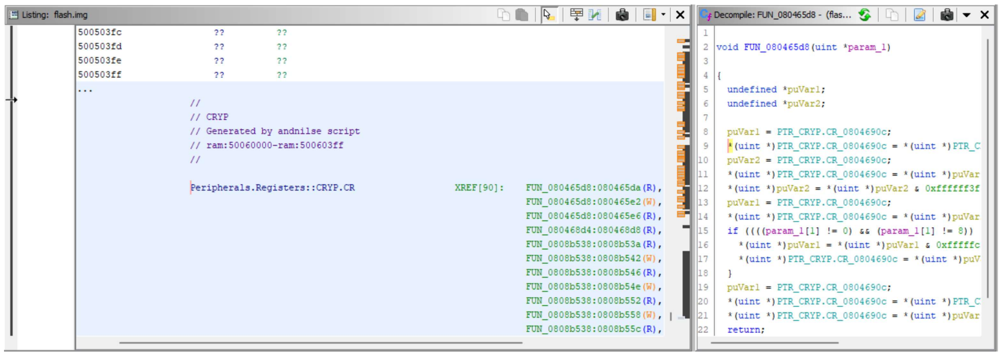
To make the reverse engineering process efficient, we decided to develop a plugin for Ghidra that would define all the peripherals and registers for our STM32 microprocessor automatically. Ghidra has an open API and there already exist a couple different tools for this exact purpose, the *svd-loader* by leveldownsecurity and the *stm32f4_loader* by Bour [51, 52]. The svd-loader relies on .svd (System View Description) files that contain a microprocessors defined peripherals and registers in an xml file structure. Instead of manually going through the STM32 reference manual, there exists a library of .svd files for common microprocessors online by Arm Kiel [53]. However, the leveldown security svd-loader only loaded some of the peripherals before throwing error messages. The stm32f4-loader were hardcoded for every peripheral and register for the STM32F4 microprocessor. For this reason we decided to develop our own version of the svd-loader that reads the .svd file similarly to leveldown securitys loader, but also with the register components like we found in Bours stm32f4-loader. Our Ghidra .svd loader were developed to work for any ARM processor with a given .svd file, and possess the ability to define both peripherals with all their register names in a Ghidra project. The code for our Ghidra .svd loader is published on the SINTEF github and is attached in the appendix D, Developed Tools & Scripts for Ghidra. The result of running our Ghidra .svd loader can be seen in the two figures below, 4.5 and 4.6.



```
08046908 70 47        bx        lr
0804690a 00           ??        00h
0804690b 00           ??        00h

               DAT_0804690c                        XREF[35]:   FUN_080465d8:080465d8(R),
                                                               FUN_080465d8:080465e0(R),
                                                               FUN_080465d8:080465ec(R),
                                                               FUN_080465d8:08046602(R),
                                                               FUN_080465d8:0804662e(R),
                                                               FUN_080465d8:08046632(R),
                                                               FUN_080465d8:0804663a(R),
                                                               FUN_080465d8:08046646(R),
                                                               FUN_080466fa:08046704(R),
                                                               FUN_080466fa:0804670c(R),
                                                               FUN_080466fa:08046722(R),
                                                               FUN_080466fa:08046738(R),
                                                               FUN_080466fa:08046746(R),
                                                               FUN_080466fa:0804674e(R),
                                                               FUN_080466fa:08046766(R),
                                                               FUN_080466fa:0804676c(R),
                                                               FUN_080466fa:08046772(R),
                                                               FUN_080466fa:08046778(R),
```

```c

void FUN_080465d8(uint *param_1)

{
  uint *puVar1;
  uint *puVar2;

  puVar1 = DAT_0804690c;
  *DAT_0804690c = *DAT_0804690c & DAT_08046910;
  puVar2 = DAT_0804690c;
  *DAT_0804690c = *puVar1 | param_1[1];
  *puVar2 = *puVar2 & 0xffffff3f;
  puVar1 = DAT_0804690c;
  *DAT_0804690c = *puVar2 | param_1[2];
  if ((((param_1[1] != 0) && (param_1[1] != 8))
    *puVar1 = *puVar1 & 0xfffffcff;
    *DAT_0804690c = *puVar1 | param_1[3];
  }
  puVar1 = DAT_0804690c;
  *DAT_0804690c = *DAT_0804690c & 0xfffffffb;
  *DAT_0804690c = *puVar1 | *param_1;
  return;
```

Figure 4.5: Pre .svd loader



```
500503fc              ??        ??
500503fd              ??        ??
500503fe              ??        ??
500503ff              ??        ??
...
               //
               // CRYP
               // Generated by andnilse script
               // ram:50060000-ram:500603ff
               //

               Peripherals.Registers::CRYP.CR      XREF[90]:   FUN_080465d8:080465da(R),
                                                               FUN_080465d8:080465e2(W),
                                                               FUN_080465d8:080465e6(R),
                                                               FUN_080468d4:080468d8(R),
                                                               FUN_0808b538:0808b53a(R),
                                                               FUN_0808b538:0808b542(W),
                                                               FUN_0808b538:0808b546(R),
                                                               FUN_0808b538:0808b54e(W),
                                                               FUN_0808b538:0808b552(R),
                                                               FUN_0808b538:0808b558(W),
                                                               FUN_0808b538:0808b55c(R),
```

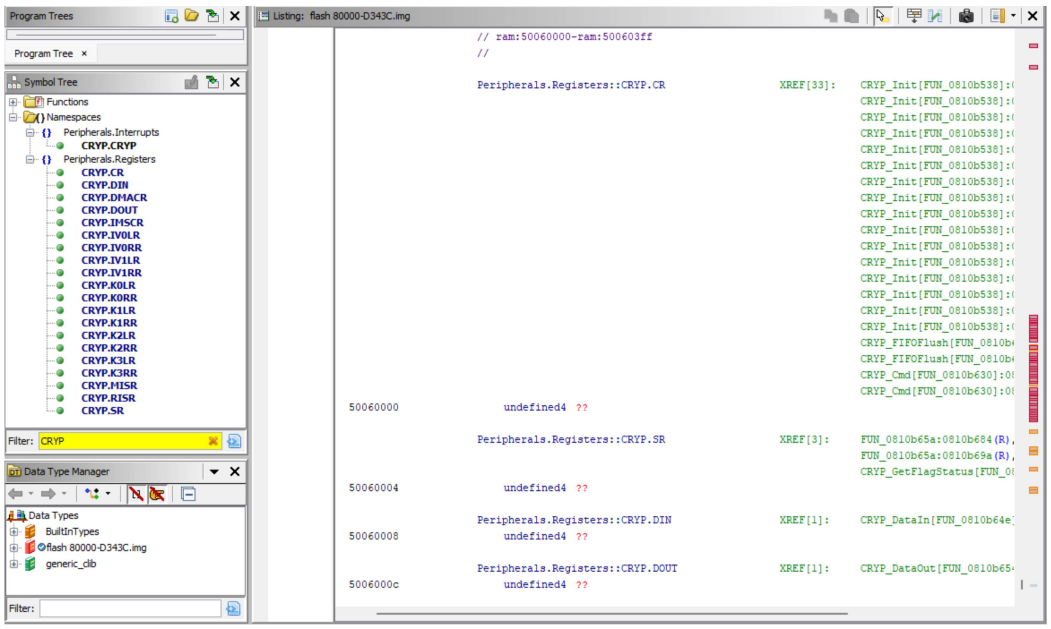
```c

void FUN_080465d8(uint *param_1)

{
  undefined *puVar1;
  undefined *puVar2;

  puVar1 = PTR_CRYP.CR_0804690c;
  *(uint *)PTR_CRYP.CR_0804690c = *(uint *)PTR_C
  puVar2 = PTR_CRYP.CR_0804690c;
  *(uint *)PTR_CRYP.CR_0804690c = *(uint *)puVar
  *(uint *)puVar2 = *(uint *)puVar2 & 0xffffff3f
  puVar1 = PTR_CRYP.CR_0804690c;
  *(uint *)PTR_CRYP.CR_0804690c = *(uint *)puVar
  if ((((param_1[1] != 0) && (param_1[1] != 8))
    *(uint *)puVar1 = *(uint *)puVar1 & 0xffffc
    *(uint *)PTR_CRYP.CR_0804690c = *(uint *)puV
  }
  puVar1 = PTR_CRYP.CR_0804690c;
  *(uint *)PTR_CRYP.CR_0804690c = *(uint *)PTR_C
  *(uint *)PTR_CRYP.CR_0804690c = *(uint *)puVar
  return;
```

Figure 4.6: Post .svd loader

From figure 4.6, our .svd loader enables us to see which peripherals and registers are being referenced in each of the decompiled functions in Ghidra. Instead of manually looking up every hexadecimal base address in the reference manual or from the Arm Kiel .svd files, we are now able to see the peripheral and register name directly from the generated functions in Ghidra. In the case of the function from figure 4.6, we can see that the function contains pointers to the cryptographic processor and its Control Register (CR).

4.4.2 Identifying the low-level Cryptographic Functions

Since one of our main goals is to find the functions that constitute the HMUs communications protocol, and reversing it, we need to start identifying the low-level cryptographic functions. These functions contain a reference to the at least one of the many registers on the cryptographic processor - named CRYP in the STM32 reference manual. We know from previous work on these HMUs that the communications from the HMU is encrypted [48, 54]. So, at some point in the communication protocol there has to be a reference to encryption, and the encryption function needs to reference onto the lowest level cryptographic functions on the cryptographic processor. Identifying the low-level cryptographic functions is therefore our first primary objective and it will enable us to do a bottom-up approach to identify all the functions that are related to the communications protocol. In figure 4.7 below, we can see all the functions that reference the cryptographic control register that was defined by our own svd-loader script.



```
Functions
Namespaces
    Peripherals.Interrupts
        CRYP.CRYP
    Peripherals.Registers
        CRYP.CR
        CRYP.DIN
        CRYP.DMACR
        CRYP.DOUT
        CRYP.IMSCR
        CRYP.IV0LR
        CRYP.IV0RR
        CRYP.IV1LR
        CRYP.IV1RR
        CRYP.K0LR
        CRYP.K0RR
        CRYP.K1LR
        CRYP.K1RR
        CRYP.K2LR
        CRYP.K2RR
        CRYP.K3LR
        CRYP.K3RR
        CRYP.MISR
        CRYP.RISR
        CRYP.SR
Filter: CRYP

Data Types
    BuiltInTypes
    flash 80000-D343C.img
    generic_clib
Filter:
```

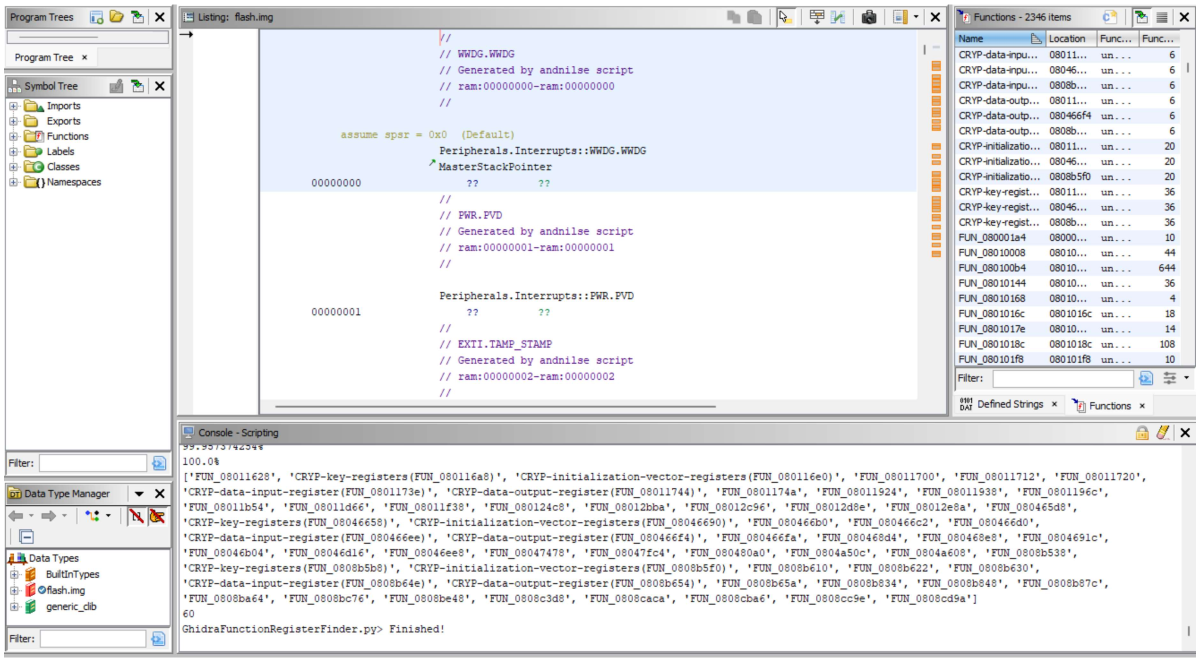
```
                    // ram:50060000-ram:500603ff
                    //

                    Peripherals.Registers::CRYP.CR          XREF[33]:    CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_Init[FUN_0810b538]:
                                                                         CRYP_FIFOFlush[FUN_0810b
                                                                         CRYP_FIFOFlush[FUN_0810b
                                                                         CRYP_Cmd[FUN_0810b630]:0
                                                                         CRYP_Cmd[FUN_0810b630]:0
50060000                undefined4  ??

                    Peripherals.Registers::CRYP.SR          XREF[3]:     FUN_0810b65a:0810b684(R),
                                                                         FUN_0810b65a:0810b69a(R),
                                                                         CRYP_GetFlagStatus[FUN_0
50060004                undefined4  ??

                    Peripherals.Registers::CRYP.DIN         XREF[1]:     CRYP_DataIn[FUN_0810b64e]
50060008                undefined4  ??

                    Peripherals.Registers::CRYP.DOUT        XREF[1]:     CRYP_DataOut[FUN_0810b65
5006000c                undefined4  ??
```

Figure 4.7: Functions calling on the cryptographic processor

However, this method is limited to finding functions for a single register at the time. If we were to look for functions containing multiple registers simultaneously this method would not be feasible. That is why we developed another script plugin in Ghidra that could look for multiple registers or strings simultaneously.

Script: Compiling & searching functions

Being able to find functions that contain more multiple registers or strings simultaneously is beneficial to identify certain specific functions. Also, it would be practical for the coming reverse engineering work, where we redefine other types of data inside a function such as variables, to fully reverse a function. Therefore we made another script plugin for Ghidra that goes through every recognized function and searches for all the keywords in a list - one or more. For every function containing every item, register or string from the list, the script would generate a file with all the matching functions and print their names in the Ghidra console. In figure 4.8, we can see the output from the script plugin after searching for the *CRYP* peripheral registers.



```
                    //
                    // WWDG.WWDG
                    // Generated by andnilse script
                    // ram:00000000-ram:00000000
                    //

      assume spsr = 0x0  (Default)
                    Peripherals.Interrupts::WWDG.WWDG
                    MasterStackPointer
    00000000            ??          ??
                    //
                    // PWR.PVD
                    // Generated by andnilse script
                    // ram:00000001-ram:00000001
                    //

                    Peripherals.Interrupts::PWR.PVD
    00000001            ??          ??
                    //
                    // EXTI.TAMP_STAMP
                    // Generated by andnilse script
                    // ram:00000002-ram:00000002
                    //
```

Name	Location	Func...	Func...
CRYP-data-inpu...	08011...	un...	6
CRYP-data-inpu...	08046...	un...	6
CRYP-data-inpu...	0808b...	un...	6
CRYP-data-outp...	08011...	un...	6
CRYP-data-outp...	080466f4	un...	6
CRYP-data-outp...	0808b...	un...	6
CRYP-initializatio...	08011...	un...	20
CRYP-initializatio...	08046...	un...	20
CRYP-initializatio...	0808b5f0	un...	20
CRYP-key-regist...	08011...	un...	36
CRYP-key-regist...	08046...	un...	36
CRYP-key-regist...	0808b...	un...	36
FUN_080001a4	08000...	un...	10
FUN_08010008	08010...	un...	44
FUN_080100b4	08010...	un...	644
FUN_08010144	08010...	un...	36
FUN_08010168	08010...	un...	4
FUN_0801016c	0801016c	un...	18
FUN_0801017e	08010...	un...	14
FUN_0801018c	0801018c	un...	108
FUN_080101f8	080101f8	un...	10

```
99.9573742548
100.0%
['FUN_08011628', 'CRYP-key-registers(FUN_080116a8)', 'CRYP-initialization-vector-registers(FUN_080116e0)', 'FUN_08011700', 'FUN_08011712', 'FUN_08011720',
'CRYP-data-input-register(FUN_0801173e)', 'CRYP-data-output-register(FUN_08011744)', 'FUN_0801174a', 'FUN_08011924', 'FUN_08011938', 'FUN_0801196c',
'FUN_08011b54', 'FUN_08011d66', 'FUN_08011f38', 'FUN_080124c8', 'FUN_08012bba', 'FUN_08012c96', 'FUN_08012d8e', 'FUN_08012e8a', 'FUN_080465d8',
'CRYP-key-registers(FUN_08046658)', 'CRYP-initialization-vector-registers(FUN_08046690)', 'FUN_080466b0', 'FUN_080466c2', 'FUN_080466d0',
'CRYP-data-input-register(FUN_080466ee)', 'CRYP-data-output-register(FUN_080466f4)', 'FUN_080466fa', 'FUN_080468d4', 'FUN_080468e8', 'FUN_0804691c',
'FUN_08046b04', 'FUN_08046d16', 'FUN_08046ee8', 'FUN_08047478', 'FUN_08047fc4', 'FUN_080480a0', 'FUN_0804a50c', 'FUN_0804a608', 'FUN_0808b538',
'CRYP-key-registers(FUN_0808b5b8)', 'CRYP-initialization-vector-registers(FUN_0808b5f0)', 'FUN_0808b610', 'FUN_0808b622', 'FUN_0808b630',
'CRYP-data-input-register(FUN_0808b64e)', 'CRYP-data-output-register(FUN_0808b654)', 'FUN_0808b65a', 'FUN_0808b834', 'FUN_0808b848', 'FUN_0808b87c',
'FUN_0808ba64', 'FUN_0808bc76', 'FUN_0808be48', 'FUN_0808c3d8', 'FUN_0808caca', 'FUN_0808cba6', 'FUN_0808cc9e', 'FUN_0808cd9a']
60
GhidraFunctionRegisterFinder.py> Finished!
```

Figure 4.8: Output in console from GhidraFunctionFinder.py

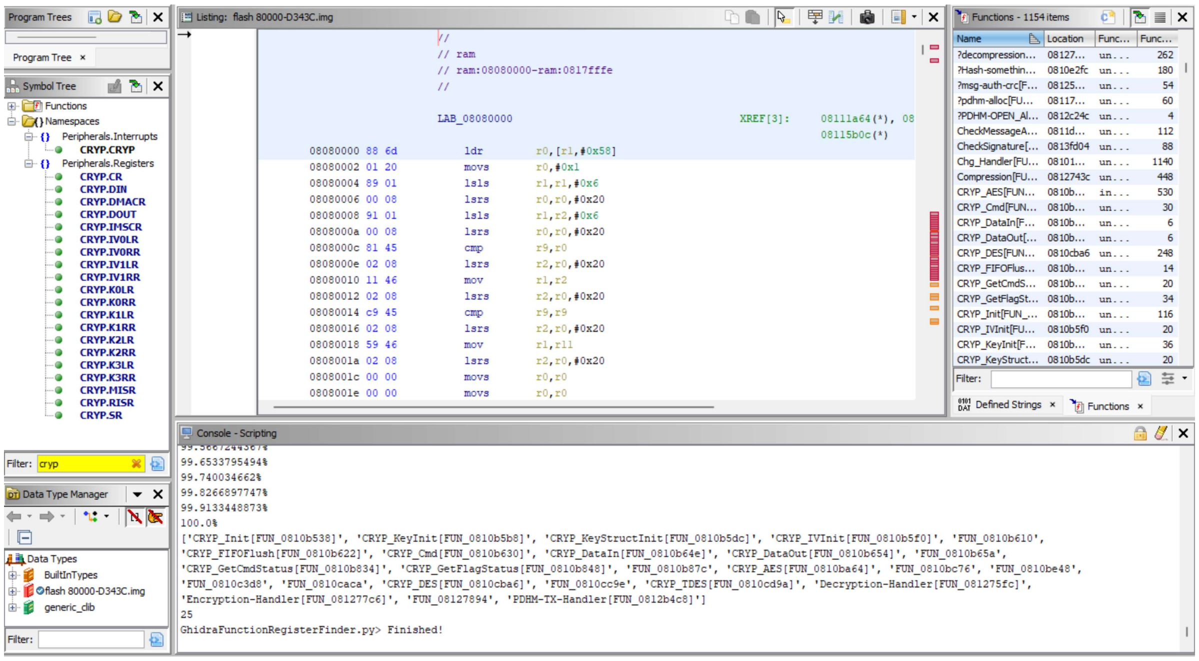
The output shows a list of 60 functions that match and contain one of the registers from the cryptographic processor. We had already mapped a couple of the functions before running this script and an interesting finding was that there appears to be 3 identical functions at different base addresses. For instance, there are three functions for *CRYP-data-input-register*, and they are all referenced at different base addresses. This is interesting because previously, in the Entropy Analysis in Binwalk section, we found that there were three blocks of code in the flash memory file, and we also found that some debugging strings were repeated.

The low-level functions calling on a register of the cryptographic processor always contained a hexadecimal number in the decompiled functions. An example of this is shown in a figure 4.10, where there is a pointer to the control register of the cryptographic processor and an offset *0x8*. When we look up the cryptographic processors control register in the reference manual for STM32, section 23.6.4 in the reference manual, we find that the offset 0x8 is the defined offset for CRYP data input register [30]. With this method we are able to name the functions based on its containing registers and offsets, and comparing those to the information in the microprocessor's reference manual. This procedure is performed for all the functions that were found by our GhidraFunctionFinder.py script. Identifying and renaming all these low-level functions will aid us in identifying other functions further on that all depend on these lower-level cryptographic functions.

Duplicated functions

Because of the issue of duplicated functions we decided to cut out and analyze one of the code blocks from the flash memory file. Our hypothesis is that the largest code block contains the same information found in the two smaller code blocks, as well as some additional data. Our analysis showed that the largest code block contained all the strings found in the two smaller preceding code blocks, and additional debugging strings that were not found in the former blocks. Thus, we are less likely to miss out on any data by only analyzing the larger code block, and we avoid overlapping work.

In figure 4.9 below, we have inserted the cut code block from *80000-D343C*-hex into Ghidra and ran the GhidraFunctionFinder.py script plugin again. We have also identified most of the functions referencing "CRYP".



```
                    //
                    // ram
                    // ram:08080000-ram:0817fffe
                    //

                    LAB_08080000                               XREF[3]:     08111a64(*), 08
                                                                            08115b0c(*)
08080000 88 6d          ldr        r0,[r1,#0x58]
08080002 01 20          movs       r0,#0x1
08080004 89 01          lsls       r1,r1,#0x6
08080006 00 08          lsrs       r0,r0,#0x20
08080008 91 01          lsls       r1,r2,#0x6
0808000a 00 08          lsrs       r0,r0,#0x20
0808000c 81 45          cmp        r9,r0
0808000e 02 08          lsrs       r2,r0,#0x20
08080010 11 46          mov        r1,r2
08080012 02 08          lsrs       r2,r0,#0x20
08080014 c9 45          cmp        r9,r9
08080016 02 08          lsrs       r2,r0,#0x20
08080018 59 46          mov        r1,r11
0808001a 02 08          lsrs       r2,r0,#0x20
0808001c 00 00          movs       r0,r0
0808001e 00 00          movs       r0,r0
```

Name	Location	Func...	Func...
?decompression...	08127...	un...	262
?Hash-somethin...	0810e2fc	un...	180
?msg-auth-crc[F...	08125...	un...	54
?pdhm-alloc[FU...	08117...	un...	60
?PDHM-OPEN_Al...	0812c24c	un...	4
CheckMessageA...	0811d...	un...	112
CheckSignature[...	0813fd04	un...	88
Chg_Handler[FU...	08101...	un...	1140
Compression[FU...	0812743c	un...	448
CRYP_AES[FUN...	0810b...	in...	530
CRYP_Cmd[FUN...	0810b...	un...	30
CRYP_DataIn[F...	0810b...	un...	6
CRYP_DataOut[...	0810b...	un...	6
CRYP_DES[FUN...	0810cba6	un...	248
CRYP_FIFOFlus...	0810b...	un...	14
CRYP_GetCmdS...	0810b...	un...	20
CRYP_GetFlagSt...	0810b...	un...	34
CRYP_Init[FUN_...	0810b...	un...	116
CRYP_IVInit[FU...	0810b5f0	un...	20
CRYP_KeyInit[F...	0810b...	un...	36
CRYP_KeyStruct...	0810b5dc	un...	20

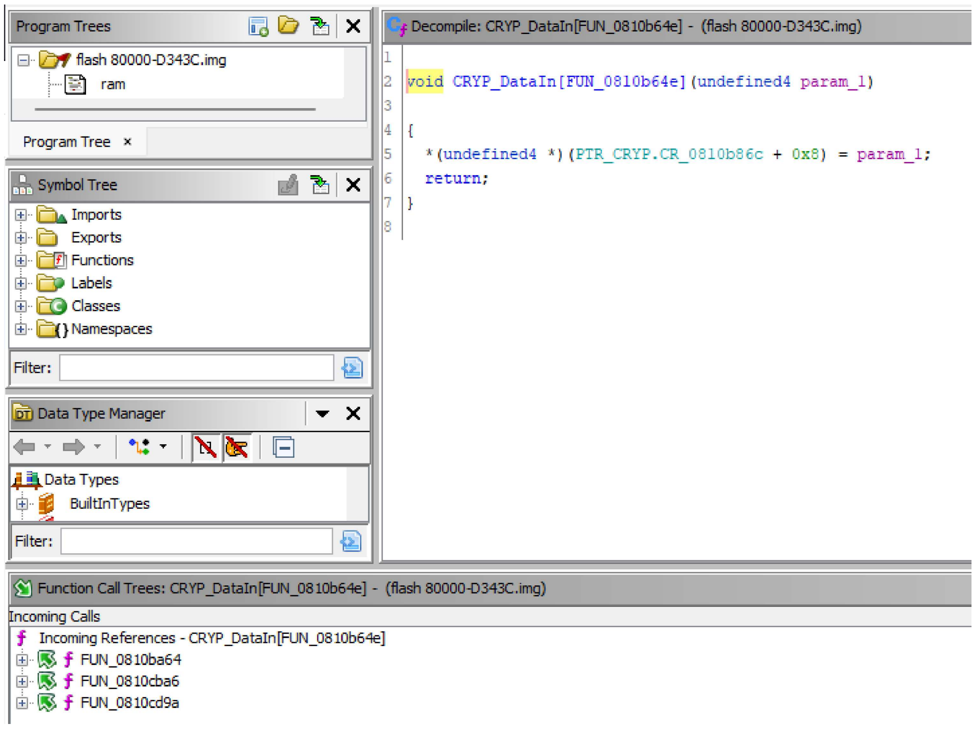
```
99.6533795494%
99.740034662%
99.8266897747%
99.9133448873%
100.0%
['CRYP_Init[FUN_0810b538]', 'CRYP_KeyInit[FUN_0810b5b8]', 'CRYP_KeyStructInit[FUN_0810b5dc]', 'CRYP_IVInit[FUN_0810b5f0]', 'FUN_0810b610',
'CRYP_FIFOFlush[FUN_0810b622]', 'CRYP_Cmd[FUN_0810b630]', 'CRYP_DataIn[FUN_0810b64e]', 'CRYP_DataOut[FUN_0810b654]', 'FUN_0810b65a',
'CRYP_GetCmdStatus[FUN_0810b834]', 'CRYP_GetFlagStatus[FUN_0810b848]', 'FUN_0810b87c', 'CRYP_AES[FUN_0810ba64]', 'FUN_0810bc76', 'FUN_0810be48',
'FUN_0810c3d8', 'FUN_0810caca', 'CRYP_DES[FUN_0810cba6]', 'FUN_0810cc9e', 'CRYP_TDES[FUN_0810cd9a]', 'Decryption-Handler[FUN_081275fc]',
'Encryption-Handler[FUN_081277c6]', 'FUN_08127894', 'PDHM-TX-Handler[FUN_0812b4c8]']
25
GhidraFunctionRegisterFinder.py> Finished!
```

Figure 4.9: Output in console from GhidraFunctionFinder.py on cut code block

When running the GhidraFunctionFinder on the largest flash code block we only got 25 matching functions containing a CRYP register. All of the previous duplications of decompiled functions were also not found in the results of the latest script execution. Cutting the binary seemed to remove the duplications of CRYP functions.

4.4.3 Identifying the Encryption Functions & Third-party Library

By using the *Function Call Trees* feature in Ghidra, we are able to see which functions that make call to the currently selected function. By selecting any of the low-level identified cryptographic functions, e.g CRYP_DataIn, we can see that it is called by three different functions as seen below in figure 4.10.



```
void CRYP_DataIn[FUN_0810b64e](undefined4 param_1)

{
  *(undefined4 *)(PTR_CRYP.CR_0810b86c + 0x8) = param_1;
  return;
}
```

```
Incoming References - CRYP_DataIn[FUN_0810b64e]
  FUN_0810ba64
  FUN_0810cba6
  FUN_0810cd9a
```

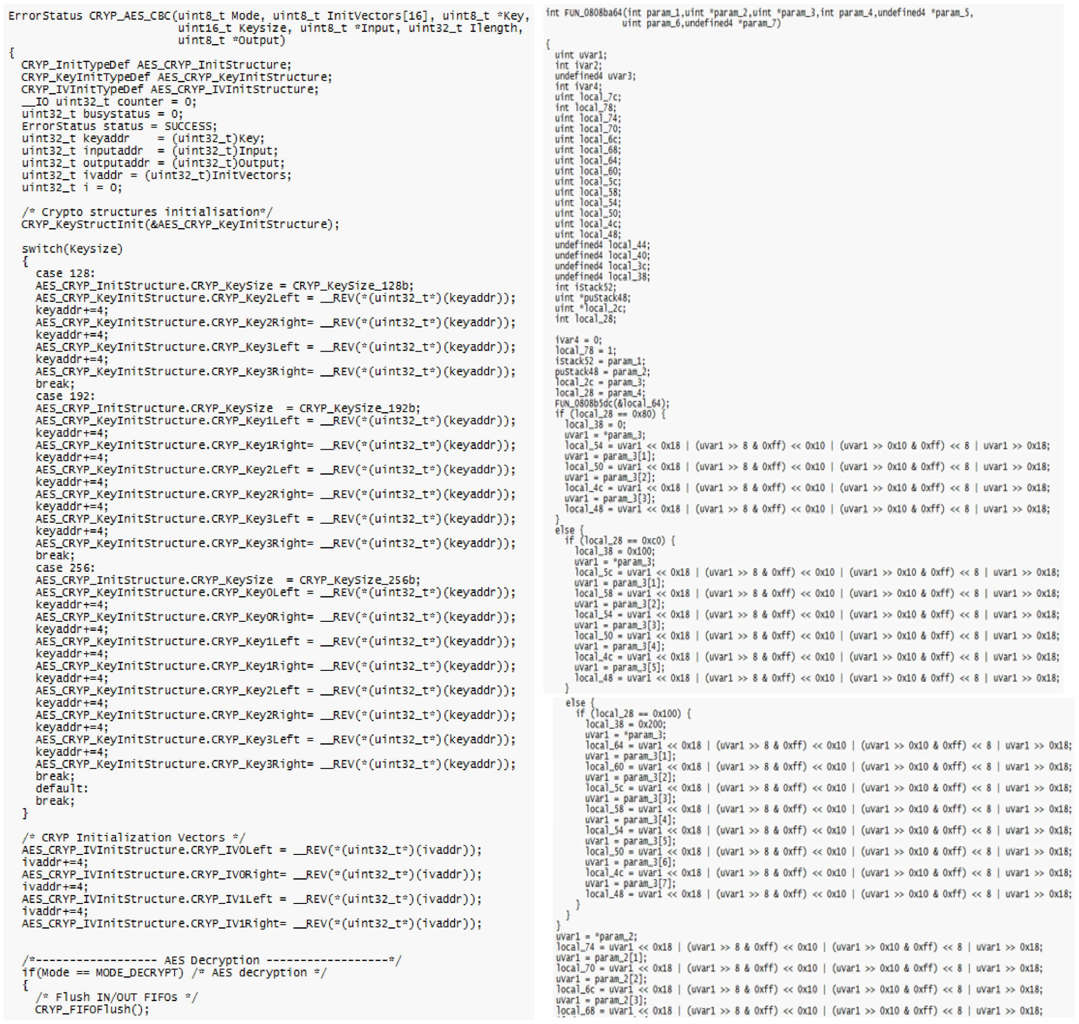
Figure 4.10: The three encryption functions calling on CRYP_DataIn

At this point we suspected that Biotronik might not have written the cryptographic code themselves. As we discovered in the string analysis, they have implemented a third-party compression library. Similarly to the compression functions, the cryptographic functions need to be dependable since their functionality is essential for the security of the data. For Biotronik it would be logical to implement an existing and well-tested cryptographic library to handle the encryption, similarly to the compression library, instead of spending time and resources to develop their own libraries. After all, Biotronik specializes in the medical aspect of the device and we would expect them to generally focus on the development of the medical code sections.

On the STMicroelectronics website, it is possible to send an application to get access to the code libraries of their microprocessors. By making an account at STMicroelectronics and sending an application for their STM32F4 DSP and standard peripherals library under research and academic purposes, we were able to download their peripheral code library for the STM32 microprocessor [55]. The package includes code and drivers for every interface and functionality such as GPIO, UART, RNG, flash, CRC, hash, and crypto(DES, TDES, and AES).

By comparing the three functions calling on CRYP_DATAIN to the code files downloaded from SRMicroelectronics website, we were able to determine that those three function were in fact the DES, TDES, and AES encryption functions. We were also able to rename most of their variables in the encryption functions and rename the parameter variable names coming from the caller functions above using Ghidra's *Function Call Trees feature.* This feature enables us to see the incoming and outgoing function calls for each decompiled function.

Finding 10. *The HMU supports the encryption algorithms DES, TDES, and AES*



```
ErrorStatus CRYP_AES_CBC(uint8_t Mode, uint8_t InitVectors[16], uint8_t *Key,
                         uint16_t Keysize, uint8_t *Input, uint32_t Ilength,
                         uint8_t *Output)
{
  CRYP_InitTypeDef AES_CRYP_InitStructure;
  CRYP_KeyInitTypeDef AES_CRYP_KeyInitStructure;
  CRYP_IVInitTypeDef AES_CRYP_IVInitStructure;
  __IO uint32_t counter = 0;
  uint32_t busystatus = 0;
  ErrorStatus status = SUCCESS;
  uint32_t keyaddr    = (uint32_t)Key;
  uint32_t inputaddr  = (uint32_t)Input;
  uint32_t outputaddr = (uint32_t)Output;
  uint32_t ivaddr = (uint32_t)InitVectors;
  uint32_t i = 0;

  /* Crypto structures initialisation*/
  CRYP_KeyStructInit(&AES_CRYP_KeyInitStructure);

  switch(Keysize)
  {
    case 128:
    AES_CRYP_InitStructure.CRYP_KeySize = CRYP_KeySize_128b;
    AES_CRYP_KeyInitStructure.CRYP_Key2Left = __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key2Right= __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key3Left = __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key3Right= __REV(*(uint32_t*)(keyaddr));
    break;
    case 192:
    AES_CRYP_InitStructure.CRYP_KeySize  = CRYP_KeySize_192b;
    AES_CRYP_KeyInitStructure.CRYP_Key1Left = __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key1Right= __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key2Left = __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key2Right= __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key3Left = __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key3Right= __REV(*(uint32_t*)(keyaddr));
    break;
    case 256:
    AES_CRYP_InitStructure.CRYP_KeySize  = CRYP_KeySize_256b;
    AES_CRYP_KeyInitStructure.CRYP_Key0Left = __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key0Right= __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key1Left = __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key1Right= __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key2Left = __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key2Right= __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key3Left = __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key3Right= __REV(*(uint32_t*)(keyaddr));
    break;
    default:
    break;
  }

  /* CRYP Initialization Vectors */
  AES_CRYP_IVInitStructure.CRYP_IV0Left = __REV(*(uint32_t*)(ivaddr));
  ivaddr+=4;
  AES_CRYP_IVInitStructure.CRYP_IV0Right= __REV(*(uint32_t*)(ivaddr));
  ivaddr+=4;
  AES_CRYP_IVInitStructure.CRYP_IV1Left = __REV(*(uint32_t*)(ivaddr));
  ivaddr+=4;
  AES_CRYP_IVInitStructure.CRYP_IV1Right= __REV(*(uint32_t*)(ivaddr));


  /*------------------ AES Decryption ------------------*/
  if(Mode == MODE_DECRYPT) /* AES decryption */
  {
    /* Flush IN/OUT FIFOs */
    CRYP_FIFOFlush();
```

```
int FUN_0808ba64(int param_1,uint *param_2,uint *param_3,int param_4,undefined4 *param_5,
                uint param_6,undefined4 *param_7)

{
  uint uVar1;
  int iVar2;
  undefined4 uVar3;
  int iVar4;
  uint local_7c;
  int local_78;
  uint local_74;
  uint local_70;
  uint local_6c;
  uint local_68;
  uint local_64;
  uint local_60;
  uint local_5c;
  uint local_58;
  uint local_54;
  uint local_50;
  uint local_4c;
  uint local_48;
  undefined4 local_44;
  undefined4 local_40;
  undefined4 local_3c;
  undefined4 local_38;
  int iStack52;
  uint *puStack48;
  uint *local_2c;
  int local_28;

  iVar4 = 0;
  local_78 = 1;
  iStack52 = param_1;
  puStack48 = param_2;
  local_2c = param_3;
  local_28 = param_4;
  FUN_0808b5dc(&local_64);
  if (local_28 == 0x80) {
    local_38 = 0;
    uVar1 = *param_3;
    local_54 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
    uVar1 = param_3[1];
    local_50 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
    uVar1 = param_3[2];
    local_4c = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
    uVar1 = param_3[3];
    local_48 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
  }
  else {
    if (local_28 == 0xc0) {
      local_38 = 0x100;
      uVar1 = *param_3;
      local_5c = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
      uVar1 = param_3[1];
      local_58 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
      uVar1 = param_3[2];
      local_54 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
      uVar1 = param_3[3];
      local_50 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
      uVar1 = param_3[4];
      local_4c = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
      uVar1 = param_3[5];
      local_48 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
    }
    else {
      if (local_28 == 0x100) {
        local_38 = 0x200;
        uVar1 = *param_3;
        local_64 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
        uVar1 = param_3[1];
        local_60 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
        uVar1 = param_3[2];
        local_5c = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
        uVar1 = param_3[3];
        local_58 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
        uVar1 = param_3[4];
        local_54 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
        uVar1 = param_3[5];
        local_50 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
        uVar1 = param_3[6];
        local_4c = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
        uVar1 = param_3[7];
        local_48 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
      }
    }
  }
  uVar1 = *param_2;
  local_74 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
  uVar1 = param_2[1];
  local_70 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
  uVar1 = param_2[2];
  local_6c = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
  uVar1 = param_2[3];
  local_68 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 | uVar1 >> 0x18;
```

Figure 4.11: Comparison of the STM CRYP-AES-CBC and Ghidra's decompiled AES function

From the figure 4.11 above, we can see several similarities. The first similarity is the amount of parameters in the function definition on top. Both the AES code from

STMicroelectronics and the AES code decompiled in Ghidra recognizes 7 parameters in the function call. They also have a keysize check which is named local_28 in the decompiled code. The keysizes are 128, 192, and 256 bit. They are defined in hexadecimal notation in the decompiled code 0x80, 0xC0, and 0x100, respectively. In the code examples we can also see that the Ghidra decompiler interprets the code to be a series of if- and else-statements, while the STM code shows that the code in reality were a switch-structure with keysize as the parameter. This is a good example of how the source code is changed during the compilation and the subsequent decompilation process. There is also a clear difference between how the keys are written to the registers. In the left image, the keys are obtained from an address, while in the right image there is a complex mathematical operation to obtain every key. This exemplifies the added complexity of reverse engineering compared to analyzing source code.

The STM AES code contained five block cipher modes, ECB, CBC , CTR, GCM, and CCM. However, GCM and CCM are only supported by the STM32F437x, not our STM32F417 model. Our decompiled code also finds code with initialization vectors which are not used by ECB. Hence, the cipher mode used in our case is either CBC or CTR. The AES decryption section of the decompiled code also match the structure of CBC mode.

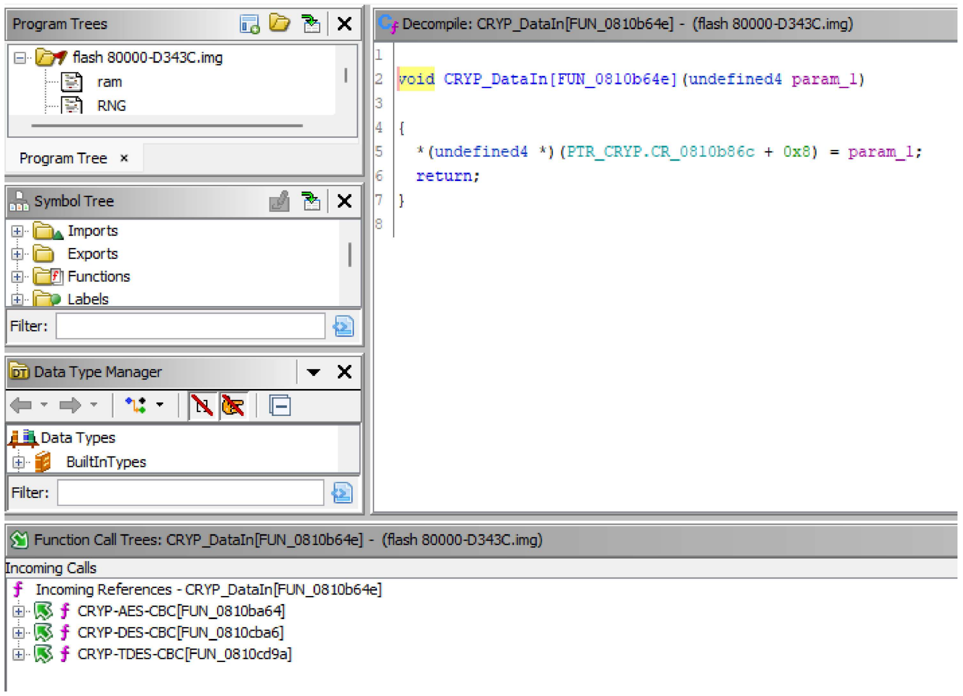
In the STM library code for AES below the AES Encryption section the *AES_CRYP_InitStructure.CRYP_AlgoMode* and the *AES_CRYP_InitStructure.CRYP_DataType* is set. In the decompiled AES function in Ghidra we find two variables at the same place, in line 157 and 159. The first variable is 0x28 in hexadecimal notation which is equal to 00101000 in binary notation. The bits in position 3-5, 101 in this case, consist with AES-CBC according to the algomode in chapter 23.6.1 in the STM32 reference manual [30]. The datatype variable was 0x80, 10000000 in binary notation, and the datatype is defined in bits 6-7. 10 is defined for 8-bit data or bytes. The same approach can be done with the DES and TDES encryption functions. For the decompiled DES function the algomode is 0x18, 00011000 in binary notation, which is defined to be DES-CBC. For the decompiled TDES function the algomode variable contained 0x8, which is 00001000 in binary notation, which is defined to be TDES-CBC in the reference manual. We see a clear structural similarities in the decompiled code compared to the STM standard peripheral library, and we found defined algomode values in the decompiled code that is consistent with the definitions in the STM32 reference manual. From that, we can tell they are using CBC mode for all three encryption algorithms.

Applying the same approach we used to identify AES-CBC on the other two functions that were calling on the CRYP_DataIn function, and we found that those functions were identical to the DES-CBC and TDES-CBC implementations of the STMicroelectronics library. The three functions calling on CRYP_DataIn were the three encryption functions AES-CBC, DES-CBC, and TDES-CBC.

We can therefore conclusively state that they are using the STM cryptographic library. We compared the encryption functions to those available in version 1.9.0 of the STM32F4xx peripheral code library, however it is unknown which version of the STM cryptographic library that is actually implemented.

Finding 11. *All three encryption functions are used in CBC cipher mode*

Finding 12. *The HMU uses the STM cryptographic code library*


Program Trees
flash 80000-D343C.img
ram
RNG
Program Tree
Symbol Tree
Imports
Exports
Functions
Labels
Filter:
Data Type Manager
Data Types
BuiltInTypes
Filter:
Decompile: CRYP_DataIn[FUN_0810b64e] - (flash 80000-D343C.img)
Function Call Trees: CRYP_DataIn[FUN_0810b64e] - (flash 80000-D343C.img)
Incoming Calls
Incoming References - CRYP_DataIn[FUN_0810b64e]
CRYP-AES-CBC[FUN_0810ba64]
CRYP-DES-CBC[FUN_0810cba6]
CRYP-TDES-CBC[FUN_0810cd9a]

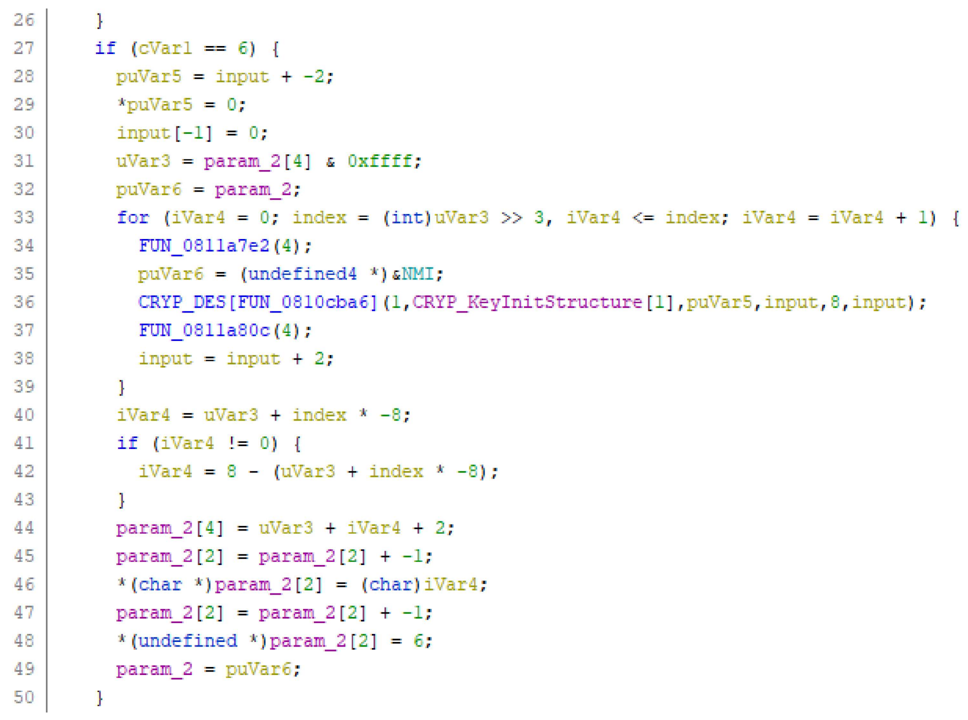
```

void CRYP_DataIn[FUN_0810b64e](undefined4 param_1)

{
  *(undefined4 *)(PTR_CRYP.CR_0810b86c + 0x8) = param_1;
  return;
}

```

Figure 4.12: The three encryption functions calling on CRYP_DataIn

By identifying the encryption function we were able to identify and rename variable names. This was especially helpful with the parameter names which would pass through to the caller functions above in the function hierarchy.

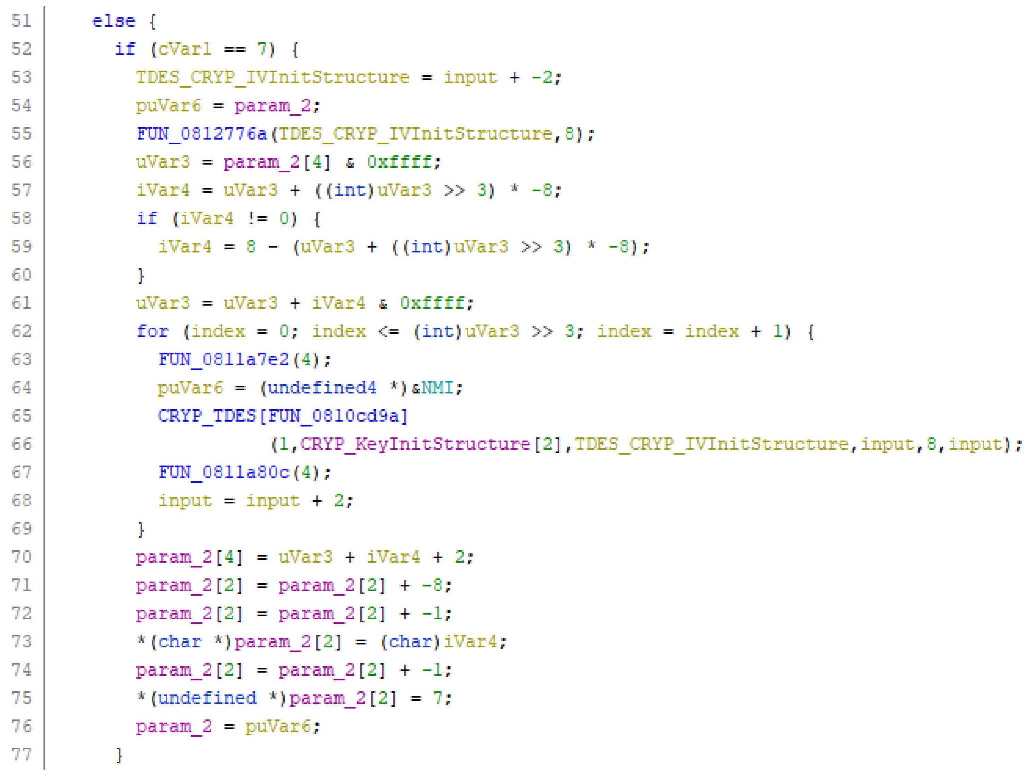
4.4.4 Bottom-up Approach in the Function Hierarchy

Similarly to how all the low-level cryptographic functions were all called by the three encryption functions, the Function Call Trees feature shows that all three of the encryption functions are called by the same two functions. These two functions were almost identical in structure. In one of these functions, the first parameter sent to the encryption functions were all 0, and in the other function the first parameter of each encryption function call were 1. Apart from this key difference, the two functions were quite similar. From the fully reversed encryption functions we know that the first parameter received is the direction, i.e either encrypt or decrypt. We chose to name these two functions the Encryption-Handler and the Decryption-Handler. Both of the encryption and decryption handler functions are added in the compressed zip-file attached with this thesis, which will not be published for security reasons.



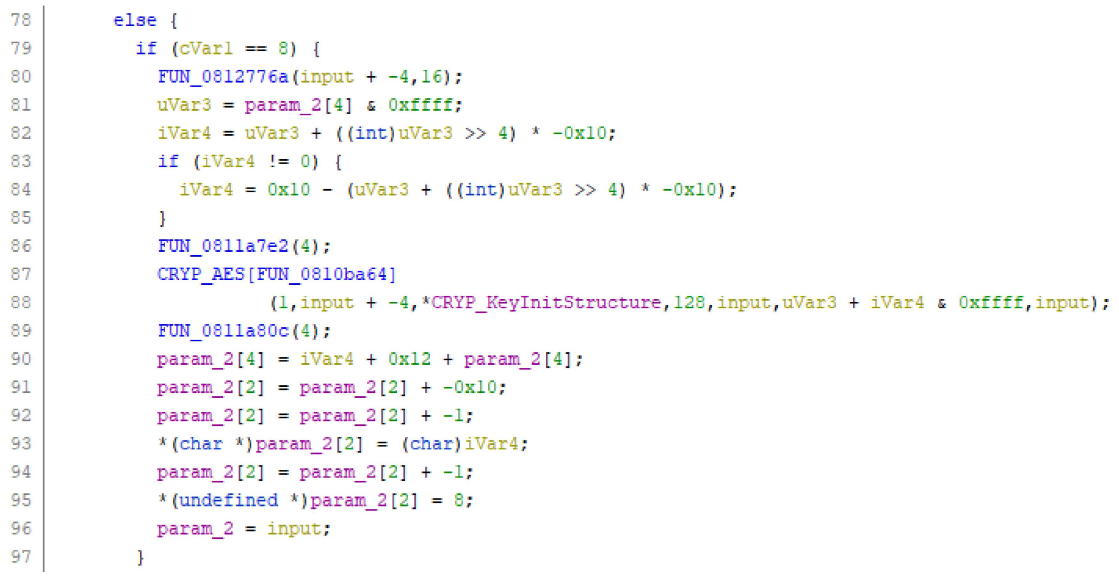
```
  }
  if (cVar1 == 6) {
    puVar5 = input + -2;
    *puVar5 = 0;
    input[-1] = 0;
    uVar3 = param_2[4] & 0xffff;
    puVar6 = param_2;
    for (iVar4 = 0; index = (int)uVar3 >> 3, iVar4 <= index; iVar4 = iVar4 + 1) {
      FUN_0811a7e2(4);
      puVar6 = (undefined4 *)&NMI;
      CRYP_DES[FUN_0810cba6](1,CRYP_KeyInitStructure[1],puVar5,input,8,input);
      FUN_0811a80c(4);
      input = input + 2;
    }
    iVar4 = uVar3 + index * -8;
    if (iVar4 != 0) {
      iVar4 = 8 - (uVar3 + index * -8);
    }
    param_2[4] = uVar3 + iVar4 + 2;
    param_2[2] = param_2[2] + -1;
    *(char *)param_2[2] = (char)iVar4;
    param_2[2] = param_2[2] + -1;
    *(undefined *)param_2[2] = 6;
    param_2 = puVar6;
  }
```

Figure 4.13: Call for DES in Encryption-Handler



```
    else {
      if (cVar1 == 7) {
        TDES_CRYP_IVInitStructure = input + -2;
        puVar6 = param_2;
        FUN_0812776a(TDES_CRYP_IVInitStructure,8);
        uVar3 = param_2[4] & 0xffff;
        iVar4 = uVar3 + ((int)uVar3 >> 3) * -8;
        if (iVar4 != 0) {
          iVar4 = 8 - (uVar3 + ((int)uVar3 >> 3) * -8);
        }
        uVar3 = uVar3 + iVar4 & 0xffff;
        for (index = 0; index <= (int)uVar3 >> 3; index = index + 1) {
          FUN_0811a7e2(4);
          puVar6 = (undefined4 *)&NMI;
          CRYP_TDES[FUN_0810cd9a]
                    (1,CRYP_KeyInitStructure[2],TDES_CRYP_IVInitStructure,input,8,input);
          FUN_0811a80c(4);
          input = input + 2;
        }
        param_2[4] = uVar3 + iVar4 + 2;
        param_2[2] = param_2[2] + -8;
        param_2[2] = param_2[2] + -1;
        *(char *)param_2[2] = (char)iVar4;
        param_2[2] = param_2[2] + -1;
        *(undefined *)param_2[2] = 7;
        param_2 = puVar6;
      }
```

Figure 4.14: Call for TDES in Encryption-Handler



```
      else {
        if (cVar1 == 8) {
          FUN_0812776a(input + -4,16);
          uVar3 = param_2[4] & 0xffff;
          iVar4 = uVar3 + ((int)uVar3 >> 4) * -0x10;
          if (iVar4 != 0) {
            iVar4 = 0x10 - (uVar3 + ((int)uVar3 >> 4) * -0x10);
          }
          FUN_0811a7e2(4);
          CRYP_AES[FUN_0810ba64]
                    (1,input + -4,*CRYP_KeyInitStructure,128,input,uVar3 + iVar4 & 0xffff,input);
          FUN_0811a80c(4);
          param_2[4] = iVar4 + 0x12 + param_2[4];
          param_2[2] = param_2[2] + -0x10;
          param_2[2] = param_2[2] + -1;
          *(char *)param_2[2] = (char)iVar4;
          param_2[2] = param_2[2] + -1;
          *(undefined *)param_2[2] = 8;
          param_2 = input;
        }
```

Figure 4.15: Call for AES in Encryption-Handler

Another quite interesting finding is that the *encryption- and decryption-handler* functions both check whether an integer variable is 6 in line 27 to call DES, 7 in line 52 to call TDES, or 8 in line 79 to call AES. This is similar to Bour's findings in their analysis of the communication protocol of the older Biotronik CardioMessenger II-S T-Line model HMU [54]. From figure 4.11 we know that the forth parameter of the AES function is the defined keysize. In figure 4.15 we can see that when the AES function is called in the ENCRYPTION-HANDLER line 87-88, the fourth parameter is 128. The implemented AES function is only called by two functions, the ENCRYPTION-HANDLER and the DECRYPTION-HANDLER. In both of these functions the keysize is defined to be 128 bits. The implemented AES function supports 128, 192, and 256 bit in keysize, but 192- and 256-bit are therefore never used.

Finding 13. *The encryption type DES, TDES, and AES is selected by the number 5, 6, and 7 respectively, which is identical to the older CardioMessenger II HMU models*

Finding 14. *AES-CBC is only used with a 128-bit key*

There are a couple key differences between the ENCRYPTION-HANDLER and the DECRYPTION-HANDLER. Both of the functions have checks for the value that indicated the selected encryption algorithm, but the DECRYPTION-HANDLER initially starts by checking if the value is less than 10. If the value is not less than 10, the else-clause checks if the value is either 10 or 0x65. 0x65 represented as the ASCII character 'e'. Perhaps e as in encrypted. If this value is neither 10 nor 0x65, the return value is set to 0. If the encryptionType value is 6 and DES-CBC is selected, its third parameter is interpreted to be 0xD8. From the STM library and analysis of the encryption functions, we know that the third parameter to all of the encryption functions is the initialization vector array variable. This variable is supposed to be an array where the first two indexes contain the initialization vector. Perhaps this number signifies something else, since it would not make sense to have a hardcoded byte as an initialization vector.

By continuing to follow the Function Call Trees upwards in the hierarchy, there was only one function that made a call to our ENCRYPTION-HANDLER and one other function that made a call to our DECRYPTION-HANDLER. At this point we were able to identify a completely different kind of function. Functions that were not part of the STM32 peripheral library. These functions had logging features with debugging strings which made it possible for us to determine the functions functionality based on the contents of the strings. Their structure was different from the other functions we had seen so far. The structure of these functions were on the form similar to the example code below. The var variable would be 0 or 1 depending on the return value of the function. This was also the case in the three encryption function were the return value was named ErrorStatus, and were also either 0 or 1.

Algorithm 4.1 The general structure of code

```
var = FUNCTION1()
if(var==1){
    # if success: Perform functionality
    FUNCTION2()
}
else{
    # Reveals naming convention
    LOG("*function_name* - debug string")
}
```

The function calling on the ENCRYPTION-HANDLER had multiple logging functions with debugging strings such those listed below.

```
= "PDHM: pdhm_decode_tx_handler - adding frame: frm=%d, len=%d\r\n"
= "PDHM: pdhm_decode_tx_handler - skip result: dropping frm=%d\r\n"
= "PDHM: pdhm_decode_tx_handler - frame error\r\n"
= "PDHM: pdhm_decode_tx_handler - no message available\r\n"
```

This function is named PDHM_DECODE_TX_HANDLER (PDHM-TX-Handler hereafter), and the clear difference in structure indicates that we have crossed the line between the functions written by STMicroelectronics and those written by Biotronik.

Similarly, the function that calls on the DECRYPTION-HANDLER contains the debugging strings that are listed below.

```
"PDHM: pdhm_decode_rx_handler - invalid pointer detected"
"PDHM: pdhm_decode_rx_handler - DECOMPRESSION error"
"PDHM: pdhm_decode_rx_handler - ENCRYPTION error"
"PDHM: pdhm_decode_rx_handler - rx-timeout"
"PDHM: pdhm_decode_rx_handler - malloc error"
"PDHM: pdhm_decode_rx_handler - frame length error"
"PDHM: pdhm_decode_rx_handler - CRC error"
"PDHM: pdhm_decode_rx_handler - nothing to send"
"PDHM: pdhm_decode_rx_handler - transport layer expected"
```

This function, calling on the DECRYPTION-HANDLER, is named PDHM_DECODE_RX_HANDLER (PDHM-RX-Handler hereafter). The first four characters *PDHM* are present in many functions that are related to communication. PDHM is therefore likely the name of their communication protocol.

The structure of the function calls can be graphed in Ghidra with its Function Call Graph feature. The figure below displays the function calls from the top and downwards. The PDHM-RX-HANDLER calls the DECRYPTION-HANDLER, which in turn calls one of the encryption functions AES, DES or TDES, and they call onto the low-level cryptographic functions, such as the previously mentioned CRYP_DATAIN. Similarly, the PDHM-TX-HANDLER calls the ENCRYPTION-HANDLER which in turn calls one of the three encryption functions. The key difference between the PDHM handler functions are the RX and TX, which stands for receiver and transceiver, and the direction parameter which is sent from the decryption/encryption handlers to the three encryption functions. Another important clarification is that the PDHM RX and TX handlers call onto many other functions that are not shown in the figure, however the figure shows the direct calling convention in the context from the AES-CBC function. The PDHM handler functions are also involved in many other functions and we will analyze them further in the next subsection, but firstly we need to continue our bottom-up approach until we are able to identify the functions in top of the hierarchy, and all the PDHM functions.

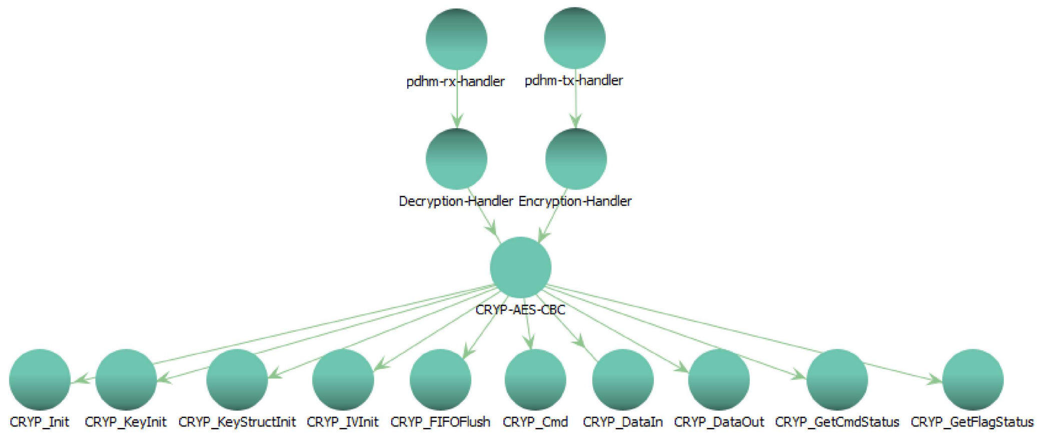
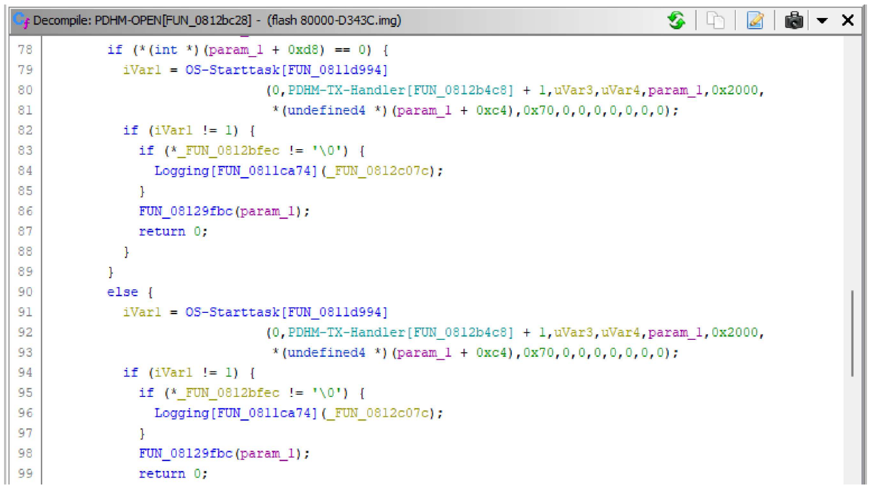


Figure 4.16: The Function Call Trees in the context of CRYP-AES-CBC. The call tree is identical for (T)DES-CBC

The Function Call Trees feature in Ghidra does not find any incoming function calls to the PDHM-RX-HANDLER function. This is strange because the receiver function needs to be initiated somewhere. The lack of this incoming function call might be the result of an inaccuracy in Ghidra's decompilation process. However, Ghidra is able to find the incoming reference for the PDHM-TX-Handler which comes from the function PDHM_OPEN. We were able to name this function based off its many debugging strings starting with *pdhm_open*. PDHM-OPEN contains an interesting nested structure of if-statements, where the function identified as OS-Starttask is called once within each if-statement. Each if-statement checks the return value of a call to the function OS-Starttask. If the return value is 1, as in success, the code advances to the next inner if-statement with another OS-Starttask function call. The difference between the calls to OS-Starttask is the parameters that are sent to the function. The second parameter is always a function, and the OS-Starttask seems to create a task or a thread for this function task to initiate. In PDHM-OPEN we can see that most of the OS-Starttask calls contain the PDHM-TX-Handler as a second parameter, as can be seen in the figure 4.17 below. We also named the function OS-Starttask because it seems to initiate multiple functions or tasks. We also found OS-Starttask function calls within other functions to initiate other functions.


Decompile: PDHM-OPEN[FUN_0812bc28] - (flash 80000-D343C.img)

```
      if (*(int *)(param_1 + 0xd8) == 0) {
        iVar1 = OS-Starttask[FUN_0811d994]
                          (0,PDHM-TX-Handler[FUN_0812b4c8] + 1,uVar3,uVar4,param_1,0x2000,
                           *(undefined4 *)(param_1 + 0xc4),0x70,0,0,0,0,0,0,0);
        if (iVar1 != 1) {
          if (*_FUN_0812bfec != '\0') {
            Logging[FUN_0811ca74](_FUN_0812c07c);
          }
          FUN_08129fbc(param_1);
          return 0;
        }
      }
      else {
        iVar1 = OS-Starttask[FUN_0811d994]
                          (0,PDHM-TX-Handler[FUN_0812b4c8] + 1,uVar3,uVar4,param_1,0x2000,
                           *(undefined4 *)(param_1 + 0xc4),0x70,0,0,0,0,0,0,0);
        if (iVar1 != 1) {
          if (*_FUN_0812bfec != '\0') {
            Logging[FUN_0811ca74](_FUN_0812c07c);
          }
          FUN_08129fbc(param_1);
          return 0;
```

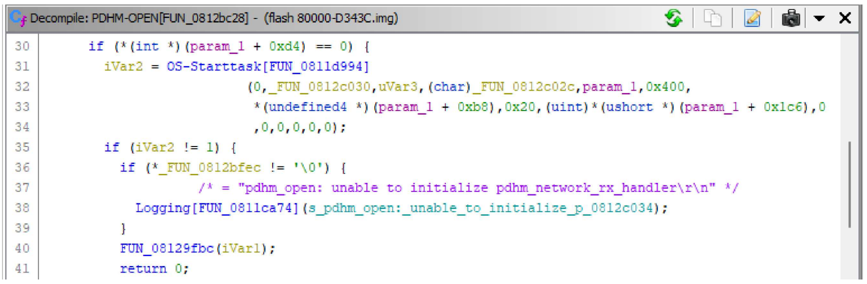
Figure 4.17: Initializing PDHM-TX-Handler in PDHM_OPEN with OS-Starttask

The other OS-Starttask calls in PDHM_OPEN referenced a function that looked malformed by decompilation. If the return value from OS-Starttask was 0, function logged the debugging string *"pdhm_open: unable to initialize pdhm_network_rx_handler"*. This means that it is trying to start a task for PDHM-Network-RX-Handler, but that the decompilation might be inaccurate. This is interesting because the PDHM-Network-RX-Handler function has a OS-Starttask statement that sends the PDHM-RX-Handler as a second parameter, but Ghidra did not recognize this as an incoming function call reference. Hence, the PDHM-OPEN initiates OS-Starttask for the PDHM-TX-Handler directly, and for the PDHM-RX-Handler through PDHM-Network-RX-Handler. The selection between PDHM-Network-RX-Handler and PDHM-TX-Handler is determined by the variable value in crypStruct at offset 0x1FF. CrypStruct is a variable structure which contains the keys used in the encryption functions. It is sent as a parameter far up in the functions hierarchy. The PDHM-Network-RX-Handler is selected if the following code is true. This is code snippet is above the code in figure 4.18.

Finding 15. *The function PDHM_OPEN initiates both the receiver and transceiver functions for the PDHM communication protocol*

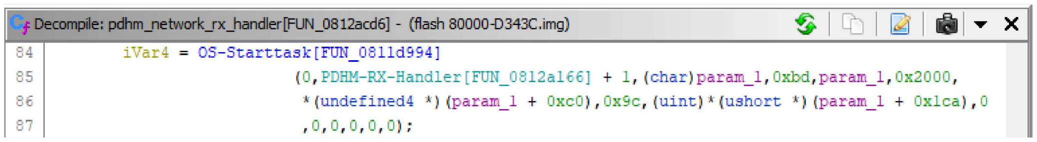
```
((*(byte *)(crypStruct + 0x1ff) & 1) == 1)
```

This code retrieves the value of a variable in the structure at crypStruct+0x1FF and performs a binary AND operation with 1. It essentially checks if a variable in the structure is equal to 1. Similarly, the PDHM-TX-Handler is selected if only the second last bit is set, and equal to 2. These types of binary operations are found in many places of the code. In this case, the byte pointer is dereferenced, and thus comparing the inherent value contained at the address (crypStruct+0x1FF). The entirety of the PDHM receiver and transceiver functions are also added in the zip-file attachment with this thesis.


Decompile: PDHM-OPEN[FUN_0812bc28] - (flash 80000-D343C.img)

```
    if (*(int *)(param_1 + 0xd4) == 0) {
      iVar2 = OS-Starttask[FUN_0811d994]
                        (0,_FUN_0812c030,uVar3,(char)_FUN_0812c02c,param_1,0x400,
                         *(undefined4 *)(param_1 + 0xb8),0x20,(uint)*(ushort *)(param_1 + 0x1c6),0
                         ,0,0,0,0,0);
      if (iVar2 != 1) {
        if (*_FUN_0812bfec != '\0') {
                    /* = "pdhm_open: unable to initialize pdhm_network_rx_handler\r\n" */
          Logging[FUN_0811ca74](s_pdhm_open:_unable_to_initialize_p_0812c034);
        }
        FUN_08129fbc(iVar1);
        return 0;
```

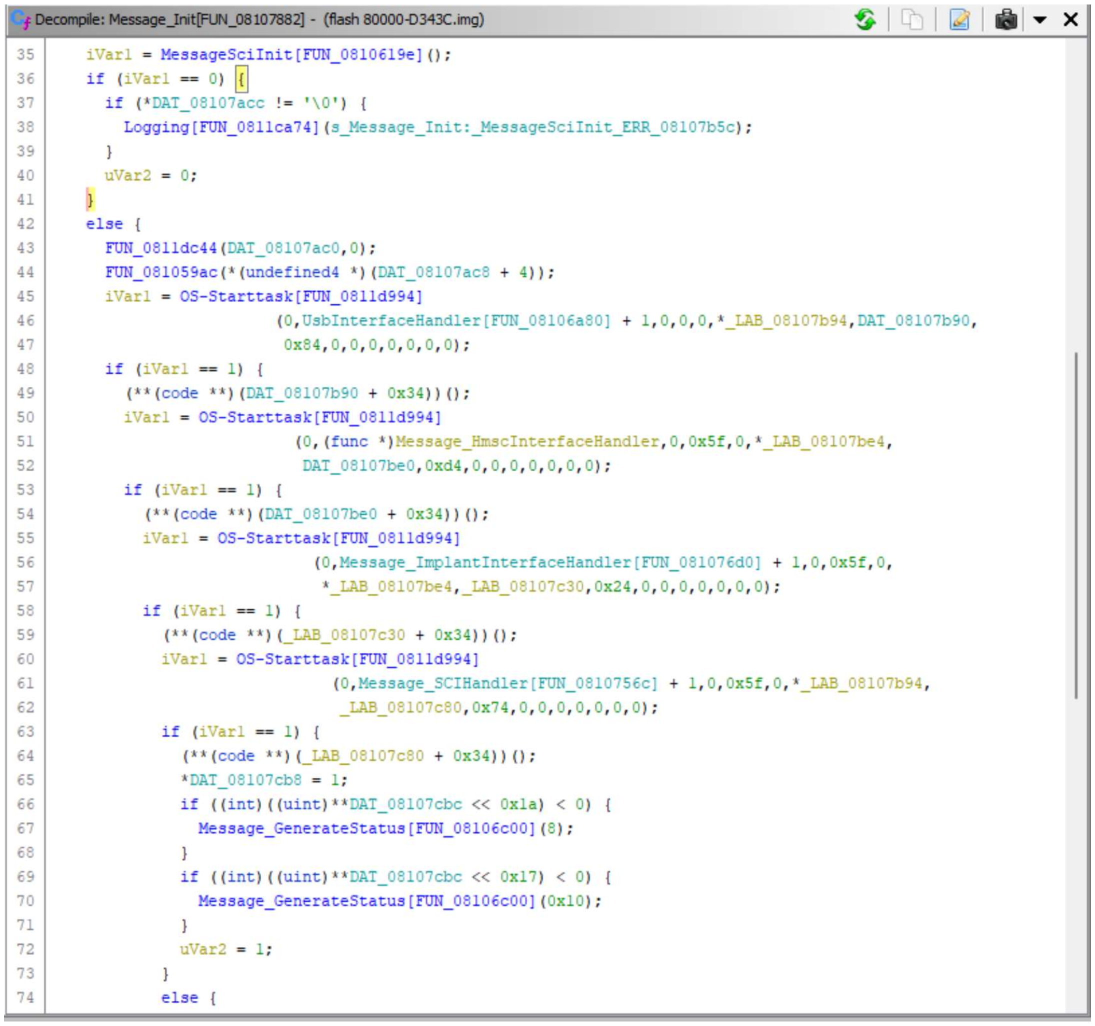
Figure 4.18: Initializing PDHM-Network-RX-Handler in PDHM_OPEN with OS-Starttask


Decompile: pdhm_network_rx_handler[FUN_0812acd6] - (flash 80000-D343C.img)

```
        iVar4 = OS-Starttask[FUN_0811d994]
                          (0,PDHM-RX-Handler[FUN_0812a166] + 1,(char)param_1,0xbd,param_1,0x2000,
                           *(undefined4 *)(param_1 + 0xc0),0x9c,(uint)*(ushort *)(param_1 + 0x1ca),0
                           ,0,0,0,0,0);
```

Figure 4.19: Initializing PDHM-RX-Handler in PDHM-Network-RX-Handler with OS-Starttask

Now we know that the PDHM-OPEN function is responsible for initializing the receiver and transceiver functions for the PDHM protocol. The PDHM-OPEN function is only called from the function UsbInterfaceHandler. The UsbInterfaceHandler is also initialized through an OS-Starttask statement within the Message-Init function which can be seen below in figure 4.20. If the PDHM-OPEN ErrorStatus return value is not 1, the error message = *"Message_UsbInterfaceHandler: unable to open virtual pdhm port"* is logged and the UsbInterfaceHandler returns 0 to Message-Init. The Message-Init has a nested structure of if-statements for starting the main services on the device. The UsbInterfaceHandler is the first of four services in a nested structure. Consequently, if the UsbInterfaceHandler return 0 as the ErrorStatus, the services for HmscInterfaceHandler, ImplantInterfaceHandler, and the SCIHandler will not be initiated.



Decompile: Message_Init[FUN_08107882] - (flash 80000-D343C.img)

```
35    iVar1 = MessageSciInit[FUN_0810619e]();
36    if (iVar1 == 0) {
37      if (*DAT_08107acc != '\0') {
38        Logging[FUN_0811ca74](s_Message_Init:_MessageSciInit_ERR_08107b5c);
39      }
40      uVar2 = 0;
41    }
42    else {
43      FUN_0811dc44(DAT_08107ac0,0);
44      FUN_081059ac(*(undefined4 *)(DAT_08107ac8 + 4));
45      iVar1 = OS-Starttask[FUN_0811d994]
46                        (0,UsbInterfaceHandler[FUN_08106a80] + 1,0,0,0,*_LAB_08107b94,DAT_08107b90,
47                         0x84,0,0,0,0,0,0,0);
48      if (iVar1 == 1) {
49        (**(code **)(DAT_08107b90 + 0x34))();
50        iVar1 = OS-Starttask[FUN_0811d994]
51                          (0,(func *)Message_HmscInterfaceHandler,0,0x5f,0,*_LAB_08107be4,
52                           DAT_08107be0,0xd4,0,0,0,0,0,0,0);
53        if (iVar1 == 1) {
54          (**(code **)(DAT_08107be0 + 0x34))();
55          iVar1 = OS-Starttask[FUN_0811d994]
56                            (0,Message_ImplantInterfaceHandler[FUN_081076d0] + 1,0,0x5f,0,
57                             *_LAB_08107be4,_LAB_08107c30,0x24,0,0,0,0,0,0,0);
58          if (iVar1 == 1) {
59            (**(code **)(_LAB_08107c30 + 0x34))();
60            iVar1 = OS-Starttask[FUN_0811d994]
61                              (0,Message_SCIHandler[FUN_0810756c] + 1,0,0x5f,0,*_LAB_08107b94,
62                               _LAB_08107c80,0x74,0,0,0,0,0,0,0);
63            if (iVar1 == 1) {
64              (**(code **)(_LAB_08107c80 + 0x34))();
65              *DAT_08107cb8 = 1;
66              if ((int)((uint)**DAT_08107cbc << 0x1a) < 0) {
67                Message_GenerateStatus[FUN_08106c00](8);
68              }
69              if ((int)((uint)**DAT_08107cbc << 0x17) < 0) {
70                Message_GenerateStatus[FUN_08106c00](0x10);
71              }
72              uVar2 = 1;
73            }
74            else {
```

Figure 4.20: The nested structure of the Message-Init function

It appears that the Message-Init function is in charge of initiating certain primary services running on the HMU. A snippet of the Message-Init function can be seen above in figure 4.20. The OS-Starttask function is called for each of these services, and the function is passed as the second parameter to the OS-Starttask call. The UsbInterfaceHandler that is initiated in line 45-46 in Message-Init is also the only function that calls the PDHM-OPEN, which in turn starts either of the PDHM RX or TX handler functions. The Message-Init function is among the top of the functions hierarchy, and it initializes the main services on the HMU. These services do in turn start sub-processes. An example of this is PDHM-OPEN which initiates the PDHM-Network-RX-Handler and the PDHM-TX-Handler sub-processes.

The function that calls to Message-Init is the last function Ghidra recognizes in the Function Call Trees. It does not contain any debugging strings or peripheral names, and it is called FUN_081055cc from decompilation. It does not contain any debugging strings with a function name. It is not a particularly long function but it calls to 12 functions, one of them is the Message-Init. Each of these functions call a vast amount of other functions, and it is clear that we have reached one of the functions at the very top of the hierarchy. However it is surprising that the FUN_081055cc functions returns either 0 or 1, but it does not have any recognized incoming function calls. The structure is similar to the other functions/services that have been started and returning an error status. They return either 1 if successful, or 0 if failed. Ghidra also has the ability to search for references to a function name or function address, but Ghidra did not find any such references for FUN_081055cc. It might be another inaccuracy in the decompilation. The function also contains many references and pointers to data addresses in the form DAT_08xxxxxx. Trying to decompile the data sections adjacent to this function gives malformed code, and is therefore hard to reverse any further. There are also no recognized references to the function's base address in memory. However, since we previously cut the binary flash file, this might be an unintended side-effect of that. Launching the older Ghidra project with the entire flash memory which had duplication, shows that the incoming calls to FUN_081055cc is recognized by Ghidra. It is called by one function that has two OS-Starttask calls. One of them contains the string *system_task* and the other OS-Starttask contains the string *idle_task*. This function is also referenced once by a function that contains OS functions that are recognized by Ghidra as *getMainStackPointer()*, *getProcessStackPointer()*, *isThreadModePrivileged()*, etc. These are common stack pointers for Cortex-M microprocessors [56]. In this function they seem to handle whether a task is running in a privileged thread mode. This is the top of the function hierarchy.

Real-Time Operating System

The name OS-Starttask was chosen because it seems to initiate services by receiving functions as input parameters. Inside the OS-Starttask function a long array is set with the different incoming parameters, and structure also looks similar to code from a real-time operating system such as the FreeRTOS [57]. Since the CardioMessenger Smart 3G was initially released in 2015, we downloaded the Keil.STM32F4xx_DFP.2.2.0 series drivers and peripherals library that was released in october 2014. A year before the release of this HMU. It is the earliest software and driver library available for download at the keil website for the STM32F417 microprocessor [55]. In the cmsis_os header file we find the core functions of the

CMSIS-RTOS documentation and API version 1.02. The os header contains the full API with descriptions of thread- and memory pool management. RTOS 1.02 has the function osThreadCreate similar to OS-Starttask and the same memory allocation function we identified. Neither the strings nor the code in our reverse engineering analysis of the HMU have revealed concrete OS information, but the functionality and the prior release date of the OS correlates with the HMU.

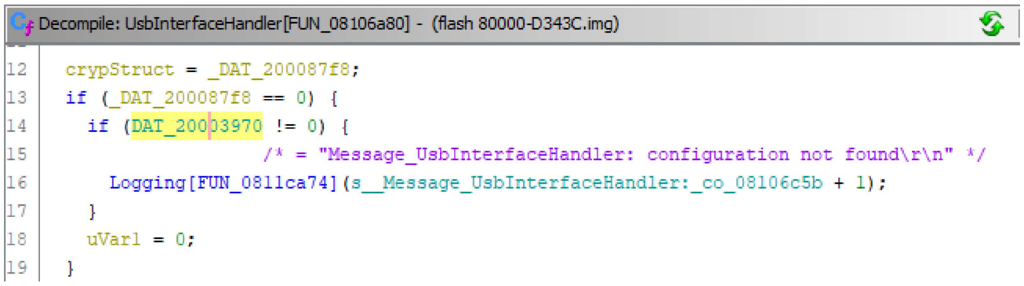
Finding 16. *The HMU uses the CMSIS Real-Time Operating System (RTOS) API version 1.02*

If the Message_Init function returns an error status of 1, the function calls four new functions. These functions contain a pointer to the IWDG peripheral and its KR register. This is the Independent Watchdog peripheral of the microprocessor, described in section 21 of the reference manual [30]. The independent watchdog peripheral serves the purpose of detecting and resolving malfunctions in software, and to trigger interrupts and system resets. The first value sent the IWDG is 0x5555 which is the key to enable access to IWDG's PR and RLR registers. The next functions sends the value 6, 110 in binary notation, to the IWDG PR register to select the 32 prescaler device to the counter clock. The third functions sends 0xFFF to the IWDG offset 0x8 which is the RLR register. The value in RLR defines where the watchdog counter starts to count down from. In this case, the 0xFFF is the reset by standby mode value. The last function within the if-clause sends the 0xCCCC value to the IWDG KR register. According to the reference manual 21.4.1, writing the value 0xCCCC starts the independent watchdog peripheral. A peripheral that serves to detect and resolve software malfunctions is likely a service that is started early on by the software. This is consistent with our function hierarchy finding. The reference manual also specifies that the microprocessor has a hardware watchdog capability, but we have not found any references to it in the code. Which may be correct as the hardware watchdog is scheduled to launch automatically at device power-on.

Configuration in SRAM

While going up the function hierarchy we are able to trace the parameters that are sent to our PDHM protocol functions. We know that the first parameter that is sent to the PDHM-RX-Handler and the PDHM-TX-Handler is the crypStruct. The object structure for all the processes of the cryptographic module that we identified in the encryption functions from the STM peripheral code library. The PDHM functions receive the crypStruct from PDHM-OPEN, which in turn receives the cryptStruct as the first parameter from the UsbInterfaceHandler function. In

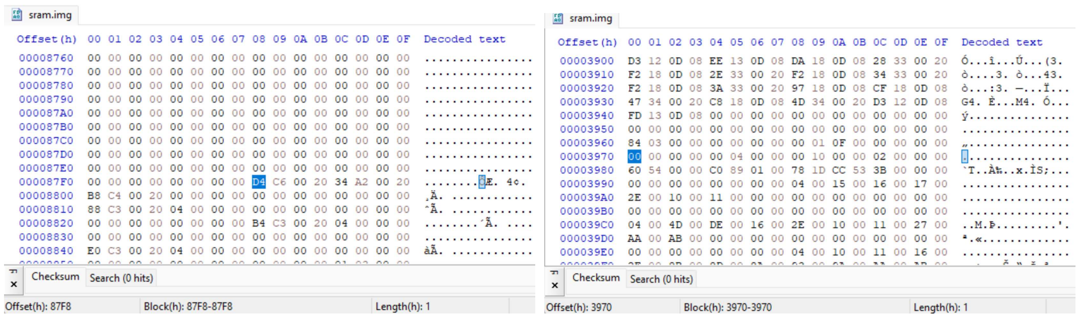
UsbInterfaceHandler we can see that the first parameter sent to PDHM-OPEN is a variable obtained directly from memory, at position 200087f8. This is in the SRAM memory section, 0x2000 0000 - 0x2001 BFFF. This finding was based on the project file that only analyzed the largest code block from the flash memory section. When analyzing the entire flash file, there was no such reference in the UsbInterfaceHandler to a location in the SRAM. Whether the crypStruct is located at address 200087f8 is therefore uncertain.


Decompile: UsbInterfaceHandler[FUN_08106a80] - (flash 80000-D343C.img)

```
  crypStruct = _DAT_200087f8;
  if (_DAT_200087f8 == 0) {
    if (DAT_20003970 != 0) {
                    /* = "Message_UsbInterfaceHandler: configuration not found\r\n" */
      Logging[FUN_0811ca74](s__Message_UsbInterfaceHandler:_co_08106c5b + 1);
    }
    uVar1 = 0;
  }
```

Figure 4.21: Code snippet from UsbInterfaceHandler. Showing memory location of crypStruct object

The first if-statement checks if the byte at crypStruct memory location is 0. If this is the case, another memory location is checked and an error message stating *"Message_UsbInterfaceHandler: configuration not found"* is logged. Therefore it seems that there is a configuration value or object in sram, starting at the offset 0x3970. Since we dumped the sram previously, we are able to open the memory dump with a hex editor to view the memory locations directly. We used the HxD editor which is a free hex- and disk editing software.


sram.img

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00008760  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008770  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008780  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008790  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000087A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000087B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000087C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000087D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000087E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000087F0  00 00 00 00 00 00 00 00 D4 C6 00 20 34 A2 00 20  ........ÔÆ. 4¢. 
00008800  B8 C4 00 20 00 00 00 00 00 00 00 00 00 00 00 00  ¸Ä. ............
00008810  88 C3 00 20 04 00 00 00 00 00 00 00 00 00 00 00  ˆÃ. ............
00008820  00 00 00 00 00 00 00 00 B4 C3 00 20 04 00 00 00  ........´Ã. ....
00008830  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008840  E0 C3 00 20 04 00 00 00 00 00 00 00 00 00 00 00  àÃ. ............
```

Checksum Search (0 hits)

Offset(h): 87F8 Block(h): 87F8-87F8 Length(h): 1

sram.img

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00003900  D3 12 0D 08 EE 13 0D 08 DA 18 0D 08 28 33 00 20  Ó...î...Ú...(3. 
00003910  F2 18 0D 08 2E 33 00 20 F2 18 0D 08 34 33 00 20  ò....3. ò...43. 
00003920  F2 18 0D 08 3A 33 00 20 97 18 0D 08 CF 18 0D 08  ò...:3. —...Ï...
00003930  47 34 00 20 C8 18 0D 08 4D 34 00 20 D3 12 0D 08  G4. È...M4. Ó...
00003940  FD 13 0D 08 00 00 00 00 00 00 00 00 00 00 00 00  ý...............
00003950  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003960  84 03 00 00 00 00 00 00 00 01 0F 00 00 00 00 00  „...............
00003970  00 00 00 00 00 04 00 00 00 10 00 00 02 00 00 00  ................
00003980  60 54 00 00 C0 89 01 00 78 1D CC 53 3B 00 00 00  `T..À‰..x.ÌS;...
00003990  00 00 00 00 00 00 00 00 04 00 15 00 16 00 17 00  ................
000039A0  2E 00 10 00 11 00 00 00 00 00 00 00 00 00 00 00  ................
000039B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000039C0  04 00 4D 00 DE 00 16 00 2E 00 10 00 11 00 27 00  ..M.Þ.........'.
000039D0  AA 00 AB 00 00 00 00 00 00 00 00 00 00 00 00 00  ª.«.............
000039E0  00 00 00 00 00 00 00 00 04 00 10 00 11 00 16 00  ................
```

Checksum Search (0 hits)

Offset(h): 3970 Block(h): 3970-3970 Length(h): 1

Figure 4.22: The memory locations of the crypStruct and configuration respectively

The left subfigure shows the memory section of the crypStruct. The value at offset 0x87F8 in the SRAM is not 0, which means the code does not progress to the inner if-statement that checks if the configuration is found. The subfigure on the right is the memory section for the configuration object. Since the value at offset 0x3970 is 0, this means the configuration is defined as expected and that the error message is not logged. The code shows that a configuration file or object is supposed to set the values of crypStruct. Otherwise the UsbInterfaceHandler returns 0 back to Message-Init. The UsbInterfaceHandler is dependent on a configured crypStruct object for the code to progress to the PDHM-OPEN call. The sizes of crypStruct and configuration is not evident, but their start addresses in memory and the CRYP object's structure is known and defined in the STM32 library. However, it is not certain that the crypStruct object is implemented identically to the CRYP object. The STM32 library definition of the CRYP object's structure can nevertheless be used for comparison to aid the reverse engineering process.

Cryptographic Keys & Initialization vectors

In the microprocessor the encryption keys are stored in the left and right key registers CRYP_KxL/R. CRYP_K1L/R for DES, CRYP_K1L/R, CRYP_K2L/R, and CRYP_K3L/R for TDES, and in CRYP_K2L/R and CRYP_K3L/R for AES-128 [30]. In our decompiled (T)DES encryption functions these keys are gotten through the Key array which we identified in the STM CRYP library [55]. In the Encryption-Handler, keyStruct[1] is sent as the Key array parameter to DES-CBC and keyStruct[2] is sent as the Key array parameter to TDES-CBC. In the DES-CBC function, the key is gotten from the first two indexes of the Key object. In the TDES-CBC function, the three keys are located in the first six indexes of the Key array. This means that keyStruct is a two-dimensional array that holds the values of the K1 registers in index 1, and the values of K1, K2, and K3 at index 2.

```
# STM CRYP library TDES key init
  TDES_CRYP_KeyInitStructure.CRYP_Key1Left = __REV(*(uint32_t*)(keyaddr));
  keyaddr+=4;
  TDES_CRYP_KeyInitStructure.CRYP_Key1Right= __REV(*(uint32_t*)(keyaddr));
  keyaddr+=4;
  TDES_CRYP_KeyInitStructure.CRYP_Key2Left = __REV(*(uint32_t*)(keyaddr));
  keyaddr+=4;
  TDES_CRYP_KeyInitStructure.CRYP_Key2Right= __REV(*(uint32_t*)(keyaddr));
  keyaddr+=4;
  TDES_CRYP_KeyInitStructure.CRYP_Key3Left = __REV(*(uint32_t*)(keyaddr));
  keyaddr+=4;
  TDES_CRYP_KeyInitStructure.CRYP_Key3Right= __REV(*(uint32_t*)(keyaddr));
```

```
# Decompiled code TDES key init
  local_4c = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 |
             uVar1 >> 0x18;
  uVar1 = puVar3[1];
  local_48 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 |
             uVar1 >> 0x18;
  uVar1 = puVar3[2];
  local_44 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 |
             uVar1 >> 0x18;
  uVar1 = puVar3[3];
  local_40 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 |
             uVar1 >> 0x18;
  uVar1 = puVar3[4];
  local_3c = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 |
             uVar1 >> 0x18;
  uVar1 = puVar3[5];
  local_38 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 |
             uVar1 >> 0x18;
```

In AES function the key is gotten from the address of keyStruct at four offsets, differently from the DES functions. The pointer to keyStruct is dereferenced, giving us the stored value in keyStruct at the offsets 0, 0x4, 0x8, and 0xC. The space between these addresses is four bytes. This can also be read directly by the decompilation since it is a cast to the uint datatype which also stores 32-bits, i.e 4 bytes. These four sections of four bytes is equal to the 128-bit AES key.

```
# STM32 cryp library AES128 key init
    AES_CRYP_KeyInitStructure.CRYP_Key2Left = __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key2Right= __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key3Left = __REV(*(uint32_t*)(keyaddr));
    keyaddr+=4;
    AES_CRYP_KeyInitStructure.CRYP_Key3Right= __REV(*(uint32_t*)(keyaddr));
```

```
# Decompiled code AES128 key init
        uVar1 = *(uint *)keyStruct;
    local_54 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 |
               uVar1 >> 0x18;
    uVar1 = *(uint *)(keyStruct + 0x4);
    local_50 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 |
               uVar1 >> 0x18;
    uVar1 = *(uint *)(keyStruct + 0x8);
    local_4c = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 |
               uVar1 >> 0x18;
    uVar1 = *(uint *)(keyStruct + 0xc);
    local_48 = uVar1 << 0x18 | (uVar1 >> 8 & 0xff) << 0x10 | (uVar1 >> 0x10 & 0xff) << 8 |
               uVar1 >> 0x18;
```

The cryptographic core definitions for TDES and AES in the reference manual state that they share the K2 and K3 key registers. Since the decompiled AES function gets its key from the address of keyStruct, we should be able to read the values of the K2 and K3 registers from the sram memory. It seems that there is a correlation between the STM CRYP structure and the Key array in the crypStruct with the four byte offsets. Figure 4.22 above shows these values.

If DES is the selected encryption function the keys are read from the K1 registers. We do not know where K1's left and right register might be in the crypStruct object. However, we know that the K2 and K3 registers are next to each other based on the STM encryption function, and based off of the offsets in the decompiled AES code. Thus, the K1 registers might be right before our crypStruct base address, at crypStruct-0x4 and crypStruct-0x8. The eight bytes before crypStruct contain all zeroes. Alternatively, if the K1 registers are positioned after the K3 register in SRAM, that would make no difference as the succeeding eight bytes are also all zeroes. Since the K1 registers are used for the DES key, it seems unlikely that DES is selected in the memory dump. For TDES the K3, K2, and K1 registers are used to store the three keys. However, since we know that the supposed K1 register only contains zeroes it implies that the implemented TDES function only uses two different keys. The TDES cryptographic core documentation states that the cryptograpic processor supports TDES in three different keying options [30]. For TDES, the cryptographic processor can use either three different keys, two different keys (where K3 = K1), or three equal keys which is equivalent to single DES. If the currently selected encryption algorithm is TDES, it uses a zero key or it reuses one of the former keys from K2 or K3. If the AES-128 is chosen, the key would be in the K2 and K3 registers. Due to the implication for DES and TDES of the all-zero K1 registers, it seems likely that the K2 and K3 registers contain the supposed AES-128 key. The hexadecimal values of the K2 and K3 registers can be seen below.

```
AES-128 key: D4 [REDACTED] 00
```

However, if it is the case that they are using TDES with an all-zero K1 register, the K2 and K3 register would contain the same hexadecimal value with 8 bytes each. Since DES only uses 56-bit keys, the last byte is either discarded or used for a parity check.

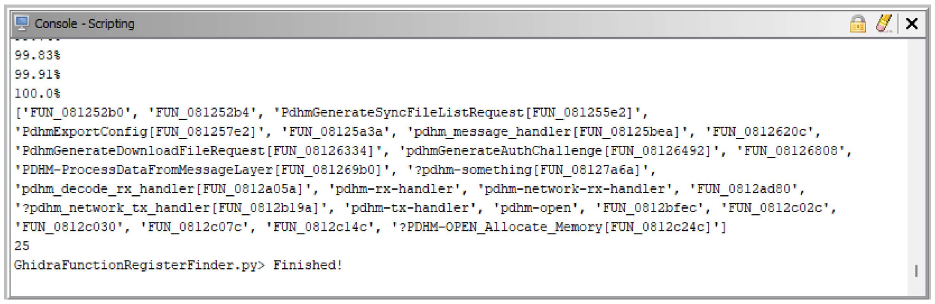
```
Alternatively, TDES keys:
K1: 00 00 00 00 00 00 00 00
K2: D4 [REDACTED]
K3: [REDACTED] 00
```

Since we dumped the memory of three HMUs, all of the same model CardioMessenger Smart 3G, we were able to check the the memory sections for those devices as well. They all contained the same 16 byte hexadecimal value starting at position 0x2000 87F8. The values in the position of the supposed K1 registers were also all zeroes in the memory dumps of all three devices. The hexadecimal values preceding the supposed K1 registers were different for the three HMUs. The memory of the HMU we analyze have many 16 byte lines of all zeroes, while the memory of the two other HMUs have 138 bytes of identical memory (length hex:8A), then their memory were different. The bytes succeeding the K3 registers were all different for all the memory dumps. The last key register in CRYP is the CRYP_K3RR at offset 0x3C. The next register defined in the CRYP object is the first initialization vector register CRYP_IV0LR at offset 0x40 [30]. Each of the initialization vector registers contain 32-bit, 4 bytes, similar to the key registers. Based on the structure of the CRYP object described in the reference manual, and its similarities to the key register structure in the decompiled crypStruct object, we would expect the initialization vector registers to be the next values in memory. However, we cannot conclusively claim at this point that these are in fact the initialization vectors without a proof of concept on the encrypted HMU transmissions. The presumed initialization vector from memory are added below. For the AES-128 key, the initialization vector is also 128-bit in length [30]. A proof of concept is needed to confirm whether the crypStruct is in fact located at the mentioned address in SRAM, due to the inconsistency of the two project decompilations. The many 0 values of the presumed encryption key and initialization vectors suggest that these findings are questionable. It might be the case that the decompilation is misleading in its interpretation, thus the supposed encryption key finding requires a proof of concept to be confirmed.

```
128-bit IV:
00 00 [REDACTED] 20 04 00 00 00
```

4.4.5 PDHM - Communication Protocol

To make sure we have identified all the functions that compose the PDHM communication protocol, we perform another search with our GhidraFunctionRegisterFunder for *PDHM*.



```
99.83%
99.91%
100.0%
['FUN_081252b0', 'FUN_081252b4', 'PdhmGenerateSyncFileListRequest[FUN_081255e2]',
'PdhmExportConfig[FUN_081257e2]', 'FUN_08125a3a', 'pdhm_message_handler[FUN_08125bea]', 'FUN_0812620c',
'PdhmGenerateDownloadFileRequest[FUN_08126334]', 'pdhmGenerateAuthChallenge[FUN_08126492]', 'FUN_08126808',
'PDHM-ProcessDataFromMessageLayer[FUN_081269b0]', '?pdhm-something[FUN_08127a6a]',
'pdhm_decode_rx_handler[FUN_0812a05a]', 'pdhm-rx-handler', 'pdhm-network-rx-handler', 'FUN_0812ad80',
'?pdhm_network_tx_handler[FUN_0812b19a]', 'pdhm-tx-handler', 'pdhm-open', 'FUN_0812bfec', 'FUN_0812c02c',
'FUN_0812c030', 'FUN_0812c07c', 'FUN_0812c14c', '?PDHM-OPEN_Allocate_Memory[FUN_0812c24c]']
25
GhidraFunctionRegisterFinder.py> Finished!
```

Figure 4.23: Results from GhidraFunctionFinder for PDHM

Running the script shows that 25 functions contains the term PDHM. Going through each of the functions, we find functionality related to the network and message layers of PDHM, the ULPAMI- and GSM interface, file installation, hashing, and authentication. In the string analysis we found mentions of a compression header, message layer, transport layer, and network layer. The message layer is part of the function name for PDHM-ProcessDataFromMessageLayer, transport layer was mentioned in the PDHM-RX-Handler, and the network layer is mentioned in a debugging string starting with PDHM-network-RX-Handler. The string findings indicate a protocol with a layered structure similar to the OSI model. Below is a sketch of our initial expectations for the PDHM protocol structure.

```
Expected protocol structure:
|(Network)Transport layer|Encryption layer|Compression layer|Message layer|
```

Now that we have an overview of all the PDHM function we will try to identify the structure and layers of the protocol. We will start by dissecting the PDHM functions initialted by PDHM_OPEN, which is the PDHM-network-rx-handler, PDHM-RX-Handler, and the PDHM-TX-Handler. We will also compare our findings in the Biotronik CardioMessenger Smart 3G with the findings of Bour on the older Biotronik CardioMessenger II-TLine model. The PDHM-Network-RX-Handler, PDHM-RX-Handler, and the PDHM-TX-Handler are added in their entirety in the zip-file attached with this thesis.

PDHM-Network-RX-Handler

The PDHM__NETWORK__RX__HANDLER is the first receiver function to dissect the incoming data. Its purpose is to allocate memory and create a OSTHREADCREATE for the PDHM-RX-HANDLER for further processing. It starts by checking if the crypStruct object is 0 or not - if it is configured correctly. Below is a simplified code snippet of the PDHM-NETWORK-RX-HANDLER.

```
if ((*(byte *)(crypStruct + 0x1ff) & 1) == 1) {
    if (*(byte *)(crypStruct + 0x200) == 0) {
        while(true){
            if (*header[pbStack40] == 5) {
                *(undefined *)(crypStruct + 0x200) = 2;
                break;
            }if (*header[pbStack40] == 0x33) {
                *(undefined *)(crypStruct + 0x200) = 2;
                break;
            }else{
                if ((*header[pbStack40] == 0xaa) && (header[pbStack40][1] == 0)) {
                    *(undefined *)(crypStruct + 0x200) = 1;
                    break;
                }
            }
        }
    }if (*(char *)(crypStruct + 0x200) == '\x01') {
        isMemAlloc = memory-alloc[FUN_081172d8](0xb0);
        if (isMemAlloc == 0) {
            log("PDHM: pdhm_network_rx_handler - malloc error")
        }else{
            isMemAlloc = osThreadCreate[FUN_0811d994]
                        (0,pdhm-rx-handler[FUN_0812a166] + 1,(char)crypStruct,0xbd,crypStruct
                        ,0x2000,*(undefined4 *)(crypStruct + 0xc0),0x9c,
                        (uint)*(ushort *)(crypStruct + 0x1ca),0,0,0,0,0,0);
        }
    }else if (*(char *)(crypStruct + 0x200) == '\x02') {
        isMemAlloc = memory-alloc[FUN_081172d8](0xb0);
        *(int *)(crypStruct + 0xc0) = isMemAlloc;
        if (isMemAlloc == 0) {
            FUN_08129fbc(crypStruct + 0xb0);
        }else{
            isMemAlloc = osThreadCreate[FUN_0811d994]
                        (0,pdhm-rx-handler[FUN_0812a166] + 1,(char)crypStruct,0xbd,crypStruct
                        ,0x2000,*(undefined4 *)(crypStruct + 0xc0),0x9c,
                        0,0,0,0,*(undefined4 *)(isMemAlloc + 0x94),0,0);
        }
      }
    }
}
```

The noteworthy elements of the PDHM_NETWORK_RX_HANDLER are the pointers to a header value in the stack. The first value that is expected in the transmission header is either 0x5, 0x33, or 0xAA. All these three header values call osThreadCreate to start an instance of PDHM-RX-HANDLER, but a couple of the parameters are different for the osThreadCreate for 0xAA compared to the two other headers. The header values define the expected values in the first field in the outer layer. The header 0x5 is also the same header which was found by Bour while analyzing the wireless transmissions of the CardioMessenger II-S TLine [54].

Finding 17. *The HMU's PDHM protocol supports three incoming transport layer headers, 0x5, 0x33, and 0xAA.*

PDHM-RX-Handler

PDHM-RX-HANDLER is the second receiver function for messages to the HMU. It is initiated by an OSTHREADCREATE in PDHM_NETWORK_RX_HANDLER. A selection of its debugging strings are added below. The strings show that this function contains code related to encryption and decompression. Those are the two next expected layers of the protocol. There is also mention of a cyclic redundancy check which means that the transmission might have error-detection. One string also mentions an error based on frame length. In the standardized OSI model, frame is a term used to describe the transmission unit in the data link layer. If this layer is consistent with OSI model data link layer we would also expect to find fields for a source and destination address.

```
"PDHM: pdhm_decode_rx_handler - invalid pointer detected\r\n"
"PDHM: pdhm_decode_rx_handler - DECOMPRESSION error\r\n"
"PDHM: pdhm_decode_rx_handler - ENCRYPTION error\r\n"
"PDHM: pdhm_decode_rx_handler - rx-timeout\r\n"
"PDHM: pdhm_decode_rx_handler - malloc error\r\n"
"PDHM: pdhm_decode_rx_handler - frame length error\r\n"
"PDHM: pdhm_decode_rx_handler - CRC error\r\n"
"PDHM: pdhm_decode_rx_handler - nothing to send\r\n"
"PDHM: pdhm_decode_rx_handler - transport layer expected\r\n"
```

First, the function checks a couple conditions on the incoming crypStruct. The crypStruct variable cannot be null - it needs to be defined. The code then goes into a while-loop that checks for available messages, calls the Decryption-Handler, calls for decompression, and then calling the PDHM_PROCESSDATAFROMMESSAGELAYER function if each step is successful. Below there is a while-loop that checks the header of dataStruct is either 0x5 or 0x33. These are two of the same values we encountered in the former PDHM-NETWORK-RX-HANDLER. Yet, it is interesting that the PDHM-RX-HANDLER has essentially three modes. The first mode is at the top within the crypStruct+0x75==0x1 if-statement. The other two are defined by the header 0x5 and 0x33. There is no mention of the 0xAA header in this function. 0xAA might be related to the first mode but it is not clear from the code. The code snippet below illustrates a simplified overview of the PDHM-RX-Handler.

```
if (crypStruct != (int *)0x0) {
    if(crypStruct+0x7e==0x1){
        do{
            do{
                var = CheckMessageAvailable()
            }while(var!=1)
            if(memory-allocation()!=0x0){
                if(Decryption-Handler(crypStruct,dataStruct)){
                    if(Decompression(crypStruct,(int *)&dataStruct)){
                        PDHM_ProcessDataFromMessageLayer(crypStruct,(int)(crypStruct + 0x2c),(int)dataStruct)
                    }else{
                        log("PDHM: pdhm_decode_rx_handler - DECOMPRESSION error")
                    }
                }else{
                log("PDHM: pdhm_decode_rx_handler - ENCRYPTION error")
                }
            }
        }while(true)
    }
    while(true){
        if (*(char *)local_30 == '0x05')
        if (*(char *)local_30 == '0x33')
        else: "PDHM: pdhm_decode_rx_handler - transport layer expected"
    }
}
```

From the simplified code snippet above we can see that below the two if-statements that check the variable for 0x5 and 0x33, there is an else-clause that logs the error message *transport layer expected*. Hence, the expected values for the header is either 0x5 or 0x33 and these are transport layer headers. No other header values appear to be accepted in the transport layer except for the 0xAA header, which is not present in this function.

To understand each of the layers in the protocol, we need to follow the code that leads to the Decryption-Handler, Decompression, and PDHM-ProcessData -FromMessageLayer functions. The code inside the 0x05 and 0x33 modes have a complex structure of nested conditionals and loops. We mainly focus on the path within each mode that takes us to the data processing functions. It is also important to note that information and configurational data is gotten from the two variable objects crypStruct and dataStruct. We know that crypStruct contains cryptographic information based on the analysis of the encryption functions, and we expect the dataStruct to contain the layered message based on the many code lines that retrieve data from this object. The crypStruct and dataStruct are also the two variables that are always sent to the data processing functions dealing with encryption, decryption, compression, and decompression. This makes sense as we would expect the processing functions to receive both the cryptographic information and the variable containing the data.

In the 0x05 code block there are no calls to the Decryption-Handler, Decompression, nor PDHM-ProcessDataFromMessageLayer. However, it does have functionality to modify values in the stack, and it has debugging strings about PDHM memory allocation and a frame error. It starts off by checking if a local variable is less than 0xB, 11 in decimal, which seems to compare a length variable. If the local variable is not less than 0xB, multiple elements on the stack are set to the value in the position of the transport header, but binary right shifted by 0x8, 0x10, or 0x18, and then cast to an int. It essentially means that one of the stack elements have been given the value at the address the header pointer points to, except the right-most byte. For the second stack object, the two right-most bytes have been removed. The stack variables seem to be given the header and some extra data. Where one, two and three of the right-most bytes have been discarded. The code lines assigning values to the stack variables are listed below. The offsets may also reveal the lengths of the first fields. The header is recognized as the data type char which is one byte in length. The next references offset is 0x8 which means this field is also one byte. The same can be said for 0x10 and 0x18, which are also one byte in length each.

```
uVar3 = *header-transport-layer;
local_3c = (undefined)uVar3;
uStack59 = (undefined)((uint)uVar3 >> 0x8);
uStack58 = (undefined)((uint)uVar3 >> 0x10);
bStack57 = (byte)((uint)uVar3 >> 0x18);
uVar3 = header-transport-layer[1];
local_38 = (byte)uVar3;
uStack55 = (undefined)((uint)uVar3 >> 0x8);
uStack54 = (undefined2)((uint)uVar3 >> 0x10);
local_34 = *(undefined2 *)(header-transport-layer + 0x2);
```

Two of the stack variables are then concatenated with CONCAT11 to append them together. The code above looks to be formating the values of the stack variables to obtain the value at certain offsets. The basic functionality inside the 0x5 code block is to manage variables on the stack and assign these stack values to different offsets in the dataStruct if memory is allocated. It is strange that the 0x5 offset code block in the receiver function does not reference any data processing functions, given that this transport header is known to encapsulate encrypted data in an older version of the protocol [54].

The 0x33 header is the only other accepted header value in the PDHM_RX_-Handler. It looks similar to the code found in the 0x5 code block, however this mode references the Decryption-Handler, Decompression, and PDHM-ProcessDataFromMessageLayer. Before the code reaches the data processing function, it performs similar binary shifting operations on multiple stack variables and concatenations of these byte values. Similarly to the binary shifting operations in the 0x5 block, it seems to obtain values from specific offset. Since we can not see the actual variable values, it is difficult to reverse. Before reaching the data processing functions, the code calls to twice to a function which seems to perform a CRC16 operation. Its code is added below.

```
uint uVar1;

for (uVar1 = 0; uVar1 < param_1; uVar1 = uVar1 + 1) {
  *param_3 = *(ushort *)(_FUN_081252b0 + ((int)(uint)*param_3 >> 8) * 2) ^ *param_3 << 8 ^
             (ushort)*(byte *)(param_2 + uVar1);
}
return *param_3 == 0x[REDACTED];
```

The function in the code snippet above takes three incoming parameters. Param1 seems to be the length of a for-loop. Param2 and param3 is part of the calculation, but param3 is also the assigned variable for the CRC result. Lastly, the function compares param3 to a hardcoded value of two bytes, which seems to be a CRC16 result. If the CRC function is not true, we do not reach the data processing functions and a CRC error is logged. It seems that the transport layer comes with a CRC16 check that needs to be equal to the hardcoded CRC value.

Finding 18. *The HMU performs CRC16 calculations in the transport layer on incoming data and compares to a hardcoded value*

If the result of the CRC function is equal to the hardcoded value, the function proceeds to call the data processing functions, Decryption-Handler, Decompression, and PDHM-ProcessDataFromMessageLayer.

```
# Structure findings so far:
mode 1: |Internal CheckMessage()|Encryption layer|Compression layer|Message layer|
mode 2: |0xAA|+ ?
mode 3: |0x5|+ ?
mode 4: |0x33|Encryption layer|Compression layer|Message layer|
```

Encryption Layer

Within the call to the DECRYPTION-HANDLER, the key- and data structure is sent. Following these parameters into the DECRYPTION-HANDLER, we can see that the byte in dataStruct at offset 0x8 is set as the encryptionType. The byte in offset 0x8 of the data variable therefore defines the selected type of encryption. We have previously found that this value selects one of the encryption algorithms DES-CBC, TDES-CBC, and AES-CBC. The encryptionType is then stored in the offset 0x2A in the dataStruct, and then the pointer to datastruct + 0x8 is added by 2. This pointer is then used to get the input for DES-CBC, and the initialization vectors for TDES-CBC and AES-CBC. Ghidra does not recognize the data type and the length of the initialization vectors are not clear from the code. However, the reference manual states that (T)DES-CBC uses only two 32-bit register CRYP_IV0(L/R), which equals 8 bytes. For AES-CBC both the CRYP_IV0(L/R) and CRYP_IV1(L/R) registers are used, which equals 16 bytes. Since the length of the initialization vectors are not clear from the decompiled code, we assume their fields are equal to the standard specified lengths in the STM32 reference manual [30]. The updated pointer is then getting the input values for both TDES-CBC and AES-CBC. Similarly, we can also see that the length parameter sent to all the encryption functions is defined in dataStruct+0x10. This is recognized as the data type int and uint which is of size 4 bytes. Hence, the length field is 4 bytes for all three encryption functions. For TDES-CBC the input in dataStruct + 8 is added by 8 and the length in dataStruct+0x10 is subtracted by 8 before each encryption call. For AES-CBC the input in dataStruct+8 is added by 16 and the length in dataStruct+0x10 is subtracted by 16. This is consistent with the CRYP_DataType_8b variable found inside the encryption functions. The 6th and 7th bit indicate the data type selection. In DES-CBC and TDES-CBC the sixth and seventh bit is 10 which indicate 8-bit input data, and in AES-CBC bits are 01 indicating 16-bit input data according to the CRYP control register section of the STM32 reference manual [30]. These sizes are consistent with the sizes of the interpreted data types for the variable (dataStruct+10) for each of the encryption functions.

Finding 19. *(T)DES-CBC is used with 8-bit input, and AES-CBC is used with 16-bit input*

```
# Encryption layer
|Encryption Type(6=DES-CBC, 7=TDES-CBC, 8=AES-CBC)|IVs|Input|Length|Padding?|

DES:
|Et=6|IVs|Input|Length|Padding?|
|1B  |8B |?B   |4B    |?B      |
TDES:
|Et=7|IVs|Input|Length|Padding?|
|1B  |8B |?B   |4B    |?B      |
AES:
|Et=8|IVs|Input|Length|Padding?|
|1B  |16B |?B  |4B    |?B      |
```

Compression Layer

If the Decryption-handler returns 1 as the ErrorStatus, the PDHM-RX-Handler calls the Decompression function with the parameters crypStruct and dataStruct. The PDHM-RX-Handler calls the PDHM-ProcessDataFromMessageLayer, if Decompression to returns 1. Decompression can return 1 in two ways. Either if the first header check is false and does not contain 9, or if the deflate function returns 0. The first if-statement in Decompression checks if the first byte in dataStruct is equal to 9. If this is not the case, Decompression returns 1 to the PDHM-RX-Handler and PDHM-ProcessDataFromMessageLayer is called. The header check of the dataStruct is similar to the first dataStruct header check in the Decryption-Handler. In that case 0 is returned to the PDHM-RX-Handler which means that the data is only accepted if the dataStruct is encrypted. Unencrypted data is not accepted, but uncompressed data is accepted if it is encapsulated by an encrypted outer layer. This finding is interesting since the data sent over SMS in the CardioMessenger II HMU was encrypted with DES, and not compressed [48]. However if the first byte header of dataStruct is 9, the Decompression functions allocates memory and calls the Deflate function. The parameters sent to the Deflate function are the address of the header containing 9, 0x1F, a string *1.2.1*, and the value 0x38. The first checks in the Deflate function checks if the version parameter is not 0, that the version has to be 1.2.1, that the flags are defined as 0x38, and that the header is not 0. The Decompression function also call another function which contain many debugging strings.

Finding 20. *The compression layer header contains the value 9 if a compressed payload is present*

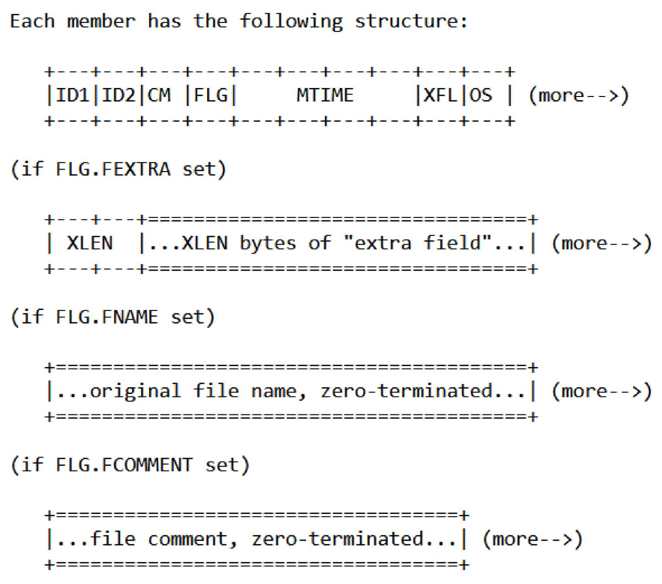
Finding 21. *The PDHM protocol requires an encrypted layer, however compression is optional*

```
# Line 68-72:
            if (((*(uint *)(pbVar7 + 8) & 2) == 0) || (uVar9 != 0x8b1f)) {
          *(undefined4 *)(pbVar7 + 0x10) = 0;
          if (((*(uint *)(pbVar7 + 8) & 1) == 0) ||
             (uVar3 = ((uVar9 << 0x18) >> 0x10) + (uVar9 >> 8), uVar3 != (uVar3 / 0x1f) * 0x1f)) {
             param_1[6] = (byte *)s_incorrect_header_check_0813a6c4;

# Line 123-124:
            if ((*(uint *)(pbVar7 + 0x10) & 0xff) != 8) {
                param_1[6] = (byte *)s_unknown_compression_method_0813a6dc;
```

In the first line of the code snippet above there is a check for 0x8B1F. 0x1F is the first identifier in the gzip wrapper and 0x8b is the second identifier. The first and second identifier combined is the gzip magic header [2]. The code snippet of lines 123-124 show that an error message is thrown if the compression method is not 8. Stating *unknown compression method.* In the gzip wrapper, 8 identifies the deflate compression algorithm. Hence, deflate is the only supported compression method. Since 9 in the header is a necessity to start the DECOMPRESSION function, it seems likely that it is a value to specify that the following data is compressed. The gzip documentation defines a lossless compressed data format. The string 1.2.1 is the same string we found earlier in the preliminary string analysis. It defines the version 1.2.1 of deflate and inflate. These algorithms are part of the zlib compression library, and its authors are also behind the gzip compression utility [49]. The last parameter 0x38, 00111000 in binary notation, defines the compression flags. The set bits are in position 2-4 which means that the compressed data also has a FEXTRA-, FNAME-, and FCOMMENT field [2].

Finding 22. *The zlib compression library is identified by the gzip wrapper and its magic header*

Finding 23. *Deflate is the only supported compression method*



```
Each member has the following structure:

   +---+---+---+---+---+---+---+---+---+---+
   |ID1|ID2|CM |FLG|     MTIME     |XFL|OS | (more-->)
   +---+---+---+---+---+---+---+---+---+---+

(if FLG.FEXTRA set)

   +---+---+=================================+
   | XLEN  |...XLEN bytes of "extra field"...| (more-->)
   +---+---+=================================+

(if FLG.FNAME set)

   +=========================================+
   |...original file name, zero-terminated...| (more-->)
   +=========================================+

(if FLG.FCOMMENT set)

   +===================================+
   |...file comment, zero-terminated...| (more-->)
   +===================================+
```

Figure 4.24: Gzip file structure [2]

Additionally to the wrapper identifiers, the compression mode, and the flags, the wrapper has an extra flag parameter that we have not been able to find. This extra flag is either defined as 2 for maximum compression or defined as 4 for fast compression. We were not able to find which mode is used or if they are both supported. We did not find the OS identifier either. Since the FEXTRA flag is set there should also be two sub identifier fields, a length field, and a subfield for the extra data. There should also be an original file name in the defined FNAME field which is zero-terminated. The FCOMMENT was also set and there should be a comment field which is also zero-terminated. The compressed data wrapper ends with a CRC32 value of the uncompressed data, and an input size field containing the size of the uncompressed data modulo 2^{32} [2].

```
Compression layer:
|Header|ID1 |ID2 |CM|FLG     |MTIME|XFL|OS|FEXTRA|FNAME|FCOMMENT|Compressed msg|CRC32|ISize|
|9     |0x1F|0x8B|8 |0011100|     |   |  |      |     |        |              |     |     |
|1B    |1B  |1B  |1B|1B      |4B   |1B |1B|2B+?B |?B   |?B      |?B            |4B   |4B   |
```

If the first byte in the compression layer data is 9, all the compression fields above should be present. However, if the header is not 9 DECOMPRESSION also returns 1. We believe this value defines the presence of a compressed message and its absence means an uncompressed message. Based on this, the compression seems to be optional for the receiver function.

Message Layer

If the decompression is successful and returns an ErrorStatus of 1, the PDHM-RX-Handler calls the function PDHM-ProcessDataFromMessageLayer with the crypStruct and dataStruct as parameters. This function is the last of the nested data processing functions called from the PDHM-RX-Handler, and should contain the remaining layer and data management.

The first check in PDHM-ProcessDataFromMessageLayer is on the crypStruct and dataStruct. The code essentially checks whether the header is equal to 0x65 (/'e'), or otherwise if the header equals 0xA. These are the two only supported modes in the PDHM-ProcessDataFromMessageLayer, else the function logs an error stating *unknown layer* and returns 0. This means that there is another inner layer encapsulating the message, which has not been identified by previous research.

Finding 24. *The PDHM protocol supports two message layer headers, 0x65 and 0xA.*

The first mode starts with a check on the header. If it is not equal to 0x65 it logs an error message saying *unknown layer* and returns 0. Otherwise the function enables the crypto module clock perform and calls to a hashing function. This function contains a call to the hashing processors status registers and to the hash digest registers. Comparing this function to the hash functions in the STM32 peripheral library we find that this is the HASH-SHA1. The microprocessor supports both MD5 and SHA1 with and without HMAC, but the implemented function has only three input and is identical in structure to the SHA1 function. The HASH-AlgoSelection and HASH-Algomode is all zeroes, which proves that the function is in fact SHA1 without HMAC according to the HASH processor documentation [30]. The 8-bit HASH-Datatype flag is 00100000 which defines 8-bit data written into the HASH data in registers.

Finding 25. *The HMU supports the hashing algorithm SHA-1*

The next function call is to a function we have identified to be CheckSignature. It takes similar input parameters as the HASH-SHA1 function and contains calculations with binary operations. The Function Call Trees feature in Ghidra shows that this function is called by five other functions. One of them is the function Message-InstallFile. This function contains many debug strings that reveal the naming convention for CheckSignature, and itself as Message-InstallFile. We will get back to Message-InstallFile in section 4.4.11 of the additional results. If CheckSignature returns 0 PDHM-ProcessDataFromMessageLayer also returns 0. The function is dependent on the follow code from CheckSignature to execute and return 1 or -1.

```
if ((((uint)param_1 | (uint)param_2) & 3) == 0) {
  while (3 < param_3) {
    uVar4 = *param_1;
    param_1 = param_1 + 1;
    uVar3 = *param_2;
    param_2 = param_2 + 1;
    param_3 = param_3 - 4;
    if (uVar4 != uVar3) {
      if ((uVar4 << 0x18 | (uVar4 >> 8 & 0xff) << 0x10 | (uVar4 >> 0x10 & 0xff) << 8 |
          uVar4 >> 0x18) <=
          (uVar3 << 0x18 | (uVar3 >> 8 & 0xff) << 0x10 | (uVar3 >> 0x10 & 0xff) << 8 |
          uVar3 >> 0x18)) {
        return -1;
      }
      return 1;
    }
  }
}
```

The CheckSignature function starts with a binary OR operation on param1 and param2. We need this calculation to be equal to 0 in order for the CheckSignature function to return 1 or -1. If the returned ErrorStatus from CheckSignature is not 0, the function logs the error message *MSG Hash error* and returns 0. This seems like a proprietary hashing function. The following function call is again to the CheckSignature function, but with parameters that are variables in dataStruct and crypStruct structures. This seems to perform a comparison between the hash of the incoming file and data in the dataStruct.

The remaining code in the 0x65 block has a check on some malformed decompiled data to check the direction and frame length. There is a while-loop that allocates memory and adds processed data to the function add-msg-out-list. The function add-msg-out-list is called by many functions related to PDHM, and error strings show that it is supposed to load or add a message to an export list for outgoing messages.

In the other mode, the header is checked and if it does not equal 0xA the function logs *unknown layer* and returns 0. Then there is a call to a CRC function. It is similar to the CRC16 function we found in the PDHM-RX-Handler except that this function only takes two input parameters, length and input. The result variable is initiated and declared to 0 inside the function before the computation. The result is compared to the same hardcoded hexadecimal CRC16 number from the previous CRC16 function in the transport layer. Hence, there is a CRC check on the message format to confirm its validity.

Finding 26. *There is a CRC16 check on the data within the message, that is compared to a hardcoded 2 byte value*

The 0xA block then has an if-statement with two binary shifting operations and comparisons that the data stored in input2, which is (dataStruct + 8). If the result of (input2 » 4 !=5) (input2 » 4 !=3) is true, then we either get a *invalid source* or *invalid target* error message. The input2 has a cast to uint and int which stores 4 bytes. Since we do not have access to the values in these four bytes we cannot know how these operations show anything related to a source or target. It is possible that there are addresses or identification numbers after the header and length fields, but this is an educated guess based on the serial number field found in the protocol used for an older HMU [54]. Based on the debugging strings there appears to be some logic to handle valid source and target of the data from the data transmission.

There is another call to the MSG-CRC16 function with a related error message in the else-clause that states *invalid container.* The inputs to the CRC function is again compared to the same hardcoded hexadecimal number we found earlier. The last parts of PDHM-PROCESSDATAFROMMESSAGELAYER are calls to a couple functions and adding crypStruct data to the ADD-MSG-OUT-LIST.

PDHM-TX-Handler

PDHM-TX-HANDLER is the transceiver function for the HMU's outgoing messages. It is called directly in PDHM_OPEN with a OSTHREADCREATE call. To get an initial overview of the PDHM-TX-HANDLER and its functionality we look at its debugging strings.

```
"PDHM: pdhm_decode_tx_handler - adding frame: frm=%d, len=%d\r\n"
"PDHM: pdhm_decode_tx_handler - skip result: dropping frm=%d\r\n"
"PDHM: pdhm_decode_tx_handler - frame error\r\n"
"PDHM: pdhm_decode_tx_handler - no message available\r\n"
```

The PDHM-TX-HANDLER starts by checking if the variable CRYP_KeyInitStructure is equal to 0, and sets the ErrorStatus return value to 0. The function starts by checking if the char value, one byte, in crypStruct+0x81 is not equal to 0. If this is true, then the function checks for available messages and tries to allocate memory. If memory is allocated, the functions sets values into the dataStruct array at different indexes. The functions does not clearly state the values inserted into dataStruct, but it is clear that this is the data which is encapsulated in the following calls to the COMPRESSION and ENCRYPTION-HANDLER functions.

The following code section in has two similar code blocks, which can be seen below. A distinct difference between them is the values in line 6 and line 20 of the code snippet. There are references to two of the transport headers we found in the receiver function. It looks like the HMU is capable of constructing data transmissions with two transport headers, while it accepted three types.

```
1   if (*(char *)(crypStruct + 0x6c) == '\x01') {
2       uVar7 = dataStruct[4] + 0xc;
3       iVar4 = dataStruct[2];
4       dataStruct[2] = (undefined *)(iVar4 + -0xc);
5       dataStruct[4] = uVar7;
6       *(undefined *)(iVar4 + -0xc) = 0x33;
7       *(uint *)(iVar4 + -0xb) =
8            uVar7 * 0x1000000 | (uVar7 >> 8 & 0xff) << 0x10 | (uVar7 >> 0x10 & 0xff) << 8 |
9            uVar7 >> 0x18;
10      *(undefined *)(iVar4 + -7) = *(undefined *)((int)dataStruct + 0x21);
11      *(char *)(iVar4 + -6) = (char)((uint)*(ushort *)(crypStruct + 0x73) >> 8);
12      *(char *)(iVar4 + -5) = (char)*(undefined2 *)(crypStruct + 0x73);
13      *(short *)(crypStruct + 0x73) = *(short *)(crypStruct + 0x73) + 1;
14      *(uint **)(iVar4 + -4) = crypStruct[8];
15  }
16  else {
17      iVar4 = dataStruct[4];
18      iVar5 = dataStruct[2];
19      puVar6 = (undefined *)(iVar5 + -10);
20      *puVar6 = 0x5;
21      *(char *)(iVar5 + -9) = (char)((uint)(iVar4 + 0xc) >> 8);
22      *(char *)(iVar5 + -8) = (char)iVar4 + '\f';
23      *(undefined *)(iVar5 + -7) = *(undefined *)((int)dataStruct + 0x21);
24      *(char *)(iVar5 + -6) = (char)((uint)*(ushort *)(crypStruct + 0x73) >> 8);
25      *(char *)(iVar5 + -5) = (char)*(undefined2 *)(crypStruct + 0x73);
26      *(short *)(crypStruct + 0x73) = *(short *)(crypStruct + 0x73) + 1;
27      *(uint **)(iVar5 + -4) = crypStruct[8];
28      local_2c = FUN_081251a8(iVar4 + 0xaU & 0xffff,puVar6);
29      puVar6[iVar4 + 0xaU] = (char)((uint)local_2c >> 8);
30      puVar6[iVar4 + 0xb] = (char)local_2c;
31      dataStruct[2] = puVar6;
32      dataStruct[4] = iVar4 + 0xc;
```

In the code snippet we also see that the code inside the if-statement, that is for the 0x33 header, has a field which seems to perform a hashing or cryptographic operation in line 7-9. This is not present for the 0x5 in the else-clause. Which means that the 0x33 transmission might contain an additional field of a hash or something cryptographic which is not present in the 0x5 transmission. Both of the transmission types also have a counter variable in lines 13 and 26 for 0x33 and 0x5 respectively, in the above code snippet. Ghidra recognizes this variable as a short data type, which has a length of 2 bytes. A counter field was also found in the protocol of the older HMU, however they found it had a length of 3 bytes [54]. The decompiled code suggests that the data transmissions of PDHM protocol in this HMU can count packets to the number 65535 - less than the previous protocol. It is also interesting to note that the data is compressed and encrypted in the PDHM-TX-Handler before the two modes, 0x33 and 0x5, in the code snippet is selected. In the PDHM-RX-Handler, only the 0x33 header had code calling decryption and decompression. Hence, the 0x5 header is encapsulating encrypted and compressed data in the outgoing transmission. The PDHM-RX-Handler receiver function did not call the data processing functions in the 0x5 code section, however it did seem to add the data to allocated memory. The first code block in PDHM-RX-Handler did not look for a transport header, and checked for available messages internally. Since there is a CheckMessage function call on the top of the receiver function, I think it is likely that the data encapsulated by the transport headers 0x5 and 0xAA might be processed there after they have been allocated memory.

The version of the protocol found on the older HMUs also had the same 0x5 transport header, and it encapsulated encrypted data. It would seem likely that this is also the case for the data encapsulated in the protocol found in the CardioMessenger Smart 3G. It also seems reasonable to assume that the incoming 0x5 transmission is similar to the outgoing 0x5 transmission. The decompilation might also have interpreted the code inaccurately, and therefore not shown the 0x5 calls to decryption and decompression in the receiver function. It is also interesting to note that the HMU supports three incoming transport layer headers, but only seems to be capable of transmitting with two of the headers. Finally, the PDHM-TX-Handler calls the add-msg-out-list function with crypStruct and the memory allocation variables to load them into the export list.

Finding 27. *The HMU's PDHM communication protocol support two of the three transport layer headers for outgoing transmissions*

The PDHM Communication Protocol of the Biotronik CardioMessenger Smart 3G

This is our final illustration of all the known layers and fields of the PDHM protocol in the Biotronik CardioMessenger Smart 3G version 1.20.

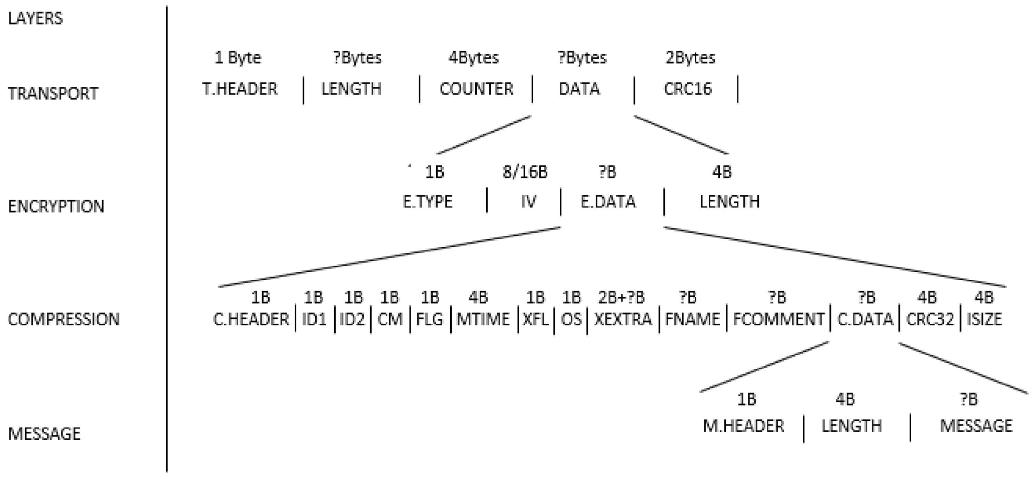


Figure 4.25: The identified layers and fields of the PDHM communication protocol

The known length of each field is written in bytes. The letters in front of the headers and data fields are shortened, and represents transport, encryption, compression, and message. The size of the IV field in the encryption layer is dependent on the selected encryption algorithm. (T)DES uses 8 bytes, and AES uses 16 bytes. There might also be a padding field in some of the layers similarly to the older protocol, but that was not clear from our reverse engineering. The reverse engineering did also not show that there is a source or target address, but the PDHM_ProcessDataFromMessage layer did have a computation of the second byte of dataStruct that would log invalid source or target. It did not compare to hardcoded values from a specific field. We also do not know the specific use cases for each of the transport headers and message headers. However, based on our reverse engineering we should be able create a valid transmission in structure that will be processed at every layer.

4.4.6 Additional Findings

In the process of identifying the function hierarchy and those related to the PDHM protocol, we also found functions that are part of the HMU's other communication interfaces such as USB, ULP-AMI, and GSM. These interfaces either share a common caller function or depend on shared underlying functions. Even though these interfaces are outside the scope of our research goal, the functions of the other interfaces are occasionally related to the PDHM protocol.

4.4.7 USB Interface

The extent of our analysis have not shown that the USB interface is used for communication. We have found that the function in charge of monitoring and managing the charging of the HMU, is the Chg_Handler. We have also been able to identify the names of many of the underlying charging functions based on their many debugging strings. Of the debugging strings in the Chg_Handler there appears to be functionality to detect abnormal charging, changing the input current, checking the status and state of the battery and the charging, and fault recovery.

4.4.8 ULPAMI Interface

We have also been able to identify many of the functions related to the ULP-AMI interface. This is the interface between the HMU and the IMD. We identified the UlpamiStartRx function which initiates the receiver function Ulpami-RX. This function contains debugging strings about an ULPAMI-string, a CRC, Raw-data, RSSI, syncflags, ULPAMI termination, and a call to the Ulpami-RX-config function that sets RSSI and frequency offset. However, these values are not hardcoded anywhere in the decompiled code. We also identified the transceiver function ULPAMI-TX function. The ULPAMI interface has a search mode function ULP_SM that initiates the ULP function and the waiting function ULP-WM. The ULP function contains many debugging strings and seems to log all the queried information from the IMD. This fits because ULP is the last function that is called in the searching function ULP-SM. The data is retrieved and logged on the HMU. The waiting mode function ULP-WM is in charge of pairing and has functionality to postpone connection for either 5 minutes or 4 hours. The conditions and extent of the back-off functionality is unknown. There also appears that there are multiple pairing states with the IMD, as well as soft- and hard-pairing. A function call graph of the supporting ULPAMI function can be seen below.

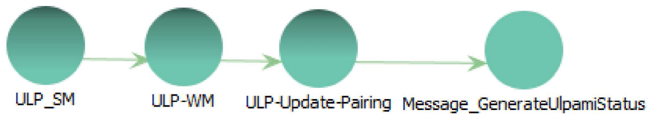


Figure 4.26: ULPAMI function for searching and pairing

4.4.9 GSM Interface

Given the name of the HMU model, CardioMessenger Smart 3G, we would expect to find the third generation telecommunication network interface UMTS. The HMU user manual appendix also specifies that the CardioMessenger Smart 3G supports UMTS W-CDMA. However, we only found telecommunication functionality related to GSM.

The handler function for GSM is the MMI_GSMSTATEHANDLER. The incoming function call trees shows that the GSMSTATEHANDLER can be initiated by many function, and it can be traced up to the USBINTERFACEHANDLER and the MMI-INIT function. MMI-INIT also starts other mmi threads such as the on-screen GSM signal and battery icons. There is also a GSM_SM, i.e statemachine, that deals with power on/off and timeouts. It also calls to other GSM subfunctions such as GSM_STATEMACHINEINIT, which in turn calls GSM_INIT and GSM_PARSER.

We also see the GSM function GSM_HANDLEPDPCONTEXT, but Ghidra is not able to find its incoming call reference. Hence, we do not know where it is initiated. However, it contains a couple interesting debug strings such as *wrong_APN* and *authentication_failure*. It seems that there is some kind of authentication implemented into their GSM functions.

4.4.10 MMI - PhysicianIconHandler

The MMI_PHYSICIANICONHANDLER is the function which we believe to be the *physician callback function* that is described in the HMU's technical manual [9]. It is initiated by a OSTHREADCREATE in the MMI-INIT. The call back functionality is used by the physician to turn on a blinking icon on the HMU display. This icon indicates that the patient should contact their physician.

4.4.11 Updating Feature: InstallFile

While analyzing the CheckSignature function we found in PDHM_ProcessData-FromMessageLayer, we found that it is called from five functions. One of them is called Message_InstallFile. Message_InstallFile is only called from Message_HmscInterfaceHandler which is initiated by an OSThreadCreate in Message_Init, right after the initiation of UsbInterfaceHandler. Message_InstallFile starts by checking if a variable in crypStruct is equal to a hardcoded *magic number*. If this is not the case, the functions logs the error message *"Message_InstallFile: invalid magic number"*. If the magic number is set correctly, then the functions calls HASH-SHA1 and CheckSignature. If the signature check returns 0, the function proceeds to process the incoming file from the PDHM file list. It is also interesting to note that the value in crypStruct is set to 3 if successful, and either 0x81 or 0x82 if the processing fails.

Based on the contents of the Message_InstallFile function, it appears that the HMU has the ability to install files over the PDHM protocol. There is no mention of the file type in the code, and the user manual has no information about file installation. Our assumption is that the file installation is related to a firmware update. However, this There is also a hardcoded magic number which needs to be set at the offset 0xC before the signature is checked.

4.5 Limitations of the Results

Most of our work is done on decompiled code in Ghidra, and this is also where most of the limitation are present. The code we are analyzing is computer generated, decompiled code of the flash memory section in the HMU. This section contains the compiled code Biotronik installed during production. As we discussed in section 3.4, a lot of information is lost in the compilation process and this is apparent when attempting to reverse engineer the firmware. After we extracted the code through the debugging interfaces and imported it into Ghidra, Ghidra's decompiler generated an interpretation of the executed code. The code we are analyzing is therefore far from the original source code developed by Biotronik. This posed several limitations and issues during the reverse engineering process.

One of the limitations was the presence of duplicated functions. In the full flash memory file we found 2 or 3 duplications of many functions, which tripled the workload of the function hierarchy identification. Remembering the three code blocks from the entropy and the duplicated string findings, we assumed that the flash might have three operational modes. However, cutting out one of the code blocks later showed to be a mistake because a few functions in the top of the function hierarchy were removed as well. Some overlapping work was therefore necessary while reverse engineering and occasionally the duplications had minor differences in values or offsets. Certain findings and values may therefore be imprecise. An example of such a difference is the decompilation of the UsbInterfaceHandler in the entire flash file and the decompilation of the one larger code block. The decompilation of the large code block said that the crypStruct was located in the SRAM. By using the offsets, we suggested that the encryption keys and initialization vector was located and hardcoded in memory. However, in the decompilation of the entire flash file there was no reference to a location in SRAM. The imported memory and the decompilation can therefore be inconsistent and misleading. In the case of the suggested encryption keys, this finding is probably not accurate. Since the 16 bytes in the supposed K2 and K3 key registers were identical on all three HMU devices, it would imply that their encryption key is shared between the devices and hardcoded in memory. In a previous private conversation between my supervisors and Biotronik representatives, they claimed that keys were no longer hardcoded in memory. Therefore the suggested encryption key finding is likely an inaccuracy in the decompilation.

While identifying the function hierarchy and all of the PDHM functions, we found that Ghidra was not always able to find the incoming function call references. An example of this is how we found the incoming call to the PDHM_RX_Handler. Thankfully, Ghidra has a feature called *Search Program Text* which can find strings, comments, and instructions. Ghidra also has the ability to find references to a specific memory address which is how we found the incoming call to Message_InstallFile from the Message_HmscInterfaceHandler. Sometimes Ghidra does not find the incoming reference call in the Function Call Trees feature, however it is able to lookup calls to a memory address.

The decompilation does also not always accurately represent the data types of each variable. In our analysis of the PDHM protocol we tried to identify the fields in each layer, the the sizes of each field. Since Ghidra interprets the data types, and often performs multiple casts between data types which can be confusing, it can be difficult to determine the fields lengths and exact position compared to the other fields in the layer.

4.6 Conclusion of Analysis

The end goal of our analysis as we defined in the scope in section 1.2, was to find and document the functionality of the HMU's communication protocol to the data servers. We achieved this goal by analyzing the code from the memory files and focusing on the cryptographic implementations. This was our defined research objective RO1. With this approach we were able to identify the cryptographic functions which enabled us to explore the function hierarchy, and find the PDHM communication protocol functions. By following the objective stated in RO2, we found similarities in the layered structure of the protocol and one identical transport layer header. We also found similarities in compression and encryption standards. In our analysis we found added functionality of the PDHM protocol, which has not been identified by previous research. It is therefore unknown whether the communication protocol had these operational modes in the previous HMU models CardioMessenger II-S and TLine, or if it is an updated version of the protocol. There has not been performed a reverse engineering of the previous models similarly to my research on this HMU model. Therefore a similar reverse engineering approach is necessary to confirm if other operational modes exist in the older HMU models. The PDHM protocol in the CardioMessenger Smart 3G has additional header values in the transport and message layer, compared to those found on the older HMUs. These headers seem to indicate that the HMU supports different transmission modes. However the explicit use cases for each mode is unknown, and an experimental proof of concept is needed to explore their operational differences. Still, our analysis reveals the specific fields and their accepted values that is necessary for the HMU to process an incoming transmission.

Throughout our analysis we found vulnerabilities and indications of plausible security weaknesses, that might impact the HMU through its PDHM communication protocol. Since much of the internal data processing in the HMU is unknown, proof of concepts are needed to verify if the HMU is susceptible to these vulnerabilities. Since the vulnerabilities uncovered in our analysis are not conclusively proven to adversely affect the HMU, a CVD process was not deemed necessary by my supervisors at this time. An example of a proof of concept would be to transmit a customized compression payload, similar to the CVEs we discovered, and observe if the HMU crashes. The implications of the findings are explored in the mitigations chapter 5, and scenarios in section 6.1.2. They answer the remaining objectives stated in RO2 and RO3.

Chapter 5 Mitigation

5.1 Mitigation of Findings

The results of our security analysis on the CardioMessenger Smart 3G show that it has inadequate security measures. The HMU is easily obtained online from a multitude of marketplaces, its PCB and components are easily accessible, and its internal security measures in the software is not conforming to modern best-practice. Our findings show that the HMU is vulnerable in several aspects. The HMU should be updated and replaced with new hardware and hardware protection mechanisms, and a software update addressing the described vulnerabilities below. The following sections describe the necessary mitigation in detail.

5.1.1 Mitigation of Hardware

Disable Debugging Interfaces

Our entire project is based on retrieving the memory files from the HMU. Having access to the debugging interfaces JTAG and SWD enables us to connect to the microcontroller and access its memory sections. Without the access through these interfaces we would not be able to analyze and reverse engineer the firmware from the flash memory. The debugging interfaces should be disabled post-production. This can be achieved either in software or physically.

Use Trusted Platform Module

Using a hardware-based secure crypto-processor such as a TPM, will be able to securely generate, store, and handle cryptographic keys, certificates, and operations [58]. A hardware-based TPM also usually has a tamper-proof physical design. The TPM has a unique RSA key which can be used for device authentication. The TPM can also be configured to detect if valid software is used when the system boots.

5.1.2 Mitigation of Software

Memory Protection

Since the debugging interfaces were enabled, it enables us to dump the code from multiple memory sections. Another form of mitigating this procedure is by implementing memory protection. STMicroelectronic's System Memory Protections provide this exact functionality. It provides read and write protection on the flash, sram, and registers to prevent dumping of code, accidentally erased data, and protects their intellectual property [59].

The firmware should also be encrypted. Since the debugging interfaces were accessible and the memory was unencrypted, we were able to reverse large parts of the firmware. Encrypting the firmware with keys stored in the previously mentioned TPM chip, would complicate any effort to reverse the firmware. This is a layered approach to security.

Discontinue the use of DES & TDES

After initially discovering the low-level function related to the cryptographic processor, we discovered that the firmware supported three encryption standards - DES, TDES, and AES. DES is insecure in modern applications due to its short key size of 56-bit [60]. It has also been discontinued as a standard by NIST. TDES and AES are the only two block ciphers approved by NIST. NIST also emphezises that in TDES, three unique keys shall not be used to encrypt more than 2^{20} blocks of 64-bit data [61]. AES is therefore the preferred encryption standard.

Update the Compression Library

The firmware uses version 1.2.1 of the Zlib compression library which has two known CVEs. The consequence of these CVEs has the potential to cause Denial-of-Service on the HMU [50]. Hence, the compression library should be updated to the latest stable version.

Use Digital Signatures

The firmware has a function called MESSAGE_INSTALLFILE which checks the input for a magic number in the header, calculates the SHA-1 hash of the input, and then call the CHECKSIGNATURE function. Based on the contents of the CHECKSIGNATURE function it seems that they are calculating another hash value instead of checking a digital signature on the downloaded file. We encourage the use of digital signatures which can be used to verify the authenticity of the sender, and the integrity of the received file. The previously mentioned TPM should also be useful to verify the provided digital signature of the received file.

ULPAMI Interface: Implement Encryption & Whitelisting

In our analysis we did not find any encryption on the transmissions over the ULPAMI interface. Based on the string analysis of the ULPAMI functions, the data transmissions appear to be sent *raw* with an appended CRC field. To protect the privacy of the patient, the monitoring data of the IMD should be encrypted before it is transmitted. The IMD has limited battery capacity, therefore a trade-off on complexity versus power consumption needs to be assessed by the manufacturer. The HMU and IMD should also implement whitelisting as a feature of its pairing functionality. Whitelisting will exclude invalid interactions that may excessively drain the battery power.

Migrate to SHA-2 or SHA-3

In the firmware we found the hashing algorithm SHA-1. It is used at three different stages of the PDHM functions and for the MESSAGE_INSTALLFILE function. It is used closely with the implemented CHECKSIGNATURE function, and we suspect that SHA-1 is somehow used in their proprietary signature checks. SHA-1 has been deprecated due to a demonstrated collision and successful brute-force attack, and it is not allowed to be used for digital signatures by NIST [62]. NIST recommends the migration to SHA-2 or SHA-3. These hashing algorithms can be used with digital signatures.

Transport Layer Security: TLS or SSH

The communication protocol between the HMU and the data servers consists of multiple layers with identifiers in the header field of each layer. The only security measure of the protocol is the selected encryption type on the encapsulated data. To improve the security of the protocol we suggest an implementation of SSL/TLS or SSH to prevent eavesdropping and message tampering.

Mutual Authentication

In our analysis we did not find an implementation of mutual authentication. However, we were not able to determine the functionality of the different transmission modes, and these might have functionality that performs authentication. Since the presence of authentication is unknown and that the HMU sends sensitive data in its transmissions, it is essential to implement mutual authentication. For the data server it is equally important to authenticate that a transmission originates from a valid HMU. Therefore we suggest that mutual authentication is implemented in the HMU's communication protocol. This can also be achieved through the implementation of SSL/TLS or SSH.

Implement Intrusion Detection System

The devices of the pacemaker ecosystem is highly sensitive. The main task of the HMU is to relay the monitored data that is transmitted from the IMD towards the data servers. Based on the many research papers that have been published in the recent years, discovering vulnerabilities in most devices in the pacemaker ecosystem, we suggest that the HMU is implemented with an intrusion detection system (IDS). An attack on the HMU can potentially cause a Man-in-the-middle on the IMD. To prevent the HMU of becoming an attack surface, we suggest that it is equipped with an IDS that can detect anomalies in its incoming transmission. By extension, the IDS could also detect malicious activity and anomalies in the IMD's behavior.

This could be an addition to the HMU's physician callback functionality. Where an icon is displayed and blinking on the HMU's display to notify the patient to seek medical aid.

5.1.3 Mitigation in Development/Business-level

The following suggestions are mitigations the manufacturer should implement in their continuous development life-cycle. It is unknown whether the manufacturer has implemented any of these systems or not, but based on the lacking security measures we question the effectiveness of their current internal processes.

Implement Threat Identification, Risk Assessment & Vulnerability Scoring System

The pacemaker ecosystem is a sensitive system where devices have a direct impact on a patients health and well-being. As a preemptive measure the manufacturer should conduct a continuous process of threat identification to monitor the current threat landscape and modern major trends. This also includes assessing the emerging CVEs and their impact on their systems. The ability to identify threats before they can impact the ecosystem will improve the safety of the patients overall.

The manufacturer should also implement a risk assessment framework to ensure that they are analyzing and evaluating the risks they are facing throughout their device ecosystem. The risk assessment should include assessment of their network infrastructure, the personal monitoring devices, and their software. The risks should be associated with the loss or impact of a security criteria such as those describes in section 3.2.

When a vulnerability is found within the ecosystem, the manufacturer should use a vulnerability scoring system such as the Common Vulnerability Scoring System (CVSS). The scoring system is used to get a metric value of the severity of a vulnerability. The value can be used by the manufacturer to assess and prioritize the needed mitigation of a vulnerability.

Return Policy of Decommissioned & EOL Home Monitoring Devices

The manufacturer of pacemaker equipment and home monitoring devices should implement a policy for the return of decommissioned and devices that have reached its End-of-Life. The sale of home monitoring devices on public online marketplaces enables access of secondhand sensitive devices which can pose threat to the entire ecosystem, and can be used to perform malicious activities. A return policy of decommissioned home monitoring devices will restrict the ease of access to these devices, thus complicate the process for an adversary's and serve as another preemptive security measure.

5.2 Legislation & Best Practice

In recent years new regulation have been entered into force, which manufacturers of medical devices have to conform to. Additionally, government agencies publish recommendations on security implementation and management which manufacturers should utilize in their development process.

5.2.1 Legal

The MDR and IVDR state that manufacturers of (in-vitro) medical devices need to implement and maintain a risk management system. The systems needs to adhere to security in their product designs, implement adequate security measures for identified threats, and provide detailed information about their products. The manufacturers are also responsible for implementing a quality management system which needs to satisfy various safety and performance requirements defined in the regulations [26, 27]. The regulation also state that the manufacturers need to design the software of their medical devices to ensure performance and reliability, and in accordance with modern development, risk management, and security. The manufacturer also needs to obtain a certification of conformity to sell their medical devices on the European market. If a device is found to not conform to the regulations specified within the MDR or IVDR, they may be withdrawn from market until the necessary corrective mitigations have been performed.

Additionally to the European legislation, the medical devices sold in Norway need to conform to national legislation. The Norwegian Security act states that the undertaking that impacts the basic security of the population shall be responsible for and implement a management system for protective security work [28]. The undertaking is also responsible for implementing an appropriate level of security to reduce the identified risks. They are also required to notify the national security agency (NSA) of suspicious activity that may impact infrastructure or personal security.

5.2.2 Recommendations from Authorities

To ensure the reliability and security of the pacemaker ecosystem, the manufacturer should take advantage of all the available resources published by government agencies, research organizations, and large industry organizations. The European union agency for cybersecurity (ENISA) have published guideline documents for threat and risk management which include a landscape report of the most prevalent threats and trends, good practices to perform risk management, and to responsibly disclose vulnerabilities [1].

The National Institute of Standards and Technology (NIST) have resources such as frameworks, FAQs, events, presentations, and newsroom of the latest updates on their webpage [2]. One of these frameworks are the Cybersecurity framework which has a guide for conducting risk assessments and a risk management framework for information systems and organizations. NIST also has resources such as a privacy framework, cybersecurity guidelines for IoT devices, and guidelines for every aspect of cryptography - e.g cryptographic key management guidelines [3].

The National Security Agency (NSA) publish advisories and guidance documents to handle security controls and commonly exploited CVEs [4,5]. The NSA has an extensive guide for network infrastructure security which should be used by the manufacturers to solidify their private networks[6].

All these resources should be used in the daily operations and every stage of the development cycle. This will ensure security at every layer and mitigate many of the identifiable threats. Pursing security by design is also a great approach to save time and resources in the long run.

[1] Available at https://www.enisa.europa.eu/topics/threat-risk-management

[2] Available at https://www.nist.gov/cyberframework/identify

[3] Available at https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-5/final

[4] Available at https://media.defense.gov/2022/May/17/2002998718/-1/-1/0/CSA_WEAK_SECURITY_CONTROLS_PRACTICES_EXPLOITED_FOR_INITIAL_ACCESS.PDF

[5] Available at https://media.defense.gov/2022/Apr/27/2002984949/-1/-1/0/JOINT_CSA_2021_ROUTINELY_EXPLOITED_CVES_20220427.PDF

[6] Available at https://media.defense.gov/2022/Jun/15/2003018261/-1/-1/0/CTR_NSA_NETWORK_INFRASTRUCTURE_SECURITY_GUIDE_20220615.PDF

Chapter 6 Discussion

6.1 Implications of our work

The following sections summarizes and highlights the implications of our findings. Firstly, we discuss the applicability of our scripts and their contribution to the reverse engineering community. We have described some attack scenarios that are possible on the HMU and some attack scenarios that are theoretically plausible, based of our findings in the HMU analysis. Most of the attack scenarios are related to findings from the software because the majority of our analysis was focused on the decompiled code. The severity of our findings are reflected and described in the proposed scenarios.

6.1.1 Contributions

Our main contributions to the reverse engineering community are the two scripts, our custom version of the svd-loader and the GhidraFunctionFinder.

Developed Scripts to Ghidra's open-source Community

The SVD-loader we developed is customized to automatically define peripherals and all their registers at their defined memory locations in the Ghidra project. The SVD-loader is applicable for any Ghidra project that analyzes an ARM microprocessor and that has a defined .svd (System View Description) file in the XML format. It is therefore useful for most projects related to an ARM microprocessor, not just the STM32F417 which we analyzed. Being able to define the peripheral and register names within Ghidra eases the reverse engineering process, since it removes the need for a continuous lookup in the reference manual and datasheet.

The GhidraFunctionFinder is also applicable for most Ghidra projects. We used it to locate the CRYP register such that we could identify the low-level cryptographic functions, and we used it to locate all the decompiled functions that contained the sting *PDHM*. The script decompiles and interprets every function as a long string, and it searches through every function for every term in our defined search list. In our case the list only contained the *CRYP* or *PDHM*, however it is applicable for more multiple search terms. The use cases for the script is thus far larger than our requirements in this project. The script could even be used to search for an entire line of code within a function, or or multiple programming terms. The script does not modify any of the functions, and can therefore be used at all stages throughout the reverse engineering. The script can therefore be used to locate functions with user-defined variables and function names. The script prints its progress as a percentage continuously in the console, and outputs a list of the function names that matches the search terms to the Ghidra console.

The applicability of the two script aided my reverse engineering process, and they will hopefully be a valuable tool for further research at the Pacemaker Project at SINTEF and NTNU, but the scripts will also be published on Github as a contribution for the open-source Ghidra reverse engineering community.

6.1.2 Attack Scenarios

The following subsection defines the currently possible attack scenarios on the CardioMessenger Smart 3G version 1.20, that are possible based of the findings in my analysis.

Connecting to the Private Data Servers

In the preliminary string analysis of the memory files we found that the memory was unencrypted, and that naming convention was clearly visible from the debug/error logs. Among these strings were also credentials such as a serial number, username, password, and a pin code for the device which was hardcoded in memory. The first attack scenario is based on the available credentials to connect to the Biotronik's private data servers. The HMU uses these credentials to authenticate against the server and to deliver the transmitted user monitoring data, which will be available for the patient's physician. Using the HMU's credentials we should be able to connect and authenticate with the manufacturers data servers. However, we do not know the extent of the HMU's capabilities connecting to the manufacturer's data servers. However, even if the credentials are still valid, we might not be able to connect to their servers since the HMU's SIM card might not be accepted anymore [24]. If we were to connect to the data servers with the obtained credentials, that would be unethical and illegal - considered as a data breach and hacking. It would also be outside the mandate and scope of our research purposes.

Passive Communication with the HMU - i.e Eavesdropping

In our analysis we discovered the layered PDHM protocol and its expected header values, as well as some of the other expected fields in each layer. We also know that the HMU communicates its PDHM protocol transmissions over the GSM interface. Previous research in home monitoring devices have shown that the GSM interface is susceptible of eavesdropping attacks, which enables us to passively listen to the HMU's transmissions and analyze the data contents of its transmissions. A simulated eavesdropping attack with an illegitimate base station can uncover the remaining fields of the layered PDHM protocol, and serve as a proof-of-concept to reproduce and validate my findings. However, this is dependent on finding the encryption key. A real-world eavesdropping attack can enable an adversary to obtain the monitored data from the patient's HMU and their IMD. Thus, having access to the personal health data breaching their privacy.

Active Communication with the HMU

Our findings also enable the scenario of multiple active attacks against the HMU. Since we know the internal code structure and conditionals of the receiver functions, we should be able to create a customized transmission to the HMU that will be accepted and processed.

The first active attack scenario is the ***battery depletion attack***. Unlike older HMU devices, this HMU can be carried around in the patient's daily life similarly to a regular smart phone. The CardioMessenger Smart 3G is a mobile device, and can be used either in stationary operation or mobile operation [9]. If it is used in mobile operation mode without being connected to the power brick, the HMU runs on its internal rechargeable battery. Similar to previous research on IMDs, our findings on the PDHM protocol opens the possibility of depleting its battery if a vast amount of messages are received and processed by the HMU. Since we know the necessary headers at the transport, encryption, and compression layer, we can craft a large amount of data transmissions which the HMU will need to decrypt and decompress. The data processing on a large number of incoming transmissions will theoretically replenish its battery rapidly, and effectively render the HMU device useless. This will effect the patient's ability to transmit health monitoring data, and possibly disable the HMU's ability to alert a physician about occurring health issues.

The second possible attack scenario is the ***remote DoS attack*** which is also enabled by our knowledge of the PDHM protocol. In our analysis we also discovered that the implemented compression library contains two known CVEs that has the potential to cause Denial-of-Service - i.e a device crash. The zlib compression library version 1.2.1 is vulnerable to the known CVEs CVE-2005-2096 and CVE-2004-0797 [50]. The former is caused by a crafted stream that has an incomplete code description and length greater than 1. This causes a buffer overflow and has been demonstrated by a crafted PNG file [1]. It is therefore plausible that an attacker might be able to cause a DoS attack on the HMU by sending a crafted PNG file in the compressed payload in the PDHM transmission. This still needs to be tested because the internal data processing is not completely understood, and there might be restrictions on the compression payload - e.g payload size limit.

6.1.3 Potential Attack Scenarios

The following subsection defines the potential and unconfirmed attack scenarios that are plausible on the CardioMessenger Smart 3G version 1.20, based of the findings in my analysis. These attacks are active and theoretically require extensive efforts to confirm.

The first potential attack scenario is an ***install modified file attack***. In our analysis we found that the PDHM protocol is able to download a file and install it through its Message_InstallFile function. We found that the incoming file needs to have a magic number in the header, and that the CheckSignature functions seems to perform a hashing calculation. If an attacker were able to craft a custom file with the magic number in the header and the correct hash signature, then the HMU could potentially be weaponized and turned into an attack vector, used against the data servers or the patient's IMD. This potential attack requires extensive effort and might be plausible based on our protocol knowledge and the hashing-based CheckSignature function. This attack is also enabled by the lack of a digital signature, and the fact that SHA-1 is not collision resistant [62]. The lack of collision resistance is also a reason for our suggested migration to SHA-2 or SHA-3 in section 5.1.2.

The second potential attack scenario is a ***MitM illegitimate BTS attack*** between the HMU and the backend data server. This can be achieved by an extension of the previously mentioned modified firmware attack, or by setting up a software-defined illegitimate base station. The latter has been proven by others as well [22, 48], however with our added knowledge of the CardioMessenger Smart 3G it is possible to extend the functionality of the BTS. The modified HMU or the illegitimate BTS can act as an intermediary that forwards forged data, or it might eavesdrop and run other malicious actions in the background.

6.1.4 Implications for Patients & the Manufacturer

From the perspective of the patient, the medical devices are a life-saving solution to their medical condition. They are only given to patients in a particularly vulnerable situation, and the patient trusts that these devices can reliably protect them if a medical emergency occurs. Our findings show that some of the implementations in

[1] Available at https://www.cvedetails.com/cve/CVE-2005-2096/

the medical devices can potentially pose a threat to the patient's security and safety, however this still needs to be conclusively proved with a proof of concept.

The manufacturers of medical devices are responsible for patients in an extremely vulnerable situation. They are not just responsible for a fully functioning IMD that performs its heart-sustaining and monitoring tasks, they are also responsible for the device doing its tasks in a secure manner. The patients gets an IMD and monitoring devices because they have no other option. If the medical devices fail or are non-functioning, the patient's life is threatened. The manufacturer has an additional layer of added responsibility. The failure or unavailability of medical devices are more critical than any other system, and they should be treated accordingly. The manufacturers of medical devices therefore needs to proactively assess their software, and continuously identify the current threats and vulnerabilities that can affect every part the pacemaker ecosystem.

6.2 Ethical Considerations

The purpose of our research is to assure and confirm that proper security mechanism are implemented in the medical devices. Due to the sensitive nature of the medical devices, I have entered a confidentiality agreement with SINTEF to not disclose any information related to our findings in the devices. The devices have also been handled cautiously, and only been kept and used inside the SINTEF lab.

Our findings are related to both hardware and software. While software is easier to update on distributed devices, there are solutions which can make up for the lack of certain hardware components. We also intent to disclose our findings with the manufacturer. We also hope the manufacturer is able to appreciate the feedback and our genuine and ethical attempt to strengthen the security of medical devices in the future, which will benefit both the company and their patients.

6.3 Future Work

Our work in this thesis is the latest of multiple projects on the HMU. The other projects have performed an outside analysis of the hardware and of its communications, while in this project we tried a new approach from the inside of the HMU and reversing certain code sections. The PDHM communication protocol has now been analyzed both from the outside with a software-defined base station, and on the inside with my reverse engineering of the decompiled code in the HMU's memory. Still, the entire functionality of the communication protocol and its different modes are not yet fully understood. To document the protocol in its entirety, further reverse engineering and developing a practical proof-of-concept is needed.

The USB interface of the CardioMessenger Smart 3G has also not been analyzed.

In my project I only found functionality related to charging and charging safety measures, but it might be a communication interface which has not been identified yet.

Since the HMU has been the primary research objective of multiple of the latest projects, there is also the option of focusing on another device in the ecosystem, either the IMD or the programmer. For the IMD there is the ULPAMI communication interface which has not been explored by any of us at the SINTEF project. Other researchers have analyzed the ULPAMI communication interface on devices from other manufacturers, however not on Biotronik's devices to our knowledge. Understanding the ULPAMI interface might also be useful to understand its relation to the PDHM protocol and the interaction between them in the HMU. However, the IMD is hard to come by and it needs to have sufficient battery power remaining to perform the analysis. The IMDs available in SINTEF's lab do not have have enough battery power remaining. Thus, a new IMD is needed to perform the ULPAMI analysis with a software-defined radio. Alternatively, the ULPAMI interface can be analyzed from the inside by continuing the reverse engineering efforts from my project. I only did minimal function identification of the ULPAMI functions while exploring the function hierarchy, but since ULPAMI was not my main objective it was not heavily analyzed.

All the projects so far have all focused on the home monitoring devices of Biotronik. The lab has a couple of older devices from other manufacturers, but it is possible to obtain newer models from other vendors and perform a security analysis comparing the hardware or software between the vendors. Some devices from other vendors also have a Bluetooth communication interface which can connect to the patients smart phone. Bluetooth was not available in any of Biotroniks devices, and could be an interesting interface to analyze.

Chapter 7 Conclusion

In this thesis we have followed a black box method to reverse engineer the communication protocol of the Biotronik CardioMessenger Smart 3G Home Monitoring Unit. We started our analysis by formulating research questions relating to what extent the HMU is preserving and protecting the patients safety, privacy, and personal data. In the process of identifying the communication protocol, we found that the HMU lacks the necessary security mechanism to securely protect itself and its data transmissions. The scenarios in section 6.1.2 describe multiple severe vulnerabilities which needs to be mitigated to maintain security in the pacemaker ecosystem.

«We never compromise on safety—it is and will remain our number one priority. We invest the time and resources necessary to ensure that every product we bring to market is one we can stand by confidently—and cybersecurity is no exception.» - Biotronik [1]

It is clear from Biotronik's statement that patient safety and cybersecurity are their main priorities. These priorities are essential to ensure the security of their medical devices, which enable patients all around the globe to live a normal life. However, to ensure security of the medical devices it is important to keep up with a dynamic threat landscape, implement security best-practices throughout their devices, and continuously seek to improve the security of their product in every aspect. Feedback from the research community should also be regarded as a well-intended input to confirm and improve the security and reliability of their medical devices.

[1] Available at https://www.biotronik.com/en-de/patients/cybersecurity

References

[1] The Open Source Security Testing Methodology Manual - OSSTMM3. https://www.isecom.org/OSSTMM.3.pdf, 2010.

[2] GZIP file format specification version 4.3. https://datatracker.ietf.org/doc/html/rfc1952.

[3] Philip Howard. The Raspberry Pi GPIO pinout guide. https://pinout.xyz/#.

[4] STMicroelectronics. Datasheet STM32F405xx STM32F407xx. https://www.st.com/resource/en/datasheet/dm00037051.pdf.

[5] Microsoft Threat Modeling Tool threats - STRIDE model. https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats#stride-model, 2022.

[6] The DREAD approach to threat assessment. https://docs.microsoft.com/en-us/windows-hardware/drivers/driversecurity/threat-modeling-for-drivers, 2021.

[7] NSA'S Top Ten Cybersecurity Mitigation Strategies. https://www.nsa.gov/portals/75/documents/what-we-do/cybersecurity/professional-resources/csi-nsas-top10-cybersecurity-mitigation-strategies.pdf, 2018.

[8] Medical Device Radiocommunications Service (MedRadio). https://www.fcc.gov/medical-device-radiocommunications-service-medradio, 2017.

[9] Biotronik. Cardiomessenger smart technical manual. https://manuals.biotronik.com/emanuals-professionals-rest/manual/HomeMonitoring/CardioMessenger/CardioMessengerSmart/GB/en/L?type=manual, Mar 2021.

[10] MDCG 2019-16 - Guidance on Cybersecurity for medical devices. https://ec.europa.eu/docsroom/documents/41863, 2019.

[11] Good practices for the security of healthcare services. https://www.enisa.europa.eu/topics/critical-information-infrastructures-and-services/health/good-practices-for-the-security-of-healthcare-services#/filters?proc-types=Medical%20devices.

[12] NIST Glossery. https://csrc.nist.gov/glossary/.

[13] Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. https://ieeexplore.ieee.org/document/4531149, 2008.

[14] Svein Færestrand. Telekardiologi for fjernmonitorering av pacemaker og icd. https://www.legeforeningen.no/contentassets/4896657d08894a6886de725113d89de4/hjerteforum3-2010web08telemedisin.pdf, 2010.

[15] Kevin Fu. On the Technical Debt of Medical Device Security. https://www.naefrontiers.org/File.aspx?id=50750, 2015.

[16] Content of Premarket Submissions for Management of Cybersecurity in Medical Devices. https://www.fda.gov/regulatory-information/search-fda-guidance-documents/content-premarket-submissions-management-cybersecurity-medical-devices, 2014.

[17] MedSec. STJ's Compromised Cardiac Device Ecosystem. https://www.muddywatersresearch.com/research/stj/mw-is-short-stj/, 2016.

[18] Marin, Eduard and Singelée, Dave and Garcia, Flavio D. and Chothia, Tom and Willems, Rik and Preneel, Bart. On the (in)security of the latest generation implantable cardiac defibrillators and how to secure them. https://doi.org/10.1145/2991079.2991094, 2016.

[19] Billy Rios & Jonathan Butts. WhiteScope: Security Evaluation of the Implantable Cardiac Device Ecosystem Architecture and Implementation Interdependencies. https://www.ledecodeur.ch/wp-content/uploads/2017/05/Pacemaker-Ecosystem-Evaluation.pdf, 2017.

[20] R, Manikandan and Sathyadevan, Shiju. Medical Implant Communication Systems (MICS) Threat Modelling. https://ieeexplore.ieee.org/document/9478155, 2021.

[21] Eivind Skjelmo Kristiansen & Anders Been Wilhelmsen. Security Testing of the Pacemaker Ecosystem. https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2609516, 2018. Norwegian University of Science and Technology.

[22] Anniken Wium Lie. Security Analysis of Wireless Home Monitoring Units in the Pacemaker Ecosystem. https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2623147, 2019. Norwegian University of Science and Technology.

[23] Jakob Stenersen Kok & Bendik Aalmen Markussen. Fuzzing the Pacemaker Home Monitoring Unit. https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2781132, 2020. Norwegian University of Science and Technology.

[24] Guillaume Bour, Marie Elisabeth Gaup Moe, Ravishankar Borgaonkar. Experimental Security Analysis of Connected Pacemakers. https://guillaumebour.fr/pdf/Experimental%20Security%20Analysis%20of%20Connected%20Pacemakers.pdf, 2022. BIODEVICES 2022 – 15th International Conference on Biomedical Electronics and Devices.

[25] ICS Medical Advisory (ICSMA-20-170-05) BIOTRONIK CardioMessenger II. https://www.cisa.gov/uscert/ics/advisories/icsma-20-170-05, 2020.

[26] REGULATION (EU) 2017/745 The Medical Devices Regulation (MDR). https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32017R0746, 2017.

[27] REGULATION (EU) 2017/746 In vitro Diagnostic Medical Devices Regulation (IVDR). https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32017R0745, 2017.

[28] Act relating to national security (Security Act). https://lovdata.no/dokument/NLE/lov/2018-06-01-24, 2019.

[29] STMicroelectronics. STM32 32-bit Arm Cortex MCUs. https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html.

[30] STMicroelectronics. Reference manual STM32F4xx. https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf.

[31] STM8 & STM32 Functional Safety. https://www.st.com/content/st_com/en/ecosystems/functionalsafety.html.

[32] ARM Debug Interface v5 Architecture Specification. https://developer.arm.com/documentation/ihi0031/a/The-Serial-Wire-Debug-Port--SW-DP-/Introduction-to-the-ARM-Serial-Wire-Debug--SWD--protocol, 2006.

[33] Packages and Binaries: Binwalk. https://www.kali.org/tools/binwalk/.

[34] Binwalk Quick Start Guide. https://github.com/ReFirmLabs/binwalk.

[35] National Security Agency. Ghidra Software Reverse Engineering Framework. https://github.com/NationalSecurityAgency/ghidra.

[36] National Security Agency. Ghidra API. https://ghidra.re/ghidra_docs/api/.

[37] National Security Agency. Ghidra Software Reverse Engineering Framework - Features. https://github.com/NationalSecurityAgency/ghidra/tree/master/Ghidra/Features.

[38] Gavin Wright. Scientific Method. https://www.techtarget.com/whatis/definition/scientific-method, 2022.

[39] Fiona Fidler and John Wilcox. Reproducibility of Scientific Results. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2021 edition, 2021. https://plato.stanford.edu/archives/sum2021/entries/scientific-reproducibility/.

[40] Monya Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533:452–454, 2016. https://doi.org/10.1038/533452a.

[41] Asher Mullard. Half of top cancer studies fail high-profile reproducibility effort. *Nature*, 2021. https://doi.org/10.1038/d41586-021-03691-0.

[42] Victoria Drake. Threat Modeling. https://owasp.org/www-community/Threat_Modeling.

[43] Pacemaker Club Famous Recipients. https://www.pacemakerclub.com/famous-recipients.

[44] National Initiative for Cybersecurity Careers and Studies. Explore Terms: A Glossary of Common Cybersecurity Words and Phrases. https://niccs.cisa.gov/about-niccs/cybersecurity-glossary#body.

[45] Venkatesh Jagannathan. Threat Modeling Architecting Designing with Security in Mind. https://owasp.org/www-pdf-archive/AdvancedThreatModeling.pdf.

[46] Microsoft Threat Modeling Tool mitigations. https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-mitigations, 2022.

[47] Thomas Hamilton. What is BLACK Box Testing? Techniques, Example Types. https://www.guru99.com/black-box-testing.html, 2022.

[48] Guillaume Bour. Security Analysis of the Pacemaker Home Monitoring Unit: A BlackBox Approach. https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2623154, 2019. Norwegian University of Science and Technology.

[49] Jean-loup Gailly & Mark Adler. zlib - A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://zlib.net/.

[50] CVE Details Zlib 1.2.1 Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-13265/product_id-111843/version_id-693868/Zlib-Zlib-1.2.1.html, 2005.

[51] leveldown security . SVD-Loader for Ghidra. https://github.com/leveldown-security/SVD-Loader-Ghidra.

[52] Guillaume Bour. stm32f4_loader. https://github.com/SINTEF-Infosec/pacemaker-hacking/tree/master/reverse-engineering/ghidra_plugins/stm32f4_loader.

[53] ARM Keil. MDK5 Software Packs - STMicroelectronics STM32F4 Series Device Support, Drivers and Examples - Keil.STM32F4xx_DFP.2.15.0.pack. https://www.keil.com/dd2/pack/#!#eula-container, 2020.

[54] Guillaume Bour. Security testing of the pacemaker ecosystem - Part 4. https://guillaumebour.fr/articles/security_testing_pacemaker_ecosystem/part_4_re_proprietary_protocl_hmu/, 2020.

[55] STMicroelectronics. STM32F4 DSP and standard peripherals library. https://www.st.com/content/my_st_com/en/products/embedded-software/mcu-mpu-embedded-software/stm32-embedded-software/stm32-standard-peripheral-libraries/stsw-stm32065.license=1655065797510.product=STSW-STM32065.version=1.9.0.html#get-software.

[56] Joseph Yiu. The Definitive Guide to ARM® CORTEX®-M3 and CORTEX®-M4 Processors (Third Edition). https://www.sciencedirect.com/science/article/pii/B9780124080829000105, 2014.

[57] Real-Time Operating System: API and RTX Reference Implementation. https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group___CMSIS___RTOS___ThreadMgmt.html.

[58] TPM fundamentals. https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/tpm-fundamentals, 2022.

[59] System Memory Protections. https://www.st.com/content/ccc/resource/training/technical/product_training/group0/f5/5e/87/93/f5/d7/45/85/STM32F7_Security_Memories_Protections/files/STM32F7_Security_Memories_Protections.pdf/jcr:content/translations/en.STM32F7_Security_Memories_Protections.pdf.

[60] Block Cipher Techniques. https://csrc.nist.gov/projects/block-cipher-techniques, 2020.

[61] Barker, Elaine & Mouha, Nicky. Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-67r2.pdf, 2017.

[62] Research Results on SHA-1 Collisions. https://csrc.nist.gov/News/2017/Research-Results-on-SHA-1-Collisions, 2017.

[63] OpenJDK. AdoptOpenJDK Latest Release. https://adoptopenjdk.net/releases.html?variant=openjdk11&jvmVariant=hotspot.

Appendix A
Binary Extraction

Extracting the device memory, i.e binary files, from the embedded HMU device is a prerequisite to reverse engineer the code. The extraction/dumping of the binaries was already performed by Guillaume Bour previously in the Pacemaker project at SINTEF. This chapter thoroughly describes the process of reproducing his findings with the same tools, in the same environment. As described in the methodology, reproducing findings from previous work is fundamental, both to validate his work, my own and future work done by others.

The following subsections in this appendix will describe the connection setup to communicate with the HMUs microcontroller through a raspberryPi running OpenOCD. The OpenOCD script for establishing connection and dumping to the microcontroller through JTAG is created by Guillaume Bour. I replicated the memory dumping over JTAG, and I did modifications to the configuration scripts to show that also the SWD interface is accessible. I was also able to establish a connection and dump the memory through the SWD interface.

The connection to the embedded HMU devices is done by connecting my laptop wirelessly to the wireless access point generated on the rasperryPi Zero. An SSH connection is then established on top from the laptop over WiFi. Enabling the possibility to remotely execute code from the raspberryPi device on the HMU.

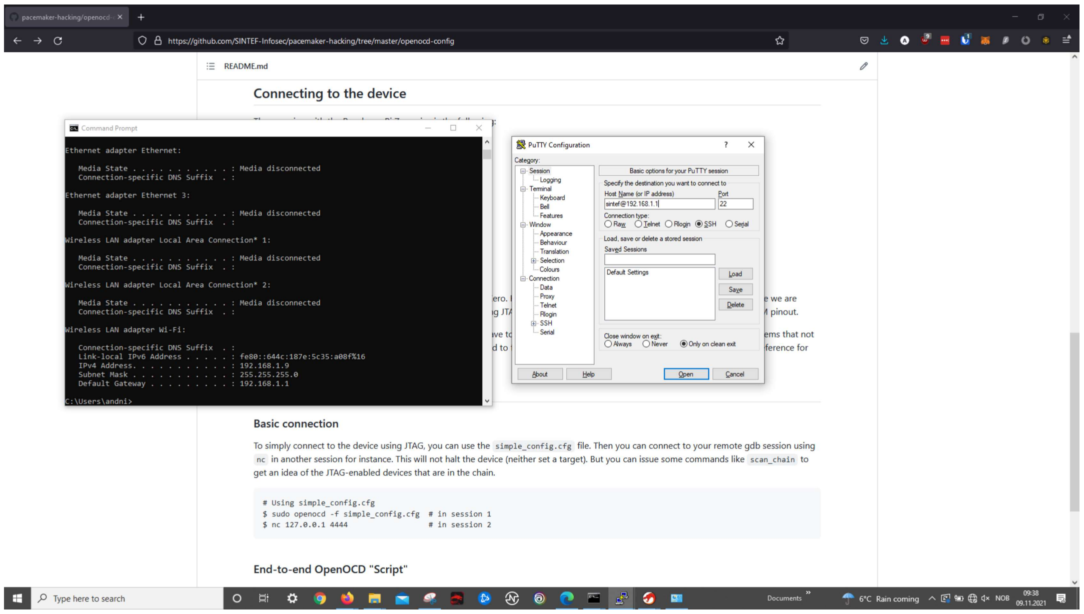
The raspberryPi uses OpenOCD to provide debugging and programming functionality on the embedded HMU device. OpenOCD configuration files are customized on the pi to specify the used transport protocols and commands to be executed on the HMU device's microprocessor. The physical connection between the raspberryPi and the HMU is connected through the PCBite wire kit - a solder-free, and flexible system with wire arms to connect onto circuit boards.

A.1 OpenOCD Installation

To install OpenOCD on the Raspberry Pi Zero running Linux, only a few lines of commands are necessary. As described in Bour's project, we run the following commands:

```
$ sudo apt-get update
$ sudo apt-get install git autoconf libtool make pkg-config libusb-1.0-0
                           libusb-1.0-0-dev
$ git clone http://openocd.zylin.com/openocd
$ cd openocd
$ ./bootstrap
$ ./configure --enable-sysfsgpio --enable-bcm2835gpio
$ make
$ sudo make install
```

A.2 Experiment Setup


https://github.com/SINTEF-Infosec/pacemaker-hacking/tree/master/openocd-config

README.md

Connecting to the device

```
Ethernet adapter Ethernet:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Ethernet adapter Ethernet 3:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Local Area Connection* 1:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Local Area Connection* 2:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::644c:187e:5c35:a08f%16
   IPv4 Address. . . . . . . . . . . : 192.168.1.9
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1

C:\Users\andni>
```

PuTTY Configuration

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): sintef@192.168.1.1

Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

Saved Sessions: Default Settings

Close window on exit: Always Never [✓ Only on clean exit]

Basic connection

To simply connect to the device using JTAG, you can use the `simple_config.cfg` file. Then you can connect to your remote gdb session using `nc` in another session for instance. This will not halt the device (neither set a target). But you can issue some commands like `scan_chain` to get an idea of the JTAG-enabled devices that are in the chain.

```
# Using simple_config.cfg
$ sudo openocd -f simple_config.cfg  # in session 1
$ nc 127.0.0.1 4444                  # in session 2
```

End-to-end OpenOCD "Script"

Figure A.1: SSH connection on laptop to raspberryPi over WiFi

In figure A.1 the setup of the SSH protocol in PuTTy is setup. Firstly, we connect to the wireless AP on the Pi Zero. We then check our IP address and connect to the

Pi using *sintef@192.168.1.1; port 22*. This gives us access to the Raspberry Pi and to navigate its directories and run code remotely.

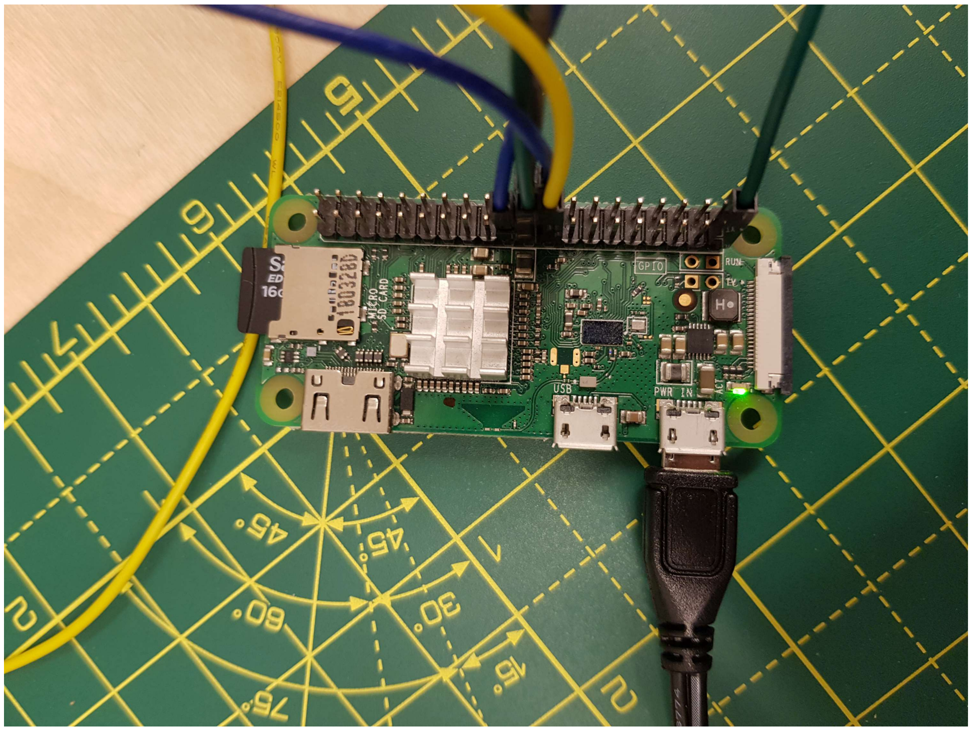
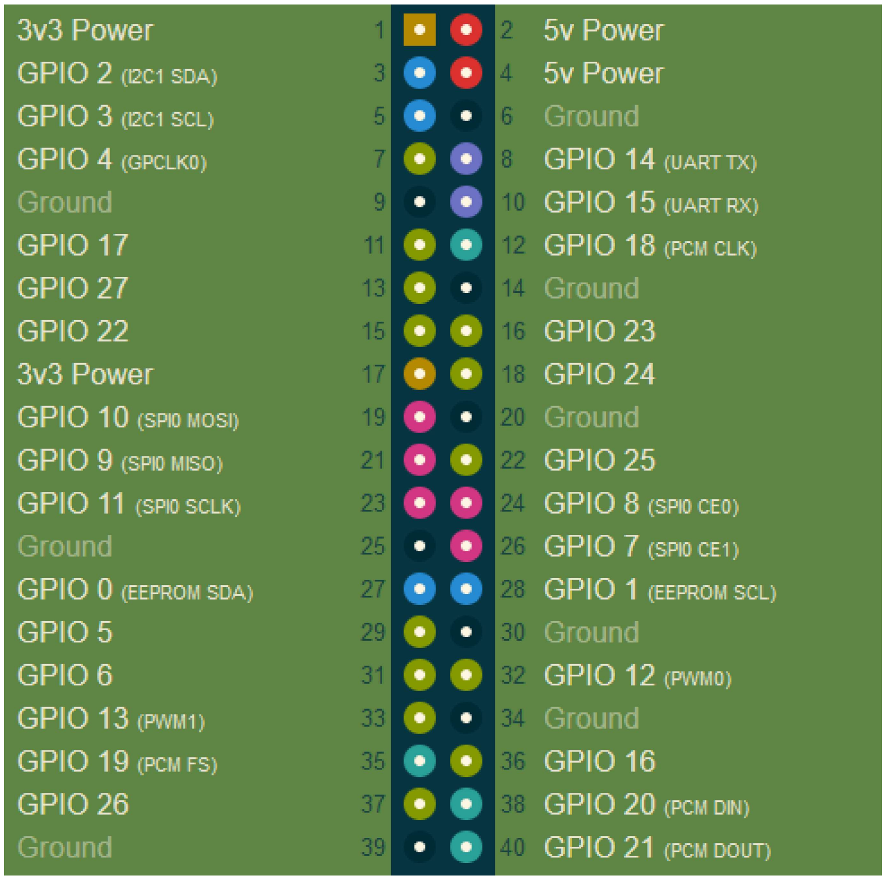


Figure A.2: RaspberryPi connecting wires on GPIO pins

The RaspberryPi Zero is also connected to the HMU circuit board with electrical wires from the PCBite kit. The wires are connected to specific GPIO pins which is defined in the OpenOCD configuration files, see *bcm2835gpio_jtag_nums* and *bcm2835gpio_swd_nums* in algorithms in Protocols & Scripts on raspberryPi. The pin numbers and GPIO pins used to connect the jtag and swd protocol correctly for the OpenOCD configuration can be found through the github project *Pinout.xyz*[3]. For example, in figure A.2 we can see the jtag connection on the Raspberry Pi Zero. The jtag numbers used in the configuration file is 11, 25, 10, and 9. These numbers corresponds to GPIO pins 23, 22, 19, and 21, as seen in figure A.3. For the swd protocol the pin numbers in configuration is 11 and 10 for SWDIO and SWCLK, which corresponds to GPIO pins 23 and 19.



3v3 Power	1	2	5v Power
GPIO 2 (I2C1 SDA)	3	4	5v Power
GPIO 3 (I2C1 SCL)	5	6	Ground
GPIO 4 (GPCLK0)	7	8	GPIO 14 (UART TX)
Ground	9	10	GPIO 15 (UART RX)
GPIO 17	11	12	GPIO 18 (PCM CLK)
GPIO 27	13	14	Ground
GPIO 22	15	16	GPIO 23
3v3 Power	17	18	GPIO 24
GPIO 10 (SPI0 MOSI)	19	20	Ground
GPIO 9 (SPI0 MISO)	21	22	GPIO 25
GPIO 11 (SPI0 SCLK)	23	24	GPIO 8 (SPI0 CE0)
Ground	25	26	GPIO 7 (SPI0 CE1)
GPIO 0 (EEPROM SDA)	27	28	GPIO 1 (EEPROM SCL)
GPIO 5	29	30	Ground
GPIO 6	31	32	GPIO 12 (PWM0)
GPIO 13 (PWM1)	33	34	Ground
GPIO 19 (PCM FS)	35	36	GPIO 16
GPIO 26	37	38	GPIO 20 (PCM DIN)
Ground	39	40	GPIO 21 (PCM DOUT)

Figure A.3: Raspberry Pi pinout [3]

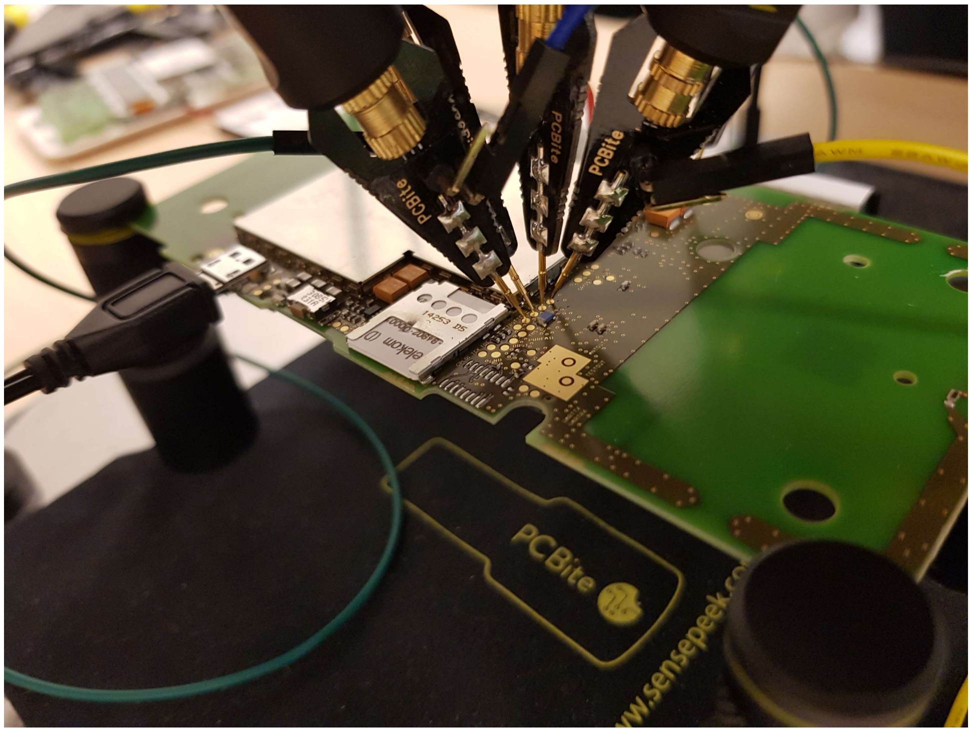


Figure A.4: PCBite JTAG connection on the HMU circuit board

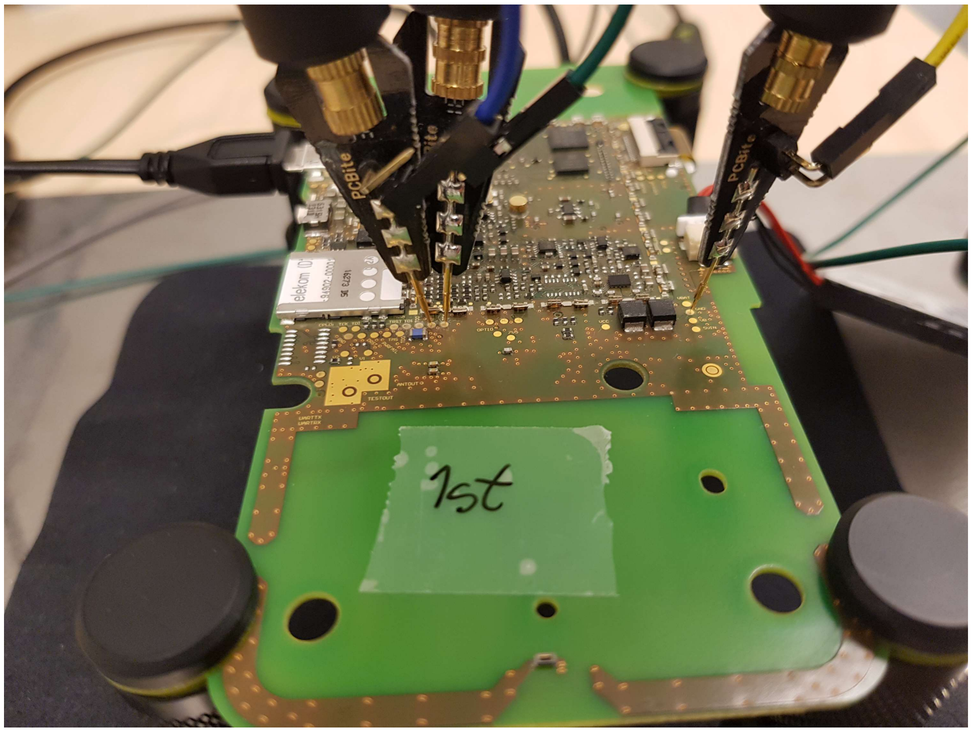


Figure A.5: PCBite SWD connection on the HMU circuit board

A.3 Microprocessor Capabillities

Running the following commands while connected to the corresponding interface establishes a connection to the microprocessor:

```
# for a JTAG connection:
sudo openocd -f simple_config_3g.cfg

# for a SWD connection:
sudo openocd -f simple_config_3g_swd.cfg
```

After setting up the simple connection on either jtag or swd, we launch another terminal session with NetCat:

```
$ nc 127.0.0.1 4444
```

Running the following commands while connected to the corresponding interface establishes dumps the memory from the microprocessor:

```
# for a JTAG connection:
sudo openocd -f dump_memory.cfg

# for a SWD connection:
sudo openocd -f dump_memory_swd.cfg
```

The memory addresses can be found in the STM32 microprocessor datasheet [4]. Its memory map section defines the name, base memory address, and size for each memory block. The memory map is added below in figure A.6 for reference.

```
flash.img 0x08000000 1048575
ccm_ram.img 0x10000000 65535
system_memory_OTP.img 0x1FFF0000 31247
sram.img 0x20000000 131071
ram.img 0x60000000 2097152
```

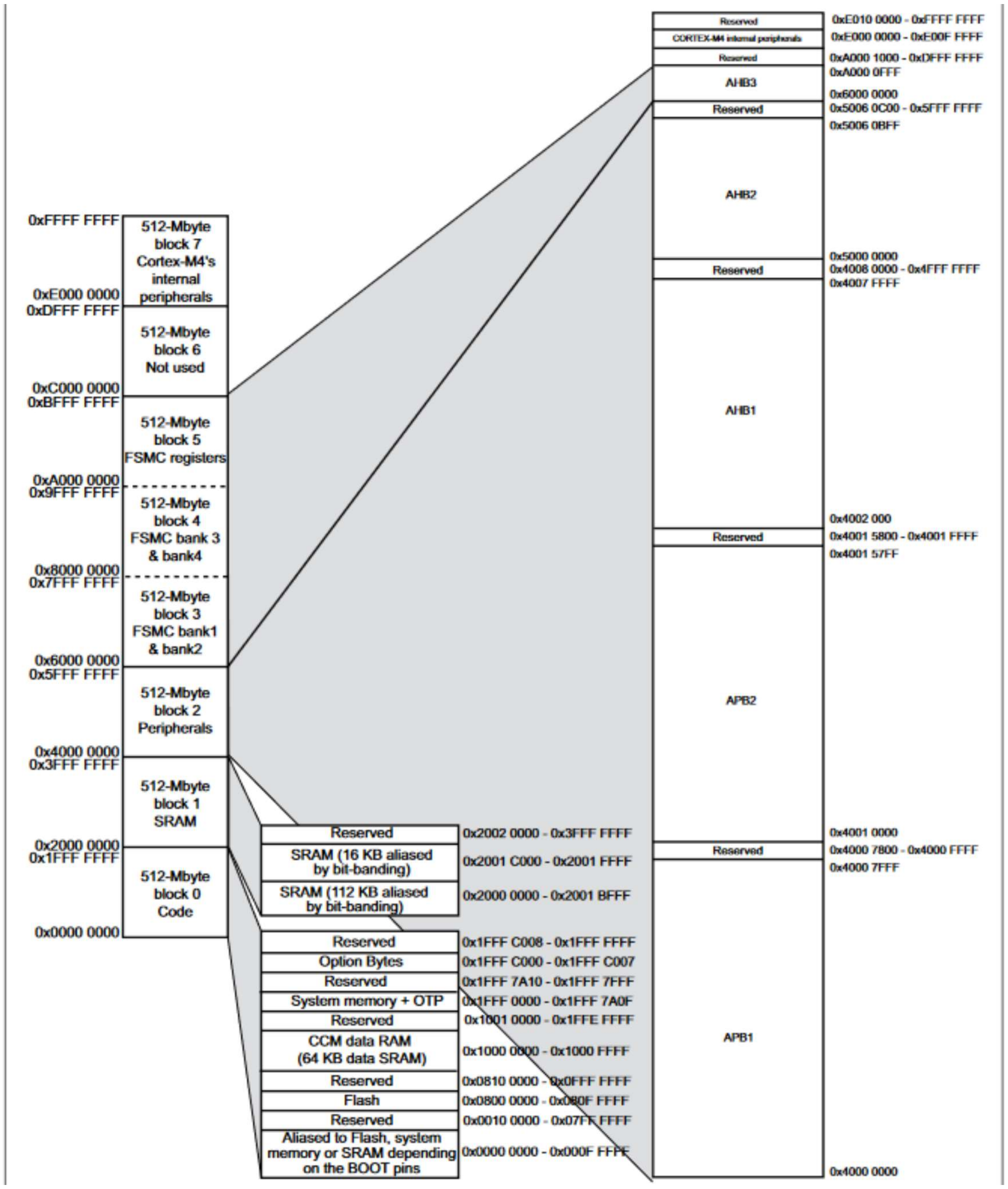

Reserved
0xE010 0000 - 0xFFFF FFFF
CORTEX-M4 internal peripherals
0xE000 0000 - 0xE00F FFFF
Reserved
0xA000 1000 - 0xDFFF FFFF
AHB3
0xA000 0FFF
0x6000 0000
Reserved
0x5006 0C00 - 0x5FFF FFFF
0x5006 0BFF
AHB2
0x5000 0000
Reserved
0x4008 0000 - 0x4FFF FFFF
0x4007 FFFF
AHB1
0x4002 000
Reserved
0x4001 5800 - 0x4001 FFFF
0x4001 57FF
APB2
0x4001 0000
Reserved
0x4000 7800 - 0x4000 FFFF
0x4000 7FFF
APB1
0x4000 0000
0xFFFF FFFF
512-Mbyte block 7 Cortex-M4's internal peripherals
0xE000 0000
0xDFFF FFFF
512-Mbyte block 6 Not used
0xC000 0000
0xBFFF FFFF
512-Mbyte block 5 FSMC registers
0xA000 0000
0x9FFF FFFF
512-Mbyte block 4 FSMC bank 3 & bank4
0x8000 0000
0x7FFF FFFF
512-Mbyte block 3 FSMC bank1 & bank2
0x6000 0000
0x5FFF FFFF
512-Mbyte block 2 Peripherals
0x4000 0000
0x3FFF FFFF
512-Mbyte block 1 SRAM
0x2000 0000
0x1FFF FFFF
512-Mbyte block 0 Code
0x0000 0000
Reserved
0x2002 0000 - 0x3FFF FFFF
SRAM (16 KB aliased by bit-banding)
0x2001 C000 - 0x2001 FFFF
SRAM (112 KB aliased by bit-banding)
0x2000 0000 - 0x2001 BFFF
Reserved
0x1FFF C008 - 0x1FFF FFFF
Option Bytes
0x1FFF C000 - 0x1FFF C007
Reserved
0x1FFF 7A10 - 0x1FFF 7FFF
System memory + OTP
0x1FFF 0000 - 0x1FFF 7A0F
Reserved
0x1001 0000 - 0x1FFE FFFF
CCM data RAM (64 KB data SRAM)
0x1000 0000 - 0x1000 FFFF
Reserved
0x0810 0000 - 0x0FFF FFFF
Flash
0x0800 0000 - 0x080F FFFF
Reserved
0x0010 0000 - 0x07FF FFFF
Aliased to Flash, system memory or SRAM depending on the BOOT pins
0x0000 0000 - 0x000F FFFF

Figure A.6: The memory map of the STM32F4xxx microprocessor [4]

A.4 Protocols & Scripts on raspberryPi

These are the scripts used in OpenOCD on the RaspberryPi communicating over JTAG and SWD, described in Background. The JTAG scripts in A.4 are developed by Guillaume Bour, and the SWD scripts in A.4 are my modifications of the JTAG scripts to support the SWD interface [48].

Algorithm A.1 simple_config_3g.cfg: Establishing JTAG connection to microcontroller

```
# Script used to establish a simple connection to a CardioMessenger 3g Smart using a JTAG acces.

# INTERFACE
interface bcm2835gpio
bcm2835gpio_peripheral_base 0x20000000
bcm2835gpio_speed_coeffs 113714 28
bcm2835gpio_jtag_nums 11 25 10 9
bcm2835gpio_srst_num 24
reset_config srst_only srst_push_pull
adapter_khz 500

# TRANSPORT
transport select jtag

# TARGET
set WORKAREASIZE 0
set CHIPNAME stm32f4x
source [find target/stm32f4x.cfg]
reset_config srst_only srst_nogate
adapter_nsrst_delay 100
adapter_nsrst_assert_width 100

# EXEC
init
targets
halt
```

Algorithm A.2 dump_memory.cfg: Establishing JTAG connection to microcontroller and dumping memory

```
# Script used to establish a simple connection to a CardioMessenger 3g Smart using a JTAG acces.

# INTERFACE
interface bcm2835gpio
bcm2835gpio_peripheral_base 0x20000000
bcm2835gpio_speed_coeffs 113714 28
bcm2835gpio_jtag_nums 11 25 10 9
bcm2835gpio_srst_num 24
reset_config srst_only srst_push_pull
adapter_khz 500

# TRANSPORT
transport select jtag

# TARGET
set WORKAREASIZE 0
set CHIPNAME stm32f4x
source [find target/stm32f4x.cfg]
reset_config srst_only srst_nogate
adapter_nsrst_delay 100
adapter_nsrst_assert_width 100

# EXEC
init
targets
halt


echo "Dumping flash..."
dump_image flash.img 0x08000000 1048575
echo "Done!"

echo "Dumping CCM RAM..."
dump_image ccm_ram.img 0x10000000 65535
echo "Done!"

echo "Dumping system memory OTP..."
dump_image system_memory_OTP.img 0x1FFF0000 31247
echo "Done!"

echo "Dumping SRAM..."
dump_image sram.img 0x20000000 131071
echo "Done!"

echo "Dumping RAM..."
dump_image ram.img 0x60000000 2097152
echo "Done!"
```

Algorithm A.3 simple_config_3g_swd.cfg: Establishing SWD connection to microcontroller

```
#Script used to estabish a simple connection to a CM 3G Smart using SWD access

# interface
interface bcm2835gpio
bcm2835gpio_peripheral_base 0x20000000
bcm2835gpio_speed_coeffs 113714 28
bcm2835gpio_swd_nums 11 10
#Is this (below) needed?
bcm2835gpio_srst_num 24
reset_config srst_only srst_push_pull
adapter_khz 500

# transport
transport select swd

# target
set WORKAREASIZE 0
set CHIPNAME stm32f4x
source [find target/stm32f4x.cfg]
reset_config srst_only srst_nogate
```

Algorithm A.4 dump_memory_swd.cfg: Establishing SWD connection to microcontroller and dumping memory

```
#interface
interface bcm2835gpio
bcm2835gpio_peripheral_base 0x20000000
bcm2835gpio_speed_coeffs 113714 28
bcm2835gpio_swd_nums 11 10
#
#
adapter_khz 500

#transport
transport select swd

#target
set WORKAREASIZE 0
set CHIPNAME stm32f4x
source [find target/stm32f4x.cfg]
#reset_config srst_only srst_nogate
#adapter_nsrst_delay 100
#adapter_nsrst_assert_width 100

#exec
init
targets
halt

echo "Dumping flash..."
dump_image flash.img 0x08000000 1048575
echo "Done!"

echo "Dumping CCM RAM..."
dump_image ccm_ram.img 0x10000000 65535
echo "Done!"

echo "Dumping system memory OTP..."
dump_image system_memory_OTP.img 0x1FFF0000 31247
echo "Done!"

echo "Dumping SRAM..."
dump_image sram.img 0x20000000 131071
echo "Done!"

echo "Dumping RAM..."
dump_image ram.img 0x60000000 2097152
echo "Done!"
```

Appendix B Binwalk Installation

To install Binwalk we followed the installation procedure from the Binwalk github repository under the *INSTALL.md*[34]. We already had Python version 3.8.10 installed on a virtual image of Linux Ubuntu 20.04.3. These are the series of commands executed in the Linux terminal to install all the standard dependencies for Binwalk. We did not need the additional dependencies listed from their github repository.

```
$ sudo pip install nose coverage

$ sudo pip install pycryptodome

$ sudo apt-get install libqt4-opengl python3-opengl python3-pyqt4
    python3-pyqt4.qtopengl python3-numpy python3-scipy python3-pip

$ sudo pip3 install pyqtgraph

$ sudo pip install capstone

$ sudo apt-get install mtd-utils gzip bzip2 tar arj lhasa p7zip
    p7zip-full cabextract cramfsprogs cramfsswap squashfs-tools
    sleuthkit default-jdk lzop srecord
```

To use Binwalk, open a terminal window and type:

```
#  Commands in Binwalk are formulated like this:
binwalk [option(s)] [file(s)]
# To run an entropy scan on the flash file:
binwalk --entropy flash.img
```

Appendix C
Ghidra Installation & Setup

This chapter explains the steps to install and setup Ghidra. In our project we started out using Ghidra version 10.0.4 released in October 2021. However, because of the Log4j vulnerability the project was migrated to Ghidra version 10.1.4 released 20th May 2022.

```
# SHA-256 checksum:
91556c77c7b00f376ca101a6026c0d079efbf24a35b09daaf80bda897318c1f1
```

C.1 Installation & Dependencies

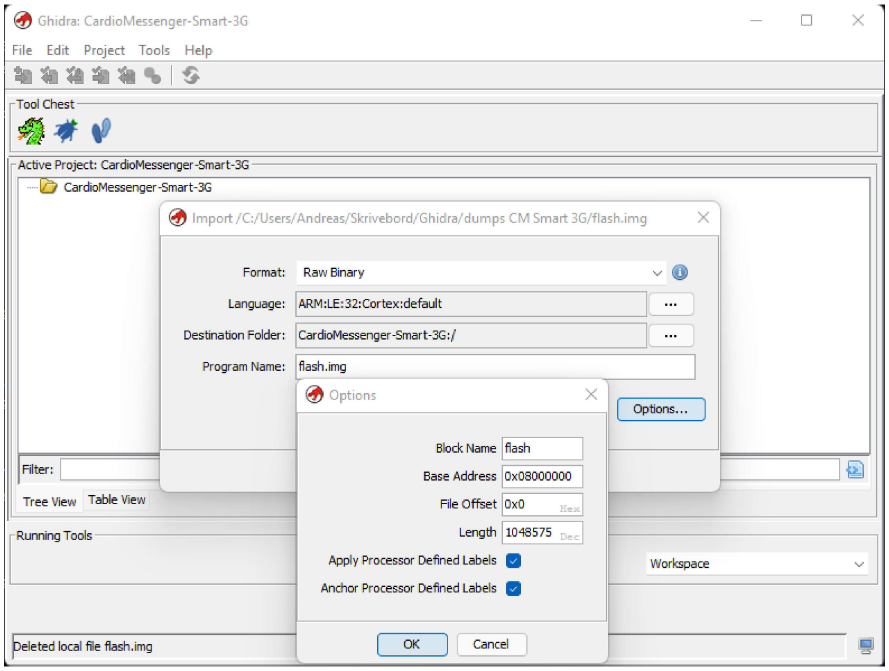
Ghidra supports Microsoft Windows 7 and 10 (64bit), Linux (64bit), and macOS 10.8.3 or newer. Ghidra only has one software dependency which is Java 11 64-bit Runtime and Development Kit (JDK): OpenJDK 11 (LTS)[63].
After clicking on Ghidra 10.1.4, a list of assets appear and we download the *ghidra_10.1.4_PUBLIC_20220519.zip*. This zip-file contains the entirety of Ghidra. After extracting the zip-file, double click the *ghidraRun.bat* file to launch Ghidra.

C.2 Setting up the Project & adding Binaries

After launching the Ghidra, a window will open where you can see your open projects, create and modify your projects, and select a few different tools.

C.2.1 Adding the Binary

When opening Ghidra the first time, we need to create a new project and import the file we are analyzing. In this case, the file is one of the memory dump binaries what we extracted from the CardioMessenger Smart 3G. The STM32F417 datasheet defines the base addresses for the microprocessor main memory is the same addresses we used during the memory dumping of the flash section [4]. The main memory starts at *0x0800 0000*-hex and ends at *0x080F FFFF*-hex, which translates to a length of *1048575*-decimal. It is important to define these addresses correctly because they will be used by Ghidra when executing its different analysis tools.



Import /C:/Users/Andreas/Skrivebord/Ghidra/dumps CM Smart 3G/flash.img

Format: Raw Binary
Language: ARM:LE:32:Cortex:default
Destination Folder: CardioMessenger-Smart-3G:/
Program Name: flash.img

Options

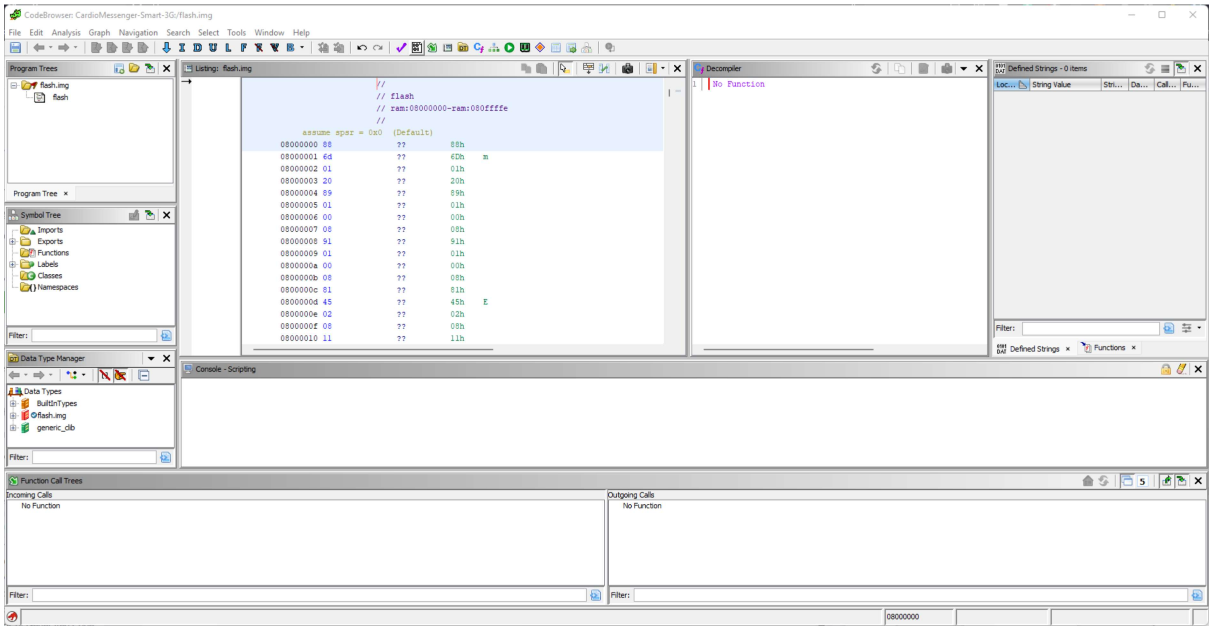
Block Name: flash
Base Address: 0x08000000
File Offset: 0x0 Hex
Length: 1048575 Dec
Apply Processor Defined Labels ✓
Anchor Processor Defined Labels ✓

Deleted local file flash.img

Figure C.1: Import a memory file to a Ghidra project

C.2.2 Ghidras Workspace

Double-clicking the imported file, or clicking on the tiny green dragon icon in the *Tool Chest*, starts up Ghidra and adds the file to the workspace. The image below shows how Ghidras workspace looks after opening the project file.



```
                              //
                              // flash
                              // ram:08000000-ram:080ffffe
                              //
           assume spsr = 0x0  (Default)
        08000000 88           ??        88h
        08000001 6d           ??        6Dh    m
        08000002 01           ??        01h
        08000003 20           ??        20h
        08000004 89           ??        89h
        08000005 01           ??        01h
        08000006 00           ??        00h
        08000007 08           ??        08h
        08000008 91           ??        91h
        08000009 01           ??        01h
        0800000a 00           ??        00h
        0800000b 08           ??        08h
        0800000c 81           ??        81h
        0800000d 45           ??        45h    E
        0800000e 02           ??        02h
        0800000f 08           ??        08h
        08000010 11           ??        11h
```

Figure C.2: Ghidra initially

C.3 Running Scripts in Ghidra/Ghidra Plugins

Image of Script Manager
Mention default folder path for scripts in Windows: "C:\Users\username\ghidra_scripts"

Appendix D Developed Tools & Scripts for Ghidra

D.1 Ghidra .svd loader

Algorithm D.1 SVD Loader for Ghidra

```
import xml.etree.ElementTree as ET
from ghidra.program.model.data import Structure, StructureDataType, UnsignedIntegerDataType,
↪ DataTypeConflictHandler
from ghidra.program.model.data import UnsignedShortDataType, ByteDataType,
↪ UnsignedLongLongDataType
from ghidra.program.model.mem import MemoryBlockType
from ghidra.program.model.address import AddressFactory, Address, AbstractAddressSpace,\
    AddressSpace
from ghidra.program.model.symbol import SourceType
from ghidra.program.model.mem import MemoryConflictException
from compiler.ast import TryExcept

class MemoryBlock:
    def __init__(self, name, start, end, peripheral):
        self.name = name
        self.start = start
        self.end = end
        self.peripheral = peripheral

    def length(self):
        return self.end - self.start

svdfile = askFile("Choose .svd", "Load .svd")
tree = ET.parse(str(svdfile))
root = tree.getroot()

#Get ghidra variables
listing = currentProgram.getListing()
symbolTable = currentProgram.getSymbolTable()
dataTypeManager = currentProgram.getDataTypeManager()
addressSpace = currentProgram.getAddressFactory().getDefaultAddressSpace()
namespace = symbolTable.getNamespace("Peripherals.Registers", None)
if not namespace:
    namespace = currentProgram.getSymbolTable().createNameSpace(None, "Peripherals.Registers",
    ↪ SourceType.ANALYSIS)

interruptnamespace = symbolTable.getNamespace("Peripherals.Interrupts", None)
```

```python
if not interruptnamespace:
    interruptnamespace = currentProgram.getSymbolTable().createNameSpace(None,
    ↪  "Peripherals.Interrupts", SourceType.ANALYSIS)

memoryblocks = []
for peripherals in root:
    for peripheral in peripherals:
        try:
            name = peripheral.find("name").text
            start = int(peripheral.find("baseAddress").text, 16)
            try:
                length = int(peripheral.find("addressBlock").find("size").text, 16)
                #Length is changed if memory overlap is detected
                end = start+length
            except:
                end = start+1024
            memoryblocks.append(MemoryBlock(name, start, end, peripheral))
        except:
            print(name+" failed")

for block1 in memoryblocks:
    for block2 in memoryblocks:
        if block1!=block2:
            if block1.start>block2.start and block1.start<block2.end:
                block2.end=block1.start-1

print("Generating memory blocks:")
for block in memoryblocks:
    print(block.name)
    name = block.name
    address = addressSpace.getAddress(block.start)
    temp = currentProgram.memory.createUninitializedBlock(block.name, address, block.length(),
    ↪  False)
    temp.setRead(True)
    temp.setWrite(True)
    temp.setExecute(False)
    temp.setVolatile(True)
    temp.setComment("Generated by andnilse script")

#Generate labels for registers
print("Generating peripherals.register labels..")
for peripheral in peripherals:
    print(peripheral.find("name").text)
    peripheralBaseAddress = peripheral.find("baseAddress").text
    for registers in peripheral:
        for register in registers.findall("register"):
            length = int(register.find("size").text, 16)
            register_start = int(peripheralBaseAddress,
            ↪  16)+int(register.find("addressOffset").text, 16)
            addr = addressSpace.getAddress(register_start)
            labelname = peripheral.find("name").text+"."+register.find("name").text
            print("\t"+register.find("name").text)
            symbolTable.createLabel(addr, labelname, namespace, SourceType.USER_DEFINED)

print("Generating peripherals.interrupts labels..")
for peripheral in peripherals:
    for interrupt in peripheral:
        try:
            peripheralname = peripheral.find("name").text
            interruptname = interrupt.find("name").text
            fullname = peripheralname+"."+interruptname
            value = "0x000000"+interrupt.find("value").text
        except:
            continue
        print(fullname+" "+value)
        try:
            address = addressSpace.getAddress(value)
            temp = currentProgram.memory.createUninitializedBlock(fullname, address, 1, False)
            # interrupt is of length 1 bit
            temp.setRead(True)
            temp.setWrite(True)
            temp.setExecute(False)
            temp.setVolatile(True)
            temp.setComment("Generated by andnilse script")
        except:
            print("Could not create uninitialized block for "+fullname+" "+str(address))
        symbolTable.createLabel(address, fullname, interruptnamespace, SourceType.USER_DEFINED)
```

D.2 Ghidra Function Identification Script

Algorithm D.2 GhidraFunctionFinder.py

```
from ghidra.program.flatapi import FlatProgramAPI
from ghidra.program.model.listing import FunctionManager, Function, Variable, VariableStorage,
↪ Listing, InstructionIterator
from ghidra.program.model.symbol import Namespace
from ghidra.app.plugin.core import instructionsearch
from ghidra.app.decompiler import DecompInterface
from ghidra.util.task import ConsoleTaskMonitor

currentprogram = currentProgram
currentprogram_name = currentprogram.getName()
fm = currentprogram.getFunctionManager()
functionlist = fm.getFunctions(True)
listing = currentprogram.getListing()

decompinterface = DecompInterface()
decompinterface.openProgram(currentprogram)
searchlist = ["CRYP"]#, "HASH"]
foundlist = []
file = open("GhidraFunctionFinder.txt", "a")

index=0
print "Number of functions:"+str(fm.getFunctionCount())
for function in functionlist:
    index+=1

    print str(round(100*float(index)/float(fm.getFunctionCount()), 2))+"%"
    results = decompinterface.decompileFunction(function, 0, ConsoleTaskMonitor())
    compiled = results.getDecompiledFunction().getC()
    found = True
    for item in searchlist:
        if(str(compiled).find(item)==-1):
            found=False

    if(found==True):#last element in searchlist( and searchlist.index(item)==len(searchlist)-1)
        print "Found function:"+str(function.getName())
        foundlist.append(str(function.getName()))
        file.write(str(compiled)+"\n")

```

```
print foundlist
print str(len(foundlist))
```

Appendix E
Entropy of Memory Files

E.1 ccm_ram

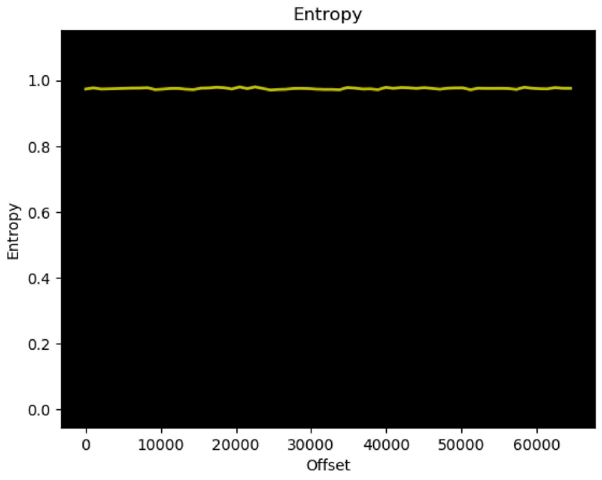


Figure E.1: Entropy of ccm_ram.img

E.2 flash

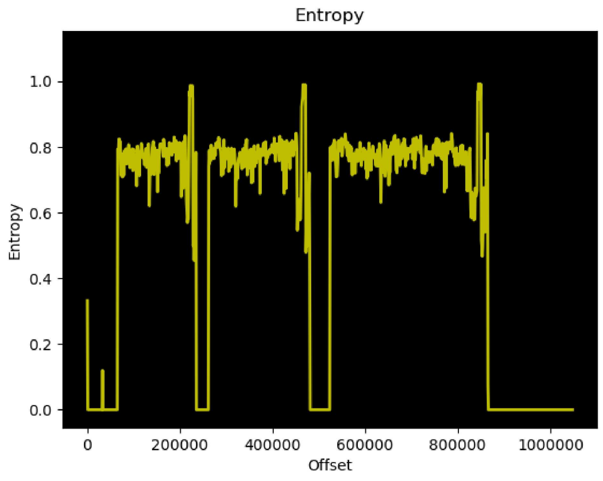


Figure E.2: Entropy of flash.img

E.3 ram

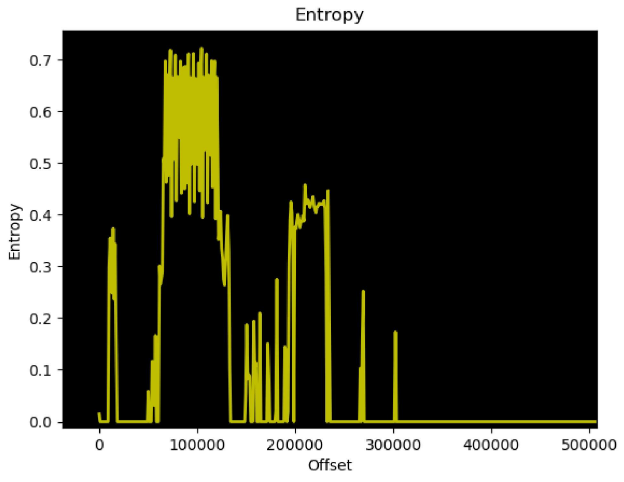


Figure E.3: Entropy of ram.img

E.4 sram

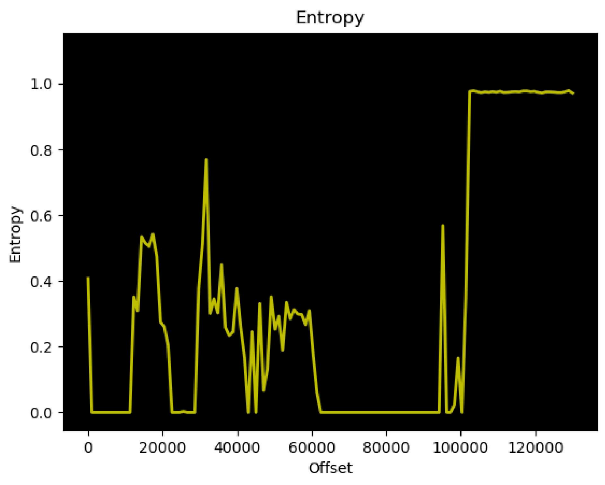


Figure E.4: Entropy of sram.img

E.5 system_memory_otp

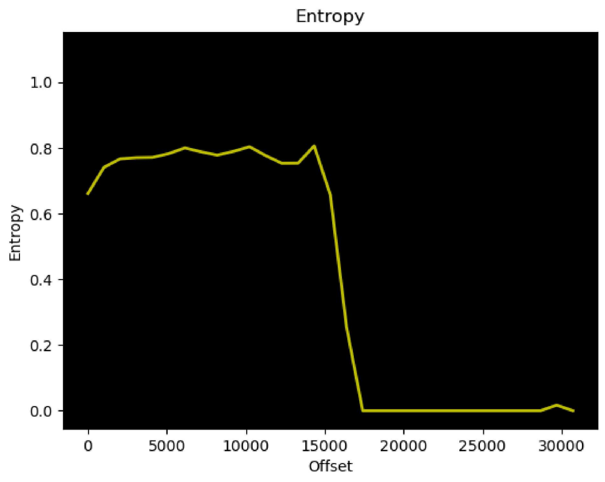


Figure E.5: Entropy of system_memory_OTP

Andreas Nilsen

NTNU

Kunnskap for en bedre verden