Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

Guillaume Nicolas Bour

Security Analysis of the Pacemaker Home Monitoring Unit: A BlackBox Approach

Master's thesis in Communication Technology
Supervisor: Marie Elisabeth Gaup Moe
May 2019

NTNU
Norwegian University of
Science and Technology

Guillaume Nicolas Bour

Security Analysis of the Pacemaker Home Monitoring Unit: A BlackBox Approach

Master's thesis in Communication Technology
Supervisor: Marie Elisabeth Gaup Moe
May 2019

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

NTNU
Norwegian University of
Science and Technology

Title: Security Analysis of the Pacemaker Home Monitoring Unit: A BlackBox Approach

Student: Guillaume Nicolas Bour

Problem description:

At the dawn of 2020, every object surrounding us starts to be connected in one way or another. Medical devices are no exception. Devices capable of remote monitoring have been on the market for several years now, and it is only a matter of time before connected intelligent prostheses are available. Connected medical devices are processing the patient's personal data and can sometimes act on the patient's condition directly. While no injuries nor death due to a cyberattack against a medical device have been reported publicly, it is easy to imagine that such an attack against medical facilities could have disastrous consequences. Namely, the WannaCry ransomware that infected multiple information systems all around the world in May 2017 and paralyzed the National Health Service in the UK, shed lights on a scary fact: cyber-attacks can kill.

This master thesis aims at analyzing the security of the pacemaker ecosystem. In particular, we will study the Home Monitoring Unit (HMU) which is an embedded device that interacts with the pacemaker and is in charge of remote monitoring. It is usually put in the patient's home and prevent her to visit her practician too often. Our objective is to determine if compromising the security of the HMU has an impact on the patient's safety or privacy.

We believe that the HMU might be misused to access patients' personal data and to tamper with it in a way that may affect patient safety. Indeed, this device is downloading the data from the pacemaker at a regular interval in order to upload them to the vendor's servers, where they will be made available to the practician.

In order to verify our assumptions, we plan to perform Black Box Testing on several HMUs from several vendors and evaluate the security level of those devices both as embedded devices and medical devices as part of an ecosystem. Commercial off-the-shelf equipment and open-source software will be used to achieve the testing. We will take care to stay within ethical and legal restrictions in this project. Special attention will be paid to the way we handle patient data, if any.

Responsible professor: Marie Elisabeth Gaup Moe, NTNU and SINTEF
Supervisor: Marie Elisabeth Gaup Moe, NTNU and SINTEF
Co-supervisor: Ravishankar Borgaonkar, SINTEF

Abstract

With the fast rise of Internet of Things (IoT), electronics systems are more and more connected, exposing new attack surfaces on devices responsible for human life, such as automobiles or medical devices. While companies in these areas are keeping the internals of their devices closed-source, the consumer or patient has no other choice than to trust the manufacturers when it comes to their security. Pacemaker monitoring systems have been on the market for several years now and are no exception to this rule. Previous research on the security of these monitoring systems indicates that they often have vulnerabilities that can be exploited by malicious adversaries in order to threaten the patient's life and privacy. This Master's thesis investigates the security of the Biotronik's Home Monitoring Unit, motivated by the absence of publicly available research on it. It was hypothesized that the Biotronik's HMUs contain security vulnerabilities that might help an attacker with physical access to the device to get patients' personal data or to be a threat to patients' safety. We conducted a Black Box Testing approach of the HMU's security and used off-the-shelves equipment along with open-source software to simulate the scenario of an external attacker having little knowledge about the device. The details of our work will be provided to the vendor to follow a coordinated vulnerability disclosure process. The research findings confirm that an attacker having physical access to the HMU has the ability to threaten both the patient's privacy and safety. We also showed that no special skills nor expensive equipments are required to perform the attacks. Our technical findings open the door to more work on the communication link between the HMU and the pacemaker, as long as future work on Biotronik's last generation of HMU. Our results also shed light on the importance for medical device manufacturers to consider security at every layer of their product and thus to include security as part of their development cycle.

Sammendrag

Oppblomstringen av Internet of Things (IoT) har ført til at elektroniske systemer har blitt mer or mer sammenkoblede, noe som åpner opp for nye angrepsflater for enheter som er ansvarlige for menneskeliv, for eksempel biler eller medisinsk utstyr. Leverandørene av slike enheter holder detaljene rundt implementeringen for seg selv, slik at forbrukeren eller pasienten ikke har et annet valg enn å stole på produsentene når det gjelder sikkerhet rundt produktet. Overvåkningssystemer for pacemakere har vært på markedet i flere år og er ingen unntak fra dette. Tidligere forskning på sikkerheten rundt slike overvåkningssystemer antyder at enhetene ofte inneholder sårbarheter som kan utnyttes av ondsinnede angripere og dermed true pasientens liv og personvern. Denne masteroppgaven tar for seg undersøkelser av sikkerheten rundt hjemmeovervåkningsenheter (HMU) fra Biotronik, og er motivert av mangelen på offentlig tilgjengelig forskning på disse enhetene. Hypotesen var at HMUer fra Biotronik inneholder sårbarheter som kan hjelpe en angriper som har fysisk tilgang til enheten til å få tak i persondata fra pasienter eller å være en trussel mot pasientens sikkerhet. Vi gjennomførte tester av sikkerheten rundt HMUene med en Black Box-tilnærming, og hyllevareutstyr ble brukt sammen med opensource programvare for å simulere et scenario med en ekstern angriper som hadde lite kunnskap om enheten. Detaljer rundt arbeidet vårt vil bli videreformidlet til leverandøren. Funnene fra forskningen bekrefter at en angriper med fysisk tilgang til en HMU, har evnen til å true både pasientens konfidensialitet og sikkerhet. Vi viste også at verken spesielle ferdigheter eller dyrt utstyr er nødvendig for å utføre angrepene. Våre tekniske funn åpner dører for videre arbeid rundt kommunikasjonskanalen mellom HMU og pacemakeren, i tillegg til videre arbeid på siste generasjon av HMUer fra Biotronik. Vårt arbeid kaster også lys på viktigheten av at produsenter av medisinsk utstyr tar sikkerhet på alvor ved å inkludere det som en del av utviklingssyklusen.

Preface

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

The research is performed by Guillaume Nicolas Bour and is part of collaboration between NTNU and SINTEF. It belongs to a larger project about medical device security. This work is supervised by Marie Elisabeth Gaup Moe and co-supervised by Ravishankar Borgaonkar, both at SINTEF's research division SINTEF Digital. Marie Elisabeth Gaup Moe is also the responsible professor for this project at NTNU. Another research is led by Anniken Wium Lie on the same topic, resulting in collaborations between the two projects.

Acknowledgments

Working on embedded medical device security involved working with all security layers, from hardware based security to infrastructure level security, going through reverse engineering and cryptanalysis processes. I would like to thank SINTEF for proposing such challenging projects. Having the opportunity to work in an area putting human's life at stake and where many ethical issues still need to be addressed was really eye-opening.

I also would like to thank the NTNU and the Institut National des Sciences Appliquées (INSA) Toulouse for giving me the opportunity to follow a double degree in both universities, and more specifically I must thank the people who worked on that agreement in each university.

The completion of this thesis would not have been possible without the support and nurturing of *Marie Elisabeth Gaup Moe* and *Ravishankar Borgaonkar* respectively our supervisor and co-supervisor at SINTEF. Their guidance, comments on my work and their extensive knowledge on the subject have revealed invaluable over the past year. I would also like to extend my thank to *Anniken Wium Lie* with whom we collaborated on several aspects of the projects and who helped me correct my abstract in Norwegian. Also, the "data extraction day" will remain as a very good memory of this thesis. I very much appreciate that *Éireann Leverett* gently lend me his hardware testing equipment, on which I relied all along this thesis. I'm also extremely grateful to *Snorre Aunet* and *Ingulf Helland* who took time to help me solder a proper connector on the board. My results would have been completely different without their kindness.

Many thanks also to my friend *Florent* for his valuable inputs and helpful advice shared during our lunch breaks. Special thanks as well to *Laura*, *Fabian* and *Gianmarco* for their constructive suggestions and review of this thesis.

Lastly, I would like to thank my parents and family for their unconditional support over the last five years of study. I would not be there today without them.

Contents

List of Figures **xiii**

List of Tables **xv**

List of Listings **xvii**

List of Acronyms **xix**

1 Introduction **1**
1.1 Context . . . 1
1.1.1 Medical devices definition . . . 1
1.1.2 Principle of an ICD . . . 2
1.1.3 The pacemaker's ecosystem . . . 3
1.2 Motivation . . . 5
1.3 Scope of the project . . . 6
1.4 Hypothesis and research questions . . . 8
1.5 Structure of the thesis . . . 9

2 Background **11**
2.1 Related work . . . 11
2.2 Summary of the Contributions . . . 14
2.3 Technical background . . . 14
2.3.1 UART communications . . . 14
2.3.2 JTAG . . . 16
2.3.3 Internet access for embedded devices . . . 18

3 Methodology **23**
3.1 Design science . . . 23
3.2 Black Box Testing . . . 24
3.2.1 Definition . . . 24
3.2.2 Steps of the methodology . . . 25
3.2.3 Use cases . . . 26
3.3 Threat model . . . 28

3.3.1 Definition . . . 28
3.3.2 Threat model considered . . . 28
3.4 Assessment . . . 29
3.4.1 Assessment criteria . . . 29
3.5 Ethical considerations . . . 31

4 HMU Security Analysis **33**
4.1 Preliminary HMU Security Analysis . . . 33
4.1.1 Analysis of the Merlin@Home HMU . . . 33
4.1.2 Analysis of the Medtronic HMU . . . 37
4.2 Security Analysis of the CardioMessenger II-S TLine . . . 38
4.2.1 Hardware analysis . . . 38
4.2.2 Debugging interfaces . . . 40
4.2.3 Eavesdropping and network emulation . . . 52
4.2.4 Memory analysis and reverse engineering . . . 60
4.3 Security Analysis of the CardioMessenger II-S GSM . . . 73
4.3.1 Hardware analysis . . . 73
4.3.2 Eavesdropping and network emulation . . . 74
4.3.3 Mobile network . . . 80
4.4 Summary of the findings on the CardioMessenger II-S . . . 83
4.5 Additional results . . . 86
4.5.1 Hardware analysis of the CardioMessenger 3G . . . 86
4.5.2 Hardware analysis of the CardioMessenger LLT2 . . . 87
4.6 Conclusion of the analysis . . . 89

5 Mitigation **91**
5.1 Mitigation of our findings . . . 91
5.1.1 Mitigating the physical tampering . . . 91
5.1.2 Mitigating the network emulation . . . 92
5.1.3 Mitigating the reverse engineering . . . 93
5.2 Best practices . . . 94
5.2.1 Embedded security pyramid . . . 94
5.2.2 Security requirements for embedded devices . . . 94
5.2.3 Recommended guidelines . . . 100

6 Discussion **103**
6.1 Implications of this work . . . 103
6.1.1 Attack scenarios . . . 103
6.1.2 Hypothetical attack scenarios . . . 104
6.1.3 Ethical considerations . . . 105
6.2 Remaining problems in the IoT and Medical Devices . . . 106
6.2.1 Best practices are not applied . . . 106

6.2.2 The medical security trade-offs . . . 107
6.2.3 Scoring systems . . . 108
6.2.4 Certifications . . . 109
6.3 Future work . . . 110

7 Conclusion **113**

References **115**

Appendices

A Methodologies and procedures **121**
A.1 Discovering the JTAG interface . . . 121
A.1.1 Updating the JTAGulator firmware . . . 121
A.2 Interacting with a JTAG interface . . . 122
A.2.1 Using the Shikra as the interface . . . 122
A.2.2 Using a Raspberry Pi as the interface . . . 123

B Tools developed **125**
B.1 Serial line communication scripts . . . 125
B.1.1 Extended monitoring script for serial line communication . . . 125
B.1.2 Modem emulation script . . . 127
B.2 Analysis scripts . . . 131
B.3 OpenOCD configuration files . . . 141

C Detailed listings **143**
C.1 CardioMessenger II-S T-Line . . . 143
C.1.1 Modem's configurations . . . 143
C.1.2 JTAGulator . . . 147

List of Figures

1.1 Picture of the pacemaker in our lab . . . 3
1.2 Diagram of the vendor's pacemaker ecosystem. . . . 4
1.3 Scopes around the HMU . . . 7

2.1 Communication between two UART . . . 15
2.2 Relation between the bit period and the Baud rate . . . 15
2.3 The Joint Test Action Group (JTAG) state machine . . . 17
2.4 Connection between a DTE and a DCE . . . 18
2.5 Data exchange's sequence diagram between two DTE . . . 20
2.6 AT commands issued to establish a connection between two DTE . . . 21

3.1 Diagram of the Black Box Testing Methodology . . . 24
3.2 Black Box Methodology iterative cycles used in our project . . . 27

4.1 Outside and inside of the Merlin@Home . . . 34
4.2 UART pins on the Merlin@Home . . . 35
4.3 Root access on the Merlin@Home . . . 36
4.4 Possible debug pins on the Medtronic board . . . 37
4.5 Mapping of the Medtronic UART Pins . . . 37
4.6 Inside of the CardioMessenger II-S T-Line . . . 39
4.7 Possible debug ports on the board . . . 41
4.8 Mapping of the CardioMessenger II-S UART Pins . . . 41
4.9 Possible JTAG pins on the board . . . 43
4.10 Details of the JTAG connector added to the board . . . 43
4.11 The JTAGulator from Grand Idea Studio . . . 44
4.12 Connection of the JTAGulator to the device . . . 44
4.13 Using a Raspberry Pi as JTAG adapter . . . 47
4.14 Mapping of the CardioMessenger II-S JTAG Pins . . . 47
4.15 Interaction with the chip through JTAG . . . 49
4.16 Available information through JTAG . . . 49
4.17 AT91RM9200 memory map . . . 50
4.18 Modem RS-232 pins . . . 52

4.19 The HMU is accessing the internet using an APN 54
4.20 Structure of the protocol's header 59
4.21 Entropy distribution of the bootloader's dump 63
4.22 Entropy distribution of the ram and of the flash memory 63
4.23 Decompiled code - Pacemaker communication 65
4.24 Partial decompiled code of the PackData function 66
4.25 Detailed structure of the communication protocol's packet 68
4.26 Content in RAM before and after PackToEncryptionLayer 69
4.27 Different encryption algorithms used in PackToEncryptionLayer 72
4.28 Inside of the CardioMessenger II-S T-Line 73
4.29 Guess of the network architecture between the HMU and the servers . . 76
4.30 Network diagram of the emulated network 77
4.31 Timeline of the SMS sent by the HMU 81
4.32 Decompiled code of the function processing SMS 82
4.33 Labelled pins on the back of the CardioMessenger 3G 86
4.34 Inside of the CardioMessenger LLT2 88

5.1 Embedded security pyramid 95

List of Tables

2.1 Description of the JTAG signals 16

3.1 Potential attackers and their capabilities 29
3.2 Findings' criticality levels 31

4.1 Raspberry Pi's JTAG connection using OpenOCD 48
4.2 Pattern of the gathered data 59
4.3 Findings summary (1) 84
4.4 Findings summary (2) 85

A.1 Raspberry Pi's JTAG connection using OpenOCD 124

List of Listings

2.1 PPPD chat script example 21
4.1 Partial booting information from the Merlin@Home 35
4.2 Default Merlin@Home's kernel options on boot 36
4.3 Output captured on the UART during the boot process 42
4.4 Performing an IDCODE scan with the JTAGulator 45
4.5 Performing an BYPASS scan with the JTAGulator 45
4.6 Reading the device ID with the JTAGulator 46
4.7 Dumping memory from the microcontroller 50
4.8 Visualizing strings sent in memory 51
4.9 Eavesdropping on the modem's pins 53
4.10 Spoofing the modem to interact with the microcontroller 55
4.11 Entropy of the data gathered 58
4.12 Hex dump of the first bytes of four data chunks gathered 59
4.13 Gathered strings related to the GSM versions 64
4.14 Brute forcing the AES key 69
4.15 Decrypting the data sent by the HMU 70
4.16 AT commands sent by the microcontroller to the GSM modem . . . 74
4.17 The HMU trying to contact the server 76
4.18 The HMU contacting the fake server 77
4.19 Using the AES key from the TLine version on the data 78
4.20 Bruteforcing the AES key using the firmware file 79
4.21 SMS are sent by the HMU 80
4.22 SMS are read by the HMU 81
4.23 Information's summary output by our APDU parser 88
B.1 CardioMessenger Serial Monitor Script 125
B.2 CardioMessenger Serial Modem Script 127
B.3 CardioMessenger Data Decryption Script 131
B.4 APDU Parser 137
B.5 Basic connection to a target 141
B.6 Dumping the memory of a target 142
C.1 Default modem's configuration 143
C.2 Modem's configuration 145

C.3 Pins determination using the JTAGulator 147

List of Acronyms

AES Advanced Encryption Standard.

APDU Application Protocol Data Unit.

APN Access Point Name.

BGA Ball Grid Array.

BSI German Federal Office for Information Security.

CIA Confidentiality, Integrity and Availability.

CIANA Confidentiality, Integrity, Availability, Non-repudiation and Authentication.

CPU Computer Processing Unit.

CVS Concurrent Versions System.

CVSS Common Vulnerability Scoring System.

DCE Data Communication Equipment.

DDoS Distributed Denial of Service.

DES Data Encryption Standard.

DoS Denial of Service.

DR Data Registers.

DSL Digital Subscriber Line.

DSP Digital Signal Processor.

DTE Data Terminal Equipment.

ELF Executable and Linkable Format.

EU European Union.

FDA US Food and Drug Administration.

FTP File Transfer Protocol.

GPRS General Packet Radio Service.

GSM Global System for Mobile communications.

HMU Home Monitoring Unit.

I^2C Inter-Integrated Circuit.

ICD Implantable Cardioverter Defibrillator.

IMD Implantable Medical Device.

INSA Institut National des Sciences Appliquées.

IoT Internet of Things.

IP Internet Protocol.

IR Instruction Register.

ISP Internet Service Provider.

JTAG Joint Test Action Group.

LCP Link Control Protocol.

MitM Man in the Middle.

NCPs Network Control Protocols.

NSA National Security Agency.

NSD Norwegian Centre for Research Data.

NTNU Norwegian University of Science and Technology.

OS Operating System.

OSSTMM Open Source Security Testing Methodology Manual.

PAP Password Authentication Protocol.

PCB Printed Circuit Board.

PKI Public Key Infrastructure.

PPP Point-to-Point Protocol.

PSTN Public Switched Telephone Network.

RAM Random Access Memory.

RFC Request for Comments.

RS-232 Recommended Standard 232.

RSA (Rivest Shamir Adleman.

RSS-MD Risk Scoring System for Medical Devices.

SDR Software Defined Radio.

SIM Subscriber Identity Module.

SMTP Simple Mail Transfer Protocol.

SPI Serial Peripheral Interface.

SSH Secure Shell.

SSL Secure Sockets Layer.

SWD Serial Wire Debug.

TAP Test Access Port.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

UART Universal Asynchronous Receiver-Transmitter.

VPN Virtual Private Network.

Chapter 1
Introduction

1.1 Context

1.1.1 Medical devices definition

Defining what is a medical device is not an easy task. The US Food and Drug Administration (FDA) defines a Medical Device as "an instrument, apparatus, implement, machine, contrivance, implant, in vitro reagent, or other similar or related article, including a component part or accessory which is: recognized in the official National Formulary, or the United States Pharmacopoeia, or any supplement to them, intended for use in the diagnosis of disease or other conditions, or in the cure, mitigation, treatment, or prevention of disease, in man or other animals, or intended to affect the structure or any function of the body of man or other animals, and which does not achieve its primary intended purposes through chemical action within or on the body of man or other animals and which is not dependent upon being metabolized for the achievement of its primary intended purposes" [55]. This definition is difficult to understand and while some devices are definitively included in it (such as a pacemaker), others stand at the border like a treadmill for instance. This question of knowing whether or not a device is considered as a medical device can have an impact on the product itself as it will not require the same certifications before going to the market.

Depending on their risk for patient safety, medical devices are classified by the FDA into three classes, the class I being the one with the lowest risk on the patient safety and class III the one with the greatest [55]. According to that classification, a treadmill is be considered as a device belonging to class I while an Implantable Cardioverter Defibrillator (ICD) is a class III device.[1] Depending on the risk, the device and the manufacturer, a notification to the FDA will be required to introduce

[1] According to the FDA search database available at https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfPCD/classification.cfm

such a device to the US market. For other devices (typically class III devices), a premarket approval might be required.

Currently, in the European Union (EU), three different directives need to be followed by medical devices, but these directives have been issued in the nineties and are going to be replaced by two new regulations that have been adopted in April 2017 [17]. These new regulations will enter in force in spring 2020. According to the European Commission, these new directives will improve the safety of medical devices by introducing "stricter pre-market controls for high-risk devices", "more transparency" and "strengthened post-market surveillance" to quote but a few. The cybersecurity aspect is also taken into account in that new regulation. Indeed, new requirements regarding information security are developed in Annex I of the paper, entitled "General Safety and Performance Requirements" [18]. Minimum requirements for a medical device in terms of IT security are given, along with principles that must be included in the product development's cycle, information security and risk evaluation being part of those. Also, the possible risks due to a malicious interaction with the device should be anticipated.

1.1.2 Principle of an ICD

An ICD is a medical device implanted in the patient's body to control patient's heartbeats, usually in case of arrhythmia.[2] ICDs are used for patients with strong arrhythmia, i.e. those whom condition can lead them to faints or even cardiac arrests. The device monitors the patient's heart rate and sends an electrical pulse when it detects an anomaly [53].

A ICD is presented in Figure 1.1. The device is of the size of a credit card and has one, two or three wires that are called leads. As explained by the Texas Heart Institute, a pulse generator along with other electronic circuits and a battery are contained inside the implant [53]. The difference between a pacemaker and an ICD is that an ICD can deliver shocks, while pacemakers only correct the heart's rhythm with small electrical stimuli. The shocks delivered by an ICD can be felt by the patient, and he is usually asked by his doctor to report when he got one or more. From now on, we will use only to the term "pacemaker" to refer to both "pacemakers" and "ICD" as both use the same communication protocol with the HMU.

Having to report and then to visit the doctor each time an event occurs can be annoying for patients. That is why pacemaker's manufacturers have connected the devices wirelessly. That way, a patient can use an external device to gather the data

[2] According to the National Heart, Lung, and Blood Institute, "an arrhythmia is a problem with the rate or rhythm of the heartbeat. During an arrhythmia, the heart can beat too fast, too slowly, or with an irregular rhythm. When a heart beats too fast, the condition is called tachycardia. When a heart beats too slowly, the condition is called bradycardia." [40]

Figure 1.1: Picture of the pacemaker in our lab

from his pacemaker during the night and automatically send them to a server where the doctor can see them. That allows the doctor to be alerted faster in case of a problem and prevent the patient from unnecessary visits.

1.1.3 The pacemaker's ecosystem

An Implantable Medical Device (IMD) is usually not functioning on its own and requires a whole ecosystem around it to make it work correctly and efficiently. This ecosystem is composed of multiple devices which, for the Biotronik's pacemaker ecosystem we are studying, are the following:

The pacemaker Implanted in the patient's body, this is the main device of the ecosystem. It generates an electric impulse that helps regulate the heart rate. Programmable pacemakers allow practitioners to select the appropriate pacing for every patient.

The programmer The programmer is an external computer used by a practitioner at the hospital to program the pacemaker. This device requires proximity with the pacemaker, which is achieved thanks to the programming head. The communication remains wireless though.

The Home Monitoring Unit The HMU is a device aiming at easing the patient's life by preventing them from visiting their practitioners too often. Indeed, placed in the patient's home, this device is in charge of downloading the data from the pacemaker and sending to the vendor's servers for the practitioner to access them remotely.

The Operator Network The HMU needs a way to access the internet in order to communicate with the vendor's servers. Depending on the HMU (see below), the internet can be accessed using a mobile network such as GSM or 3G but also using a regular telephone line.

Vendor's Servers These are the servers the HMU connects to in order to export the patient's data. This is achieved using the operator network. These servers can be accessed by the practitioner through an online platform.

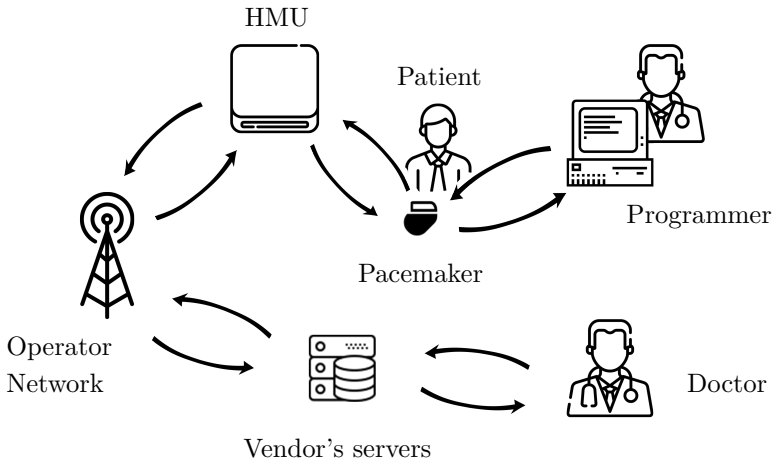


Figure 1.2: Diagram of the vendor's pacemaker ecosystem.

Several versions of the HMU exist and can be used with multiple pacemakers from the same vendor. Also, the programmer is designed to be used with several pacemaker from Biotronik. We have five main types of HMU of that vendor in our lab. They do not have all the same functionalities and the attack surfaces might be different from one to another. The five HMU's types are the following:

- CardioMessenger LLT
- CardioMessenger LLT II
- CardioMessenger II-S GSM
- CardioMessenger II-S TLine
- CardioMessenger 3G Smart

The CardioMessenger LLT, LLT II, II-S GSM and 3G Smart are using a mobile network to communicate with the Biotronik's Services Center while the CardioMessenger II-S TLine is using a regular telephone line.

1.2 Motivation

Implantable Medical Device are devices implanted inside the human body to help the patient and prevent life-threatening conditions. Implantable devices such as insulin pumps are a real game changer for people suffering from diabetes as they monitor and regulate the insulin level for the patient, preventing them to do blood analysis multiple times a day. Regarding pacemakers, 1.14 Million devices were implanted in 2016, and it is estimated that this number will reach 1.43 Million by 2023.[3] These devices aim at improving patients' lives and are more and more connected with that purpose in mind. The HMU is a good example as it prevents patients from having to visit their practitioner too often as the relevant data is automatically sent to the cloud and the practitioner is warned in case of a suspicious event.

Contributing to securing medical devices is the main reason we are interested in this specific topic. Security of such medical devices has not been publicly debated as a real concern until recently. IMD vendors tend to base part of the device security by practising "security by obscurity." That means that most of the technology and protocols used in the IMD ecosystem are proprietary and kept secret. This is, however, very bad practice in security and it is understandable that patients want to know on what technology and security mechanisms their lives rely. Hence, the case opposing St. Jude Medical to Muddy Waters, MedSec Holdings et al, in the United States District Court for the District of Minnesota in 2016 [44], shed the light on the fact that medical devices manufactured by one of the biggest medical companies can be unsecured and have some vulnerabilities that, if exploited, can have disastrous consequences on the patient's life [10]. Even though St. Jude Medical claimed that Muddy Waters and MedSec Holdings et al. acted with financial interests, investigations revealed that vulnerabilities in St. Jude Medical's devices are a reality. As a result, the FDA issued a security notification affecting 465,000 pacemakers in the US [19]. A firmware update was required. Another example, the WannaCry ransomware[4] that stroke the world in May 2017 also impacted medical services. The National Health Service in the UK has been completely stuck and the UK parliament detailed in a report published in 2018 how patients had to be moved in order to get their surgery and how 20,000 appointments had to be postponed or cancelled due to the shutdown of the IT systems [41]. Those two examples demonstrate well how a cyber-attack can put human lives at stake. Scenarios in which an attacker can hack an ICD and threaten their victim's life is not fictional anymore. One can wonder if an attack on an IMD might ever occur and it is true that the probability of such a targeted attack is low. In fact, no cyber-attack has been publicly recorded against a medical device. This is particularly true for an ordinary person but some "high

[3] According to www.statista.com

[4] A ransomware is a malicious software that blocks some data (usually by using encryption) and ask a ransom to be paid in exchange of the decryption key.

valuable" targets might be concerned. In fact, some of these people have already taken some precautions against that kind of attacks. Dick Cheney for instance, the former Vice President of the United States asked his doctor to disable the wireless functionality of his pacemaker [43]. His fear is understandable given the ingenuity used by some security agencies around the world to assassinate targets (the assassination of Georgi Markov in 1978 in London, to name only this one). It is hard to believe they haven't thought about hacking medical devices or already used it.

Working on this topic is also motivated by the fact that the IoT is developing fast and will have more and more impact in the future. Trade-offs and challenges to secure those embedded devices with small energy capability are similar to the one met with securing IMD. Security guidelines used in the IoT world could be useful for developing more secure IMD and vice-versa. Moreover, human beings are more connected using phones, smartwatches, sensors, etc. and it is only a matter of time before the technology is integrated into the human body. It is then crucial to design those devices with security in mind ("secure by design") and not to reproduce the mistakes that have been done too often in the past, such as considering obfuscation as a viable security technique.

Finally, this project stands at the border between information security and medicine, raising multiple ethical questions. It lets us think not only about our relationship with technology but also about the future of this relation. Technology is evolving faster than ever and prostheses might become connected and intelligent, becoming thereby new targets for attackers. Dealing with IMD security also raises the dilemma between safety and security which must be considered by security researchers in order to provide new innovative solutions that also provide adequate security levels. Researchers conceiving new medical devices will have to face the same dilemma as the one met in autonomous driving. Indeed, designing such equipments that can have impacts on human lives also requires to make choices and design decisions that can have fatal issues. This topic is discussed in Chapter 6.

1.3 Scope of the project

As mentioned in 1.1, an IMD rarely runs on its own and depends on a whole ecosystem. This ecosystem might contain multiple other devices and machines, from a programmer in the hospital to a whole cloud-based architecture, to handle the gathered data and allow monitoring. Our project is focusing on one element of that ecosystem which is the HMU. This device, even though it is not compulsory, is really interesting in terms of security as it directly interacts with the IMD and handles the patient's data. Most importantly, it has been proven that an HMU can be transformed into a "weapon" and a threat to a patient's life, by using it to drain the pacemaker's battery [46].

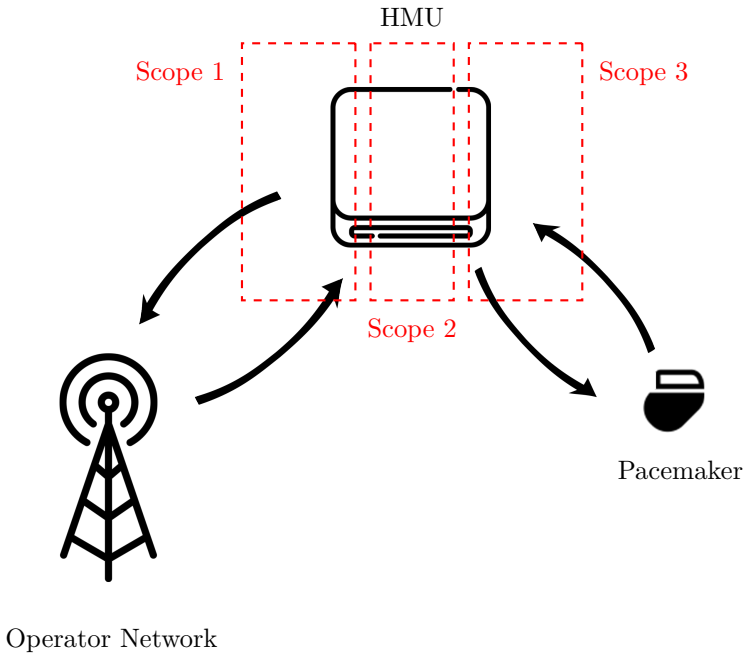


Figure 1.3: Scopes around the HMU

Our project is also focusing on one specific brand of IMD, which is Biotronik. To our knowledge, there are a very few published research papers on that vendor's products, even though it is one of the four most used vendor in Norway in 2017 regarding pacemakers and ICD [26]. That is mainly why we are focusing on that vendor. The equipment we got has mainly been acquired through eBay and donations.

As presented in 1.1.3, the Biotronik's HMU is interacting not only with the pacemaker but also with Biotronik's servers. Three main attack surfaces can be outlined here and are presented in Figure 1.3:

Scope-1 The interface between the HMU and Biotronik's servers.

Scope-2 The HMU itself as a standalone embedded device.

Scope-3 The interface between the HMU and the pacemaker

Focusing on the three scopes in a single master thesis was not possible and that is why, in this project, we decided to focus on the HMU itself, i.e. scope 2. Our original goal was to study all five types of HMU and compare their level of security. However, we focus our research mainly on the CardioMessenger II-S in both

its versions (Global System for Mobile communications (GSM) and TLine). Some testing has been performed on the other ones, but this remains superficial.

The interface between the HMU and Biotronik's servers using the Mobile network is studied by Anniken Lie this semester [34]. Their project aims at developing a fake base station to spoof a real base station for the HMU (scope 1) and perform a security tests based on this lab set-up.

Even though our scopes are distinct, many collaborations took place between our projects. Indeed, our findings on the CardioMessenger TLine and GSM along with the Fake Base Station set up by Lie allowed us to perform more interactions with the devices. We also provided Lie with the results of some of our hardware testing. These additional results are described more in detail in Section 4.5.

1.4 Hypothesis and research questions

The HMU is part of the pacemaker's ecosystem and is interacting with it in order to monitor the patient's heart. It might then contain personal data of the patient, but might also be used to perform attacks attempting to the patient's safety as it has already been proven in previous studies [46]. Given that, it is vital that the HMU, as part of the ecosystem, is secure.

Our main hypothesis is that *the Biotronik's HMUs contain security vulnerabilities that might help an attacker with physical access to the device to get patients' personal data or to be a threat to patients' safety.*.

Based on that assumption and on our motivations, we define three main questions for our work to be a contribution to the security of the pacemaker ecosystem.

Q1 Are the vendor's HMUs secured regarding the patient's safety?

Q2 Are the vendor's HMUs secured regarding the patient's privacy?

Q3 What can vendors systematically do to improve their devices' security?

Answering these questions points out two main objectives:

O1 Testing the vendor's HMUs against multiple hypotheses to determine if they can be used to attempt on the patient's safety or to steal confidential data.

O2 Defining a framework/methodology to assess embedded devices interacting with IMD.

1.5 Structure of the thesis

Chapter 1 presents the context and scope of our research along with the motivations to work on the topic of medical devices. In Chapter 2, we provide the reader with information regarding the previous studies that have been performed on medical devices. The technical background useful to understand the results of our work is also provided. The methodology of our research is explained in Chapter 3 along with the ethical issues that must be considered when working in an area where human's lives are at stake.

Chapter 4 gives an overview of our results on the HMU security. We detail our findings along with their impact on the ecosystem. Countermeasures and best practices are outlined in Chapter 5. Chapter 6 sheds the light on the impact of our work but also on the remaining problems in the medical devices and more generally in the IoT sphere. Future work and leads for future work on the vendor's ecosystem are also presented in that chapter. Chapter 7 summarizes and concludes our work.

For the sake of readibility, methodologies and procedures related to the configurations used during the security assessment have been explained in Appendix A. Tools developed during this Master's Thesis are available in Appendix B. Finally, detailed listings encountered during our research are presented in Appendix C.

Chapter 2
Background

2.1 Related work

Although IMD exist since the sixties [8], their security has only been a concern for the last decade. Back in 2008, Halperin et al. [24] performed radio-based attacks against a commercial pacemaker. Using Software Defined Radio (SDR) they managed to partially reverse-engineer the proprietary protocol used on the communication link between a programmer and a pacemaker. With that knowledge, they were able to show that it is possible to compromise both patient's privacy and safety. Their work has been the foundation for many others that contribute to the improvement of IMD ecosystems security.

Two types of approaches exist when it comes to analyzing the pacemaker ecosystem. The first one aims at giving an overview of the problem with a high level of abstraction. The global security of the different pacemaker ecosystems is analyzed, and the challenges are highlighted. General solutions are also proposed to guide IMD companies to produce more secure devices.

Camara et al. [13] give an overview of the security objectives regarding IMDs along with the current protections that have been developed. They also highlight the fact that multiple constraints exist around the development of a medical device ecosystem. Energy, storage, computing and communication are some limitations of those devices. In their review of the medical devices' ecosystems, Zheng et al. [59] reach the same conclusions regarding the trade-offs that exist in the pacemaker ecosystem. The first constraint would be the security vs the accessibility. Indeed, the devices should not threaten the patient's life, especially if an urgent surgery is required and the pacemaker must be deactivated. Also, securing checkup access while having emergency access is a challenge. The pacemaker cannot be accessed the same way in both situations and the battery life of the device must be taken into account. Finally, the limited resources of the IMD are opposed to the implementation of strong security requirements. Cryptographic computations require power, and this

can be abused by an attacker to launch Denial of Service (DoS) attacks, leading to surgery for the patient to replace the battery, which could threaten the patient's life. Those papers help us to understand what are the challenges of securing pacemaker ecosystems. They also provide us with a good overview of the attackers' profiles along with their motivations. Those challenges and trade-offs are further discussed in Chapter 6.

Some research focuses on a specific device and/or specific vendor. The goal of these papers is to show the (in)security of medical devices by demonstrating the feasibility of the attacks and their impact on the patient in terms of safety and privacy.

A technical report from Bishop Fox [11] about the case opposing St Jude Medical to Muddy Waters in 2016 gives an overview of the flaws that have been discovered on one of the St Jude Medical's HMU and on the impact these vulnerabilities have on the pacemaker ecosystem. As detailed in the report, using discovered flaws, they were able to fully compromise the HMU and to launch attacks against the pacemaker, potentially causing a threat to the patient's life. They also describe a scenario in which it would be possible to compromise multiple devices and to launch large-scale attacks against pacemakers leading to disastrous consequences. Even though some parts of the report have been redacted for a security reason, it still gives an overview of the attack vectors used and the methodology they have used to implement and verify some attacks. Marin [36], while not targeting a specific brand or device in his paper, gives technical details on how to implement attacks against IMDs. He describes known attacks against IMDs such as hardware-related attacks and provides us with methodologies that can be used in a reverse engineering process. Finally, he gives indications of the existing solutions to develop secure IMD ecosystems.

Some papers and articles present more interest in our project as they can be used as a foundation for our work. Rios and Butts [46] evaluated the security of four different HMUs from four different vendors. Vendors are not explicitly cited, but we can guess that one of them is the vendor we are studying by comparing the pictures of the HMUs' boards. This information was helpful when we had to identify possible attack vectors. The authors are indeed testing multiple attacks and access methodologies to the HMUs. They also give a list of questions that aims at helping vendors evaluating the security of their ecosystem when developing it. Those questions have been used as both a start in the attack phase and as the foundation for the testing guide detailed in Chapter 5. Marin [36], as explained in the paragraph above, gives multiple information regarding IMD security and his article is recent, meaning that the information he gives is not outdated. He describes methods used for reverse-engineering and for firmware extraction along with the tools and the precautions to take. In their Master's Thesis, Kristiansen and

Wilhelmsen [33] already worked on the vendor's pacemaker ecosystem. Their work constitutes, in fact, a foundation for our project. While they have focused their study on the programmer, the results they have obtained could have led to discoveries on the HMU. Indeed, one of the contributions of their thesis is a virtual machine of the programmer, which means that it can be used to reverse-engineer protocols used between the programmer and HMU for instance. Indeed, according to Block [11], implementations can sometimes be the same between those two entities. More recently, Marin et al. [37] have disclosed new vulnerabilities on ICDs security. The research focuses on the communication channel between the ICD and the programmer and they were able to successfully reverse engineer the communication protocol used. As a result, they managed to activate an ICD and to launch privacy attacks, DoS attacks along with spoofing and replay attacks.

Last but not least, Borgaonkar et al. [12] interest themselves in femtocell security testing. Even though it is not directly linked to the IMD ecosystem security, a femtocell is an embedded system and in that, assessing its security presents similarities with the HMU security testing, especially regarding the approach. In their paper, they are first studying the security of the femtocell and then the impact of the vulnerabilities they found on the whole system, including the end-user and the operator. Both these entities exist as well in the pacemaker ecosystem, being the patient and the vendor's infrastructure. Their approach is then particularly interesting for us and has been reproduced partially as explained more in detail in Chapter 3.Another aspect of the research of Borgaonkar et al. [12] is the usage of an end-device to attack the core network. Using physical level vulnerabilities, they were able to get root access on the femtocell and thus to target the network. This highlights the need for vendors to consider physical access as a possible attack vector. This is indeed a very common way for attackers to breach an infrastructure. Already in 1999, Kocher et al. [31] performed differential power analysis on a smart card and were able to gather secret information such as cryptographic keys or PIN code. Physical layer security should always be considered in IoT device, as they can also constitute a second stage vulnerability, i.e. a vulnerability that is used in a second time. Indeed, an attacker who can gain access to a system remotely using a breach in a remote system could potentially escalate her privileges using a physical vulnerability discovered through physical analysis of the device. This threat has been proven real by Miller and Valasek [39] when they hacked a 2014 Jeep Cherokee and demonstrated that "a remote compromise of a vehicle could result in sending messages that could invade a driver's privacy and perform physical actions on the attacker's behalf". Security researchers at Baidu will present in August 2019 their work about 4G modem security. According to their abstract, all 4G modems have vulnerabilities, from remote access with weak passwords to command injection by SMS [49]. These vulnerabilities linked to hardware vulnerabilities on an IoT device, and more specifically on automotive and medical devices, can have a significant

impact.

Since the initial work conducted by Halperin et al. in 2008, researchers started to study the security of IMDs, putting lights on that topic and leading agencies and organizations such as the FDA or MITRE to come up with standards and guidelines regarding IMDs implementation of security [6, 5]. However, multiple devices already implanted do not meet those requirements and may lack of security, threatening their owner's life. Several research has studied vulnerabilities in HMUs and their impact on the pacemaker's ecosystem security but to our best knowledge, the paper of Rios and Butts and the one of Kristiansen and Wilhelmsen are the only research that describe attacks against Biotronik's products.

2.2 Summary of the Contributions

Our research investigates the security level of Biotronik's HMUs and evaluates the impact of a compromised HMU on patients' safety and privacy. We propose a Black Box Testing approach to the security of the CardioMessenger II-S, based on the assumptions that the attacker has low budget equipment and very little knowledge about the system. To our best knowledge, this thesis hereby describes the first known vulnerabilities in the Biotronik's CardioMessenger II-S. The discovered vulnerabilities can be chained to impact both patients' safety and privacy and raise several ethical concerns about the security level in medical devices today. Moreover, these vulnerabilities lie in almost all layers of the device and do not have easy remediations for boards already deployed. We still describe mitigation solutions and guidelines that help in raising the overall security level of an embedded device. Finally, we discuss the remaining problems in the medical device area and in IoT in general.

2.3 Technical background

This section aims at providing the reader with some technical background information that is useful to understand the results of our work as well as their implications.

2.3.1 UART communications

A Universal Asynchronous Receiver-Transmitter (UART) is a hardware component often directly included in microcontrollers that enable the chip to use serial communications. It means that the UART will take a value on a data bus and send it over a serial line. The communication takes place using at least two signals (Tx and Rx) and is full duplex. It is also important to note that there is no clock signal, which is not the case in other common communication protocols such as the Serial Peripheral

Interface (SPI) or Inter-Integrated Circuit (I^2C). The configuration has to be done by the user.

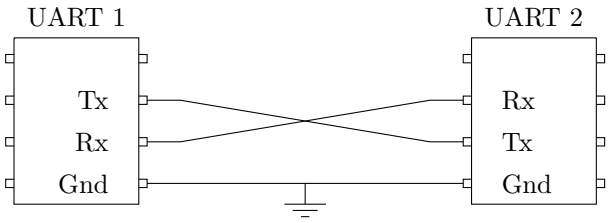


Figure 2.1: Communication between two UART

A UART "packet" is composed of the following elements:

- 1 start bit
- 5 to 8 bits of data (9 if no parity bit is used)
- 0 or 1 parity bit (odd or even)
- 1 or 2 stop bit(s)

As explained above, there is no clock signal, and both UARTs must be configured to use the same Baud rate. The Baud rate is the speed at which data can be transferred and is expressed in bits per second. Common Baud rates are 9 600, 57 600 or 115 200. Figure 2.2 shows the relation between the bit period and the Baud rate.

$$bit\ period = \frac{1}{Baud\ rate}$$

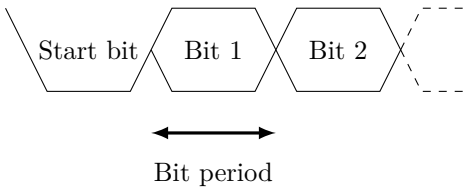


Figure 2.2: Relation between the bit period and the Baud rate

It is also important in the context of this thesis to know that when no data is sent or received on Tx and Rx, those lines are held "high".

Using UART communication has several advantages. There is no clock signal, only three wires (Tx, Rx and GND) are required and some error checking is implemented. Also, it is well documented and easy to use. Drawbacks exist as well. Indeed, the size of the data frame is limited and there is no slave support. Finally, the Baud rate must be set within 10% on each UART for the decoding to work properly.

In some case, UART pins are labelled on the board and it is easy to establish a connection with the device. However, finding the UART pins can sometimes be challenging as some vendors try to hide them. The methodology used in this thesis to determine the UART pins is described in Chapter 4.

2.3.2 JTAG

JTAG, or "IEEE Standard Test Access Port and Boundary-Scan Architecture", is an industry standard used to perform integrated circuit testing [4, 29]. With Printed Circuit Board (PCB) being more and more complex and composed of multiple layers, testing circuits using the traditional bed of nails was not convenient anymore. That is why this new standard was developed.

As explained by Rath, JTAG-compliant devices have one Instruction Register (IR), several Data Registers (DR) and a Test Access Port (TAP) Controller that handles the test operations [45]. Scan cells connected to a component's inputs and outputs allows boundary scans: the output signal of a component is connected to the input of the next one, creating a serial shift register [45]. Table 2.1 presents the different signals that are defined in JTAG.

Signal	Acronym	Description
Test Data Input	TDI	The data that goes in the component
Test Data Output	TDO	The data that goes out of the component
Test Mode Select	TMS	The signal that controls the JTAG state-machine (presented in Figure 2.3)
Test Clock	TCK	The Clock signal (going in the component)
Test Reset	nTRST[1]	The reset signal (optional)

Table 2.1: Description of the JTAG signals

Many of microcontroller's manufacturers are building debug features on top of JTAG. This is the case for the microcontroller we are studying the most in this thesis. It is featured with the support for JTAG-ICE. That has several advantages such as the support for both hardware and software breakpoints but also single stepping. One can have the full control over a microcontroller and more generally over a device if connected through JTAG. Indeed, it is then possible to sometimes read and write memory, modify the value of the Computer Processing Unit (CPU) registers, etc. From now on we will only refer to those debug features as JTAG.

When explaining what is JTAG, we also have to mention Serial Wire Debug (SWD) which "replaces the 5-pin JTAG port with a clock and single bi-directional

[1]The "n" in nTRST means it is a negative logic: active when low.

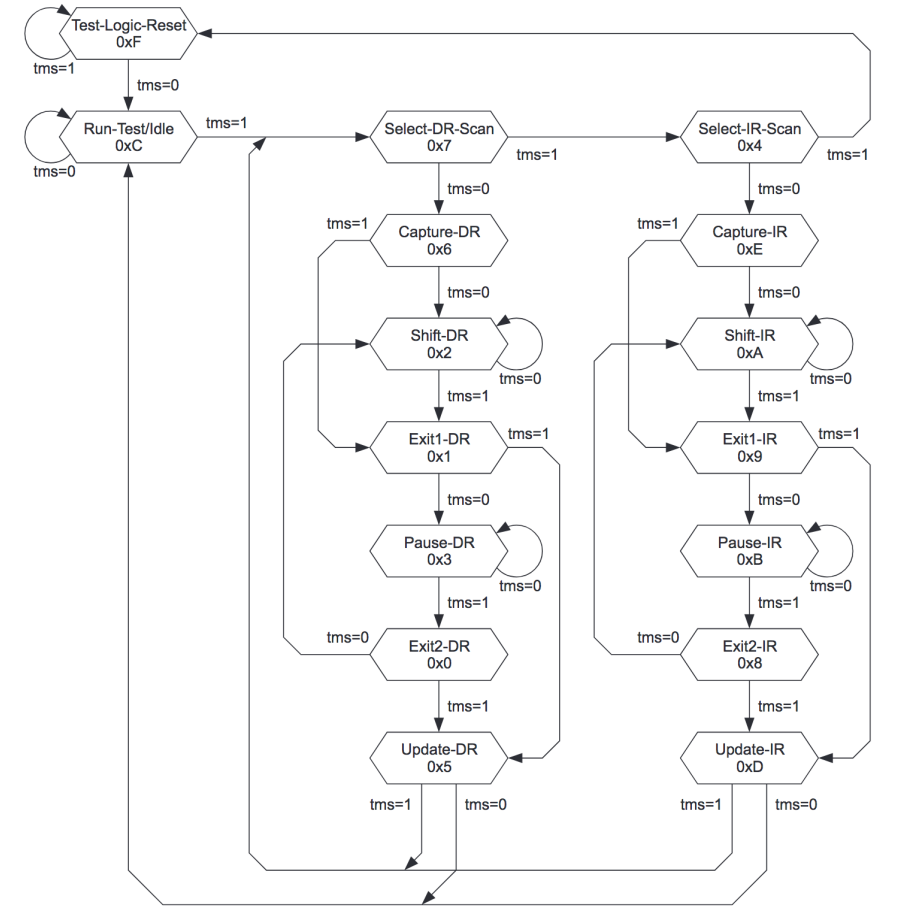


Figure 2.3: The JTAG state machine [35]

data pin, providing all the normal JTAG debug and test functionality plus real-time access to system memory without halting the processor or requiring any target resident code."[2] In this thesis, we were able to connect using SWD but were mostly using JTAG.

Determining if there is a JTAG interface and what is the pinout can be harder than determining the UART pins if the former is not labelled on the board. The detailed methodology we followed to achieve this is described in Chapter 4.

[2]As described by ARM on https://developer.arm.com/architectures/cpu-architecture/debug-visibility-and-trace/coresight-architecture/serial-wire-debug

2.3.3 Internet access for embedded devices

When one wants to connect an embedded device to the internet, many solutions exist: Dial-up, the Digital Subscriber Line (DSL), Wi-Fi or the Mobile broadband to quote only but a few. How one can get access to the internet over Dial-up and over GSM is further explained in the following subsections. First and foremost, we need to introduce the concept of DTE and DCE.

A Data Terminal Equipment (DTE) is a piece of equipment that lays at the end of the data link and handles the user information. A computer can be a DTE for example. A Data Communication Equipment (DCE) is a device connecting a DTE to the actual circuit like a telephone line. The way those two devices are interacting is described below and represented in Figure 2.4.

The RS-232 standard

The Recommended Standard 232 (RS-232) is a communication standard used to connect a computer to another device, usually a modem. It has been designed in the sixties to standardized data exchange for modems and electromechanical typewriters [3]. This standard does not define any encoding, data framing nor error detection and that is why it is combined with UART communications. Figure 2.4 shows the way a DTE and a DCE are inter-connected.

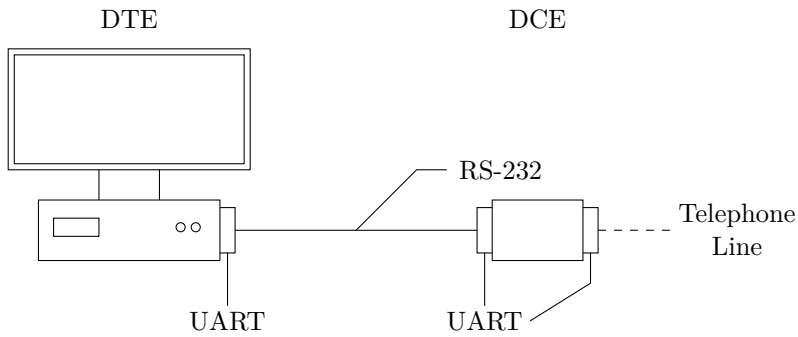


Figure 2.4: Connection between a DTE and a DCE

Dial-up Internet Access

When one has a device that needs an internet connection, Dial-up used to be a convenient solution. It is a way to transfer data using the Public Switched Telephone Network (PSTN). The data line is provided by a modem who calls the Internet Service Provider (ISP) and is in charge of establishing the connection. However, even though the device is able to exchange data at this point, it is not able to interact with the internet (that means to exchange IP packets) because there is no data link

layer and IP packets cannot be sent directly over the telephone line. A protocol called the Point-to-Point Protocol (PPP) can be used to solve that problem. That way, the device is provided with an IP by the ISP (usually inside a Virtual Private Network (VPN)). It can then interact with other devices in the same network or on the internet if available.

PPP is a data link layer protocol that has been conceived to be implemented over "links [that] provide full-duplex simultaneous bi-directional operation and are assumed to deliver packets in order" [2]. The telephone line meets those requirements. PPP can handle multiple higher protocols, and notably Internet Protocol (IP). According to the Request for Comments (RFC), PPP has three main components [2]:

- A method for encapsulating multi-protocol datagrams.
- A Link Control Protocol (LCP) for establishing, configuring, and testing the data-link connection.
- A family of Network Control Protocols (NCPs) for establishing and configuring different network-layer protocols.

Knowing the implementation details of PPP is not necessary to understand our work, but we truly recommend to the interested readers to have a look at the excellent books from Carlson and Sun on the subject [14, 52].

In the context of this thesis, we have a closer look at the services that are offered by PPP to the upper layers. More specifically, we are interested in the way a connection is established using a modem.

Not all the modems work the same way and support the same functionalities. Establishing a connection between two DTE still requires some steps that one can find in every connection processes. If for example, a device A wants to exchange data with a device B, it will first set up its modem. Once configured, it will ask for a connection by providing the modem with the number of the device to call. The called device's modem will be notified by a ring and notify device B of the call. That last one can accept or reject the call. If accepted, the modems are now connected and can start exchanging data. One of the devices can decide to close the connection by sending special characters on the line. Figure 2.5 shows the different steps.

Once a layer 2 connection is established, data can be exchanged between two DTE. It is then possible to have an IP/TCP stack. It is worth mentioning that the TCP/IP stack can be either on the DTE or on the modem depending on the configuration and of the available feature on that latter one. Recent modems are

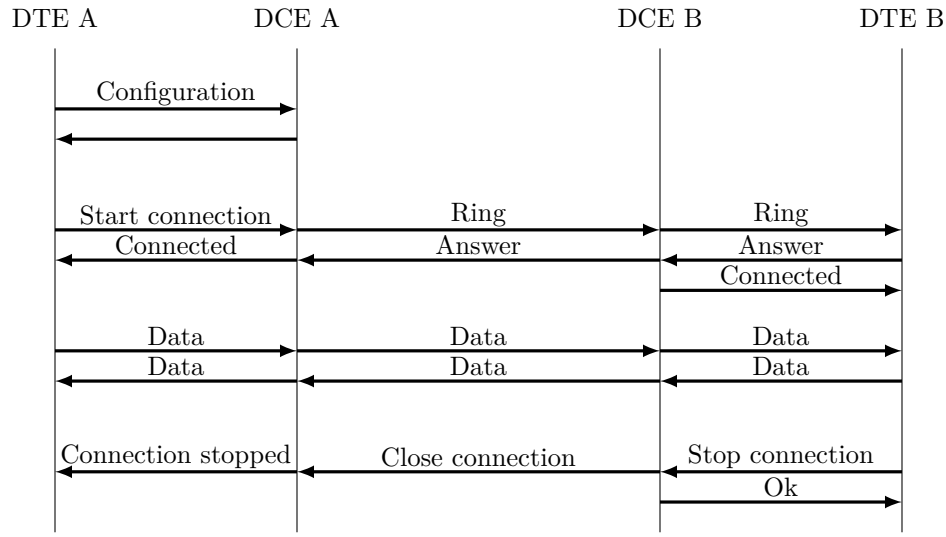


Figure 2.5: Data exchange's sequence diagram between two DTE

also able to handle higher layer protocols such as the Simple Mail Transfer Protocol (SMTP) or the File Transfer Protocol (FTP).

Internet access over GSM

GSM was designed with voice calls in mind. It is a circuit-switched network, which means that resources are reserved at the beginning of a session and released at the end. The circuit for a given session is then fixed and all bits of information will follow it. That has several advantages, such as having a constant bandwidth and delay time which suits well with voice calls [47]. However, it does not fit as good with applications that have various bandwidth usage (browsing the web is an example).

A solution to that problem was to introduce a packet-switched network instead of the circuit-switched one. General Packet Radio Service (GPRS) was introduced for that purpose. Sauter [47] explained that GPRS was "designed as a packet-switched addition to the circuit-switched GSM network". With that addition come multiple advantages such as faster access to the internet, but also the ability for the consumers to be charged for a volume of data and not for a time. Even though it is possible to send IP packets over a "regular" GSM network, it takes more time due to the fixed circuit that is used before reaching a packet-switched network.

When accessing the network, the device is provided with an IP address inside the ISP's network that allows it to communicate with other devices in that network and also to access the internet.

Hayes command set

Hayes command set or "AT commands" as they will be referred to in this thesis, are a set of commands developed in the eighties to interact with modems. They are used to configure and give instructions to such a device, for example, the command "AT+CPIN=4321" is used to give the PIN code to the modem so it can unlock and use the SIM card. Figure 2.6 presents the usage of these commands to establish a connection between two DTE. The "ATA" command is issued by the DTE B to answer the call and then the "ATH" command allows it to hang up. A modem can switch between two modes which are the *command mode* and *data mode*. Once a modem has reported "CONNECT" it means it is in data mode and is waiting for data to be sent or received. An escape sequence (usually "+++") is required to switch back to the command mode.

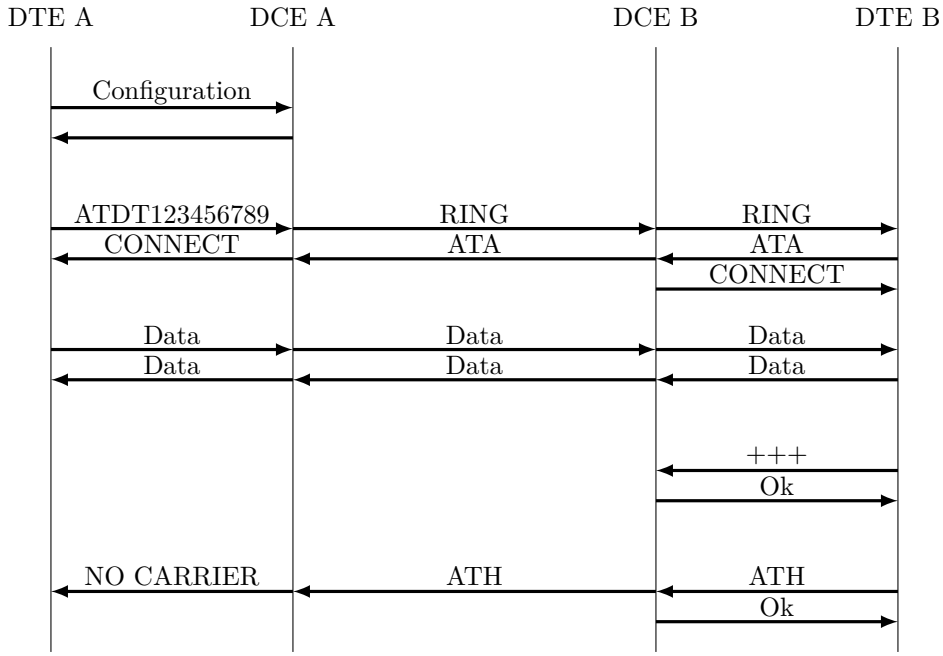


Figure 2.6: AT commands issued to establish a connection between two DTE

On a Linux system, the `pppd` daemon which is interacting with the PPP kernel driver to establish a connection with another peer and is also in charge of the negotiation of the IP addresses.[3] The `pppd` daemon is using chat-scripts to interact with the modem and establish the connection.

[3]According to the man page of the pppd daemon: https://linux.die.net/man/8/pppd

Listing 2.1: PPPD chat script example

```
"" "at"
OK "at&d0&c1"
OK "atdt2468135"
"name:" "^Umyuserid"
"word:" "\qmypassword"
"ispts" "\q^Uppp"
```

Listing 2.1, extracted from the `pppd` man page, presents part of a typical chat-script to establish a connection to an ISP. A script is usually composed of two text columns. The right one corresponds to the commands to be sent in response to the ones on the left. For instance, by issuing the "at" command, an "OK" is expected from the modem and one can then issue "at&d0&c1" and expect a second "OK". This goes until the connection is established and one of the peers hangs up.

Chapter 3
Methodology

The main motivation of this project is to assess the security level of the vendor's HMUs and to give general guidelines to biomedical technology companies to develop more secure implants and mitigate problems that might exist in the current models. With that goal in mind, we have chosen to follow two main methodologies to guide our work. Our research is driven by design science [58]. In addition, the practical part of the thesis, which is assessing the security of the devices, is following a Black Box Testing methodology.

3.1 Design science

Design science is defined as "*the design and investigation of artefacts in context*" [58]. Wieringa explains that an artefacts is "*something created by people for some practical purpose [...] used when designing, developing, implementing, maintaining, and using information systems and software systems.*" This can be software, hardware, methods, etc. He also highlights that those artefacts are designed to be used *by people* which means in a given context. A context is then defined as anything that interacts with the studied artefacts or which has an influence on it. It could be a person, other artefacts, a budget, etc.

Our study takes place in the context of the pacemaker ecosystem. Multiple artefacts can be defined here, such as the pacemaker itself, the HMU, the programmer, the protocols or the method(s) used to transfer data between the pacemaker and the HMU. Our project can be defined as the validation of a human created artefacts (the HMU) in a given context (the pacemaker ecosystem).

Special care needs to be taken when assessing medical devices. It is indeed particularly true that a vulnerability that could seem harmless in a regular IoT device can have a huge impact on the patient's safety or privacy. As explained more in details in Section 3.4, we have kept in mind the context of our artefacts when evaluating its security. Considering the context is also really important as it allows

to take new attack vectors into account and sometimes discover large scale attacks. A good example of such a flaw is the study on femtocells' security, where researchers were able to interact with other devices once inside the provider's network [12].

Design science is not only about evaluating artefacts, but also about building some new artefacts to make improvement to the area. That is why we aim at assessing the security of the vendor's devices, but also at giving some guidelines on the conception of a medical device and more generally of IoT devices. As defined by Wieringa, those results are also artefacts and their conception is following the design science. They are indeed based on the knowledge acquired on the system but also on external knowledge. Hypotheses that have been tested, along with the methodology used to assert or dismiss those hypotheses, are described and can constitute a base for a testing guide on medical devices. We hope that these guides will help other students and researchers in assessing embedded devices.

Even though design science helps us keep our initial goal in mind as well as present our research, it is not the best methodology to assess the security of the HMU under study. Indeed, it gives us general guidelines on how to drive our research but no information on how to perform the security assessment in itself. That is why our methodology is mainly based on another one that fits better: the Black Box Testing Methodology, which is explained more in depth in the next section.

3.2 Black Box Testing

3.2.1 Definition

The Black Box Testing methodology is a way to assess a software, a device or more generally a system from the outside while having very little knowledge about its internals. The attacker is analyzing the outputs of the box obtained by sending some inputs or just by passively listening, and then tries to deduce the internals of the target. Having made some guesses, the attacker can adjust her inputs to confirm her thoughts or to exploit the target (see Figure 3.1).

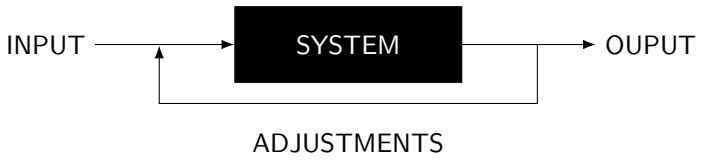


Figure 3.1: Diagram of the Black Box Testing Methodology

This methodology has several advantages compared to others that can be used to assess a system. Indeed, the goal here is to test under real conditions, to fit a real scenario. That means that such a test will include real errors that can be made

during the deployment of a system, such as default passwords, misconfigurations in general or even a lack of training of the employees (weak password for instance). It also has a low false-positive ratio as the attackers can assess the risks associated with a vulnerability, i.e. if a given vulnerability is really exploitable or not. A White Box Testing on the other hand, where the attacker has full access to source code and implementation details, might lead to vulnerabilities that are not relevant (not given the goal of a Black Box Security testing at least) because they would never be triggered in a real-world scenario. However, this methodology also has some drawbacks. By definition, the attacker has very little information about the target and might miss some vulnerabilities that would have been detected by code and/or configuration review. It is then clear that a Black Box Testing should not be the only security test performed, even though it is an excellent way to assess a device under real conditions.

Our project focuses on a proprietary ecosystem, on which we have almost no information apart from the one the vendor is providing the patient with, which is in fact really vague and non-technical. Moreover, our goal is to assess the security risks related to that ecosystem in a real-world scenario. Black Box Testing Methodology then fits well with our constraints and objectives and that is why we have decided to use it.

3.2.2 Steps of the methodology

Our process can be split into five different tasks (see Figure 3.2). The very first one is the *Hardware Analysis*. Once a device is acquired, one starts analyzing its components, which means knowing what the exposed interfaces are, debug interfaces, but also the chips that are on the board. In order to know that, opening the device to access the PCB and analyze it is often required.

Knowing the components and available interfaces, one can then start looking for *documentation* such as datasheets, RFC or any other relevant information about the device. The goal of that second step is to understand the overall system and come up with some first hypotheses about it.

From those hypotheses, we then come with *testing* scenarios to be performed on the device and that will have two possible outcomes: either a success or a failure. However, the success or the failure of a testing scenario is determined by the expected result, which means in reality that even failure brings us information about the system.

Indeed, the results of a specific scenario need to be interpreted. This interpretation will be called a *finding*. Those findings are then used to look for new documentation and/or infer new hypotheses about the device. For example, if the hypothesis is “the

device has debug ports exposed on the UART pins and is providing the attacker with a shell when connected to it", then the testing of that particular scenario will lead to either a confirmation or a rejection of the hypothesis. The interpretation is here quite easy as it is the hypothesis itself. This finding can then be reinjected in the documentation phase to infer new testing hypotheses like "the attacker is given a root shell when connecting to the UART console" or "the attacker can access the filesystem when connecting to the UART console".

Those findings are finally gathered to be *reported*. The reporting step is the one where the device is considered back in its whole context, as explained in Section 3.1. That means the findings are interpreted again, but that time in regards to different metrics. In the case of the above example, one can wonder what is the impact of the attacker having access to the filesystem on a medical device? Linked to other findings such as "The users' data are stored in cleartext on the device" or "The users' data are not stored on the device" the interpretation can be very different.

The mitigation is also included in the reporting step. The real value added to a product when performing a security test is this mitigation part as it explains to the vendors how to improve the security of their products.

Those steps can be mapped with the Open Source Security Testing Methodology Manual (OSSTMM) which is widely used to assess ITs' system's security [25]. Indeed, the first two steps (*hardware analysis* and *documentation*) correspond to the *information gathering* (or approach) phase in the OSSTMM. The *contact* phase is then used followed by the *exploitation* phase, which are here mapped with the testing and findings phases. In the OSSTMM, the information gathered during the first phase along with the one gathered directly by the *contact* phase is then use to exploit the system and gain access. In our process, the information required to exploit, and gain access comes from previous testings. Finally comes the *reporting phase*, including mitigation. In the OSSTMM, one more phase is sometimes used depending on the engagement: the persistence one. In our case, persistence is studied as a hypothesis which is then tested and reported as any other findings.

3.2.3 Use cases

When it comes to the choice of a way to assess the security of a system, there is no better choice between a Black Box Testing and a White one. The decision highly depends on the company's objectives. Indeed, they will prefer a White Box testing if the goal is to analyze the code in detail and maximize the amount of time spent on the penetration test. If they aim at identifying attack vectors and targets that are more likely to be used by attackers in a real-world scenario, then they might prefer a Black Box testing.

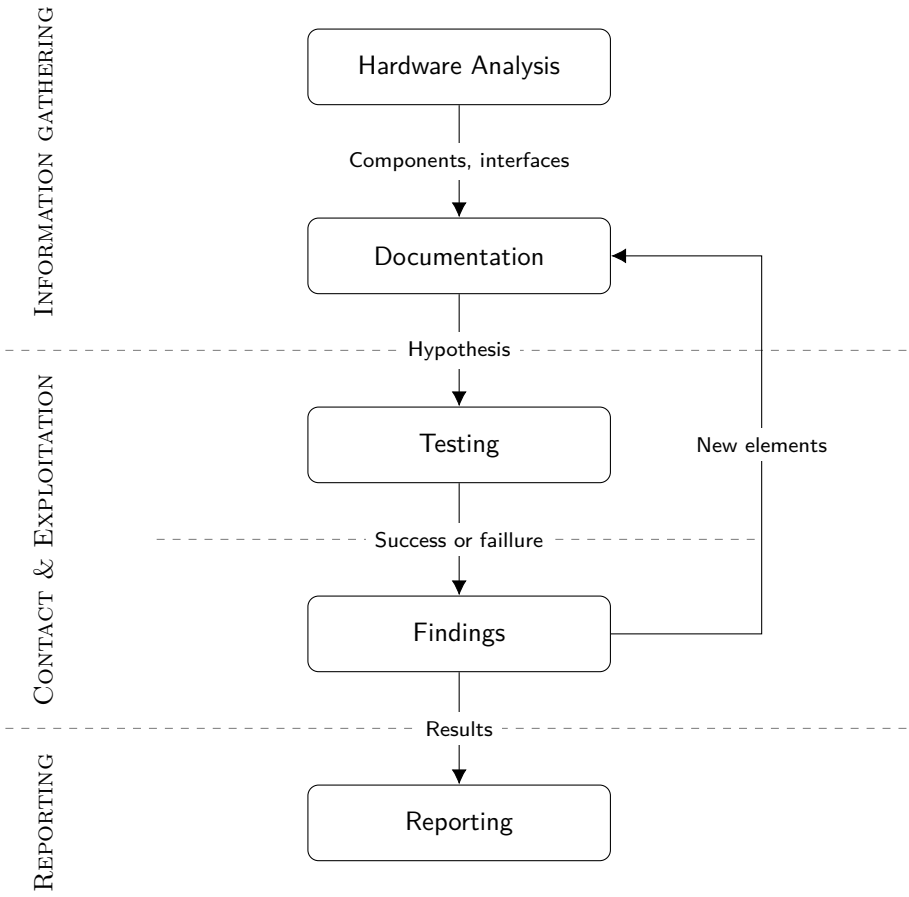


Figure 3.2: Black Box Methodology iterative cycles used in our project

Black Box Testing is for sure a wise choice for security managers who want to assess the security of their IT systems, as it will help to perform a risk evaluation given different scenarios and threats (external hacker to the company, malicious employee, etc.). With that information, it is then easier to harden the weakest points in the system. We can then advise to start first with Black Box Testing that will help to identify urgent problems and once these problems are solved, go deeper into the security with White Box testing. Even though we are here describing a situation where a product is assessed once already produced, it is clear that the best practice is to consider security during the design process.

As explained in Subsection 3.2.1, Black Box Testing is an obvious choice for our project given our scope. It allows us to perform a realistic analysis of the system and to evaluate its security from the point of view of an external attacker. We also

performed a risk assessment of the findings we got, based on patients' safety and privacy. The threat model we are using and the way we are assessing the findings are described in the next section.

3.3 Threat model

3.3.1 Definition

Threat modelling is a methodology on its own that allows a company to consider its critical assets along with the potential threats and attack vectors that could be used. It is defined by Shostack as "the use of abstractions to aid in thinking about risks." [48] Even though we are not using the threat modeling methodology in this thesis, we still define a threat model to help us assess our findings.

3.3.2 Threat model considered

In the pacemaker ecosystem, and more generally in the medical devices industry, two elements are vital: patients' safety and privacy. In the case of the pacemaker ecosystem, both elements are at stake and can be targeted by an attacker, or a group of attackers in order to harm or kill someone or to generate money out of it. Selling personal data can be very prolific and is a real target for attackers as demonstrated by the breach in the Norwegian Healthcare systems in January 2018 [50]. As already mentioned in our motivations to work on that topic (see Section 1.2), attackers can also target a high-valuable person with the intent to harm or kill them.

Many people are interacting with the pacemaker's ecosystem and can thereby be considered as potential attackers. In this thesis we are studying the impact of an active attacker, that means, an attacker that will directly interact with the HMU if possible. Interactions can be passively eavesdropping, modifying, replaying or blocking messages from or to the HMU on its wireless interfaces, but also physical attacks if the attacker has the possibility to get access to an HMU. Table 3.1 details the potential attackers and their capabilities. If not specified, one can assume that we are considering an attacker which is an outsider to the ecosystem who has acquired an old HMU, on eBay or other resellers of electronic equipment.

When considering the pacemaker's ecosystem, not all parts have the same probability of being targeted. First of all, they are not exposed and protected the same way: the programmer is in a hospital and the mobile network is not accessible by everyone. Secondly, not all devices carry valuable information or allow an attacker to perform a specific attack that could harm the patient. Finally, the skills required to attack a pacemaker are not the same than the skills required to attack a website. From those assumptions, we can tell that the link between the pacemaker and the

Potential attacker	Role in the ecosystem	Capabilities
Outsider	None	Can acquire an old HMU
Medical Staff	Programming the pacemaker	Access to the programmer
Patient's doctor	Monitoring	Access to the online console
Patient's entourage	None	Access to a working HMU
Patient	Patient	Access to a working HMU

Table 3.1: Potential attackers and their capabilities

server is probably the more likely to be attacked. It is indeed quite easy to get access to an old HMU, using eBay for instance.

Moreover, the HMU is directly connected to the vendor's backend servers, through which all the data are gathered. Finding a vulnerability on those could lead to a massive data breach and be of interest for attackers. It is therefore relevant to consider the HMU as a target and to study its security.

Defining this threat model helps us evaluate the probability of an attack and thus the overall risk.

3.4 Assessment

3.4.1 Assessment criteria

In order to evaluate the impact of a finding on our assets and more importantly on the patient's safety and privacy, assessment criteria are required. For that, we are using the *Confidentiality, Integrity and Availability (CIA)* triad, extended by the *Non-repudiation* and the *Authentication* (further referenced as CIANA in this thesis). The impact of a given finding or set of findings will be assessed regarding these five keys.

Confidentiality

In its "Glossary of key information security terms", Kissel defines confidentiality as "*the property that information is not disclosed to system entities (users, processes, devices) unless they have been authorized to access the information*." [30] Personal data is very valuable on the black market, and medical information systems are a privileged target to obtain it. The Norwegian HealthCare Data breach that occurred in January 2018 is a good example [50].

Integrity

The integrity is defined as "*the property whereby an entity has not been modified in an unauthorized manner.*" [30] Successful attacks on the pacemaker impacting the integrity could lead to the death of the patient. For instance, if an attacker is able to reprogram the pacemaker and to change its therapy, or even worse if he is able to deliver a shock to the patient.

Availability

According to Kissel, the availability is "*the property of being accessible and useable upon demand by an authorized entity.*" [30] This property is one of the most important when it comes to the pacemaker. Indeed, if the pacemaker has no more battery or if it is not functional anymore because of an attack, the consequences can be disastrous, and the life of the patient might be endangered.

Non-repudiation

Non-repudiation is defined as a "*protection against an individual falsely denying having performed a particular action. [It] provides the capability to determine whether a given individual took a particular action such as creating information, sending a message, approving information, and receiving a message.*" [30] In the context of the pacemaker ecosystem, each entity interacting with the pacemaker should be clearly identified, and its actions should be logged in such a way that makes it impossible for it to repudiate its acts. For instance, a practitioner that configures a treatment on the pacemaker should not be able to deny he did.

Authentication

Authentication is "*the process of verifying the identity or other attributes claimed by or assumed of an entity (user, process, or device), or to verify the source and integrity of data.*" [30] When it is applied to medical devices, it means that entities should be able to verify the source of the data that is exchanged. The pacemaker should be able to check that the incoming data ordering a new configuration is indeed coming from a trusted programmer. Similarly, the vendor's servers should authenticate the data, and confirm their origin before processing them. An attacker could indeed forge data's packets and send them to the server to corrupt the real data and trick the practitioner into thinking that something is wrong.

Two levels of interpretation exist when it comes to the risk assessment of our findings: the device level and the ecosystem level. The findings will first be evaluated given their impact on the Confidentiality, Integrity, Availability, Non-repudiation and Authentication (CIANA) criterion (yes or no), then on the patient's safety and

privacy (yes or no) and finally, we will give our evaluation of the finding's criticality with the levels described in Table 3.2.

Criticality level	Description
None	No impact on the system
Low	Cannot be used on itself to compromise the system or gather data from the system but can be linked to other findings to do so
Medium	Gives the attacker the ability to extract sensitive information out of the system
High	Gives the attacker the ability to control the system

Table 3.2: Findings' criticality levels

As explained in Section 3.3, one needs a context to really make sense of our findings and of their evaluation. That is why our evaluation is based on a specific scenario: we assume that our attacker is an outsider attacker (see Table 3.1) who acquired an old HMU on the internet. The goal of the attacker is to threaten her target's privacy and safety. We also assume that the attacker has the ability to get physical access to the patient's HMU for a few hours (by breaking into the patient's home for instance). Motivations and costs other than the costs of the tools required to perform such an operation are not taken into account in this thesis. They are however discussed in Chapter 6.

3.5 Ethical considerations

Ethics and information security are closely linked together, and security researchers can be faced with a dilemma like whether or not they should publicly disclose their findings. This is particularly true in areas like medical devices, automotive and other critical industries. Indeed, working on the security of a product can have several ethical concerns.

Researchers can find personal and sensitive data during their research and need to deal with them carefully. As described in Section 1.1.3, multiple devices are available in the lab for testing. Those devices have been acquired through the internet or donated to the project. Most of them are not new, such as the pacemaker itself and contain patient data. These data consist of the patient's identity, age, condition, etc. which must not be disclosed and have been systematically redacted from this report and from any document that has been published or read by people not involved in the project.

Vulnerabilities discovered in the pacemaker ecosystem during the security assessment will be reported to the vendor in a coordinated vulnerability disclosure process. The parallel with the automotive industry is here quite obvious. Fixing vulnerabilities in such areas is not as straight forward as deploying an update for Adobe Acrobat for instance and can require the devices to be recalled and/or certified again which can take several months. Those problems will be discussed more in depth in Section 6.2 at the end of this thesis.

To deal with those issues, the Norwegian Centre for Research Data (NSD) has been notified at the beginning of our project. In order to disclose our findings in an ethical and responsible way, we are in contact with the German Federal Office for Information Security (BSI), which has already helped in the coordinated vulnerability disclosure process for the last year project [33].

Chapter 4
HMU Security Analysis

This chapter presents the security analysis of several Home Monitoring Unit. In Section 4.1 we introduce the preliminary testings performed on HMU that are known to be vulnerable. That step helped us becoming familiar with the problems that can exist in such devices along with the testing methodology. In Section 4.2 and 4.3 we present our main practical results for this thesis. These results focus on both the T-Line and GSM versions of the CardioMessenger II-S. The main findings of this analysis are summarized in Section 4.4. Finally, section 4.5 presents the analysis that has been performed during the thesis on other devices of the Biotronik's ecosystem, but on which no risk assessments have been performed. The main reason for this analysis was notably to provide useful information to the other ongoing project (see the MSc Thesis of Lie at NTNU [34]).

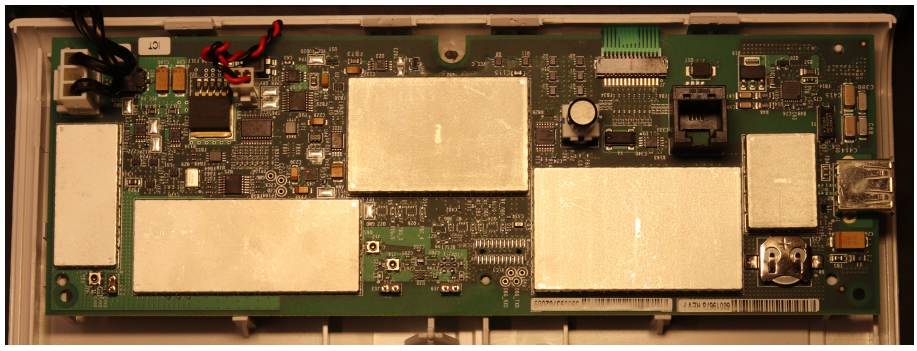
4.1 Preliminary HMU Security Analysis

When designing a testing methodology, a tool or simply a test to verify an assumption, it is good practice to check that everything is working as expected on known cases. Indeed, knowing both the input and the expected output helps troubleshooting. Starting our research by testing two devices from other vendors, that are known to be vulnerable, was then a way to get to know the tools at our disposal in the laboratory and also to confirm some of the findings found by other researchers on these devices. Having an idea of the level of security that can be implemented in medical devices that have been produced at the same time than the CardioMessenger II-S was also a motivation to study other devices.

4.1.1 Analysis of the Merlin@Home HMU

The Merlin@Home, from St Jude's Medical (see Figure 4.1), is the first board that was tested. It is a device from 2008, like the CardioMessenger II-S, and is known to be vulnerable [10]. The goal of the first testing was attempting to reproduce the results of [10].

(a) Outside of the Merlin@Home



(b) Inside of the Merlin@Home

Figure 4.1: Outside and inside of the Merlin@Home

Finding and connecting to the debug ports

The first testing point was the debug ports. As one can see on Figure 4.2, UART pins are labelled on the PCB. The pins that can be observed on the left of the picture correspond probably to the JTAG interface. However, this device has not be tested against that hypothesis.

Using an oscilloscope and monitoring what is happening on TXD during the boot process confirms that the device is sending information on the UART. In order to determine the Baud Rate and avoid guessing, we acquired the observed signal using a Logic Analyzer[1] and calculated an approximation of the Baud rate as explained more in details in Section 2.3.1.

[1] A Logic Analyzer is an instrument that can capture several digital signals simultaneously and convert them into a timing diagram that can then be interpreted given a specific protocol.

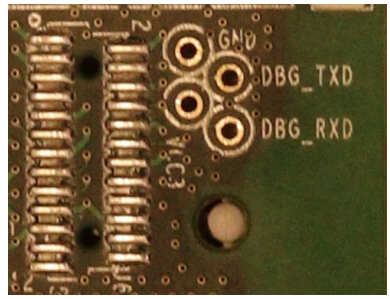


Figure 4.2: UART pins on the Merlin@Home

$$B = \frac{1}{8.625 \times 10^{-6}} = 115\,942 \approx 115\,200\ Bd.s^{-1}$$

After having connected a computer to the device using the Shikra[2] along with `screen` utility, one can hit the reset button and the boot process will start. This is partially presented in Listing 4.1.

Listing 4.1: Partial booting information from the Merlin@Home

```
Post device verification...
Serial2In string: ATi0
Serial2In string:
56000
Modem Post : Passed with retries = 0

Time taken by POST : [1.197000] seconds
nand_init: manuf=0x000000EC  device=0x000000F1
scanning for bad blocks...
nand_check_blocks: nand_read_page() failed, addr=0x04940000
```

Once finished, the user is prompted for login. Unfortunately, we did not guess the password (assuming that the login is “root”).

During the boot process, one can press any key to stop it. We proceeded and gained access to a terminal in the Blob bootloader. Some commands are then available, such as the *flash* commands. The *boot* command is particularly interesting as it allows to boot with the kernel options provided as parameters. One can use the *init* option to configure a shell as the init program. However, when issuing the command `blob> boot init=/bin/sh`, the bootloader crashed, complaining not to find the root path. This information is available in the normal boot process, as highlighted in Listing 4.2.

[2]The Shikra is described in Appendix A.2.1.

Listing 4.2: Default Merlin@Home's kernel options on boot

```
Kernel command line: console=ttyMX0,115200n8 root=/dev/mtdblock6
     ip=dhcp BOARD_REVISION=
```

Figure 4.3 presents the root shell that is obtained by issuing the `boot` command with the correct kernel options.

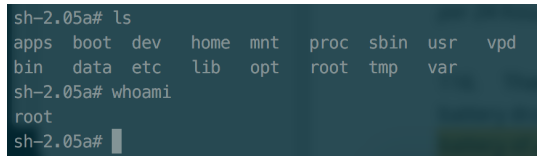


Figure 4.3: Root access on the Merlin@Home

As a consequence of having a root shell, one has full control over the device. The environment seems to be BusyBox.[3] The `/etc/passwd` file can be gathered and the root password cracked in less than three minutes using the password cracker `John The Ripper`[4] on a Virtual Machine. It is a seven characters password with the regex `[a-zA-Z]{3}[0-9]{4}`. Connecting again to the device after the boot using the root password did not work even with the correct pair of username/password. Our guess is that the root user is not allowed to connect directly, which is a good security measure. To enable direct connection, one could set a password to one of the other accounts that exists on the board (such as the "operator" account) and try to connect with it. However, this could brick the system if any checksum is verified during the boot process. Several other information can be gathered:

- Known hosts
- SSH keys
- Development shell scripts

It is worth noticing that a script called `hotplug.sh` is in charge of executing scripts that are in a specific directory each time a device is plugged in the HMU. For instance, it will run all scripts placed in `/etc/hotplug.d/usb/` whenever a device is plugged in the USB port. That can probably be used as a "backdoor". This device can also be connected to the internet and accessed via SSH. But no modifications have been done to verify these assumptions.

[3] BusyBox is a single binary containing several Unix utilities (like cat or vi) and is optimized to be small. That is why it fits well the constraints of embedded devices and is usually found on routers.

[4] Available at https://www.openwall.com/john/

No patient's data have been found on that device, probably because it was not associated to any pacemaker device. However, given the fact that getting a root shell is possible, we assume that all data going through the HMU can be intercepted.

4.1.2 Analysis of the Medtronic HMU

Having some experience with the Merlin@Home, we decided to investigate the Medtronic device using the same attack vector: the debug ports. Indeed, the pins on the Medtronic's HMU are not labelled, which allowed us to gain experience in determining the UART pins. Once the box is opened, one can notice three (four) different pin's headers (soldered by the previous students), as shown in Figure 4.4.

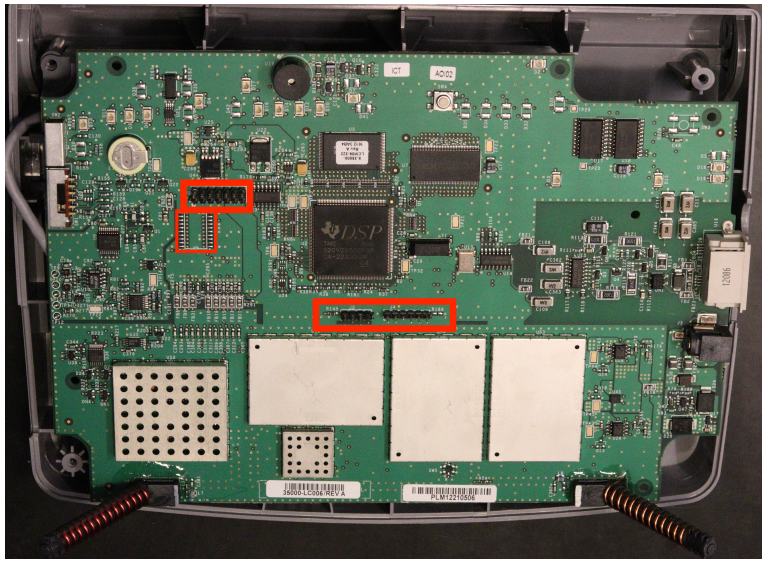


Figure 4.4: Possible debug pins on the Medtronic board

Following the methodology detailed in Subsection 4.2.2, one can determine the UART pinout on the 4-pins header of the Medtronic's HMU. The result is presented Figure 4.4.

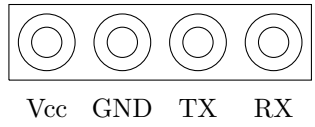


Figure 4.5: Mapping of the Medtronic UART Pins

When connecting a USB-to-TTL adaptor to the UART pins, one can see the boot process of the device. However, no interaction is possible. No further trials have been performed to interact with the UART.

Determining the JTAG pinout on the Medtronic's device has also been tried. However, it gave no results. Since our objective was to study the Biotronik devices, we did not pursue this work any further.

4.2 Security Analysis of the CardioMessenger II-S TLine

While all Biotronik's devices at our disposal have been tested against certain hypotheses, we have decided to focus on the two versions of the CardioMessenger II-S: GSM and T-Line. Although it was designed in 2008, it is still widely in use today. Thus, those devices could be a good target for malicious adversaries. Some testings performed on the other devices of the ecosystem have also been used to deduce information on the CardioMessenger II-S.

In order to ease the mitigation but also the readability, we are presenting our results in a way that is mapped onto our methodology. We first present the hardware analysis performed on the device, followed by the different hypotheses the HMU has been tested against, split into three categories:

- hypotheses related to the debugging interfaces
- hypotheses related to the hardware components and to the network infrastructure
- hypotheses related to the memory analysis and the reverse engineering of the firmware

For each of these sections, we give the general hypothesis $\mathcal{H}_x$ the device has been tested against along with the motivations to do so. We then describe sub-hypothesis $\mathcal{H}_{x.y}$ and the testing process used to verify/dismiss each of hypothesis. Our main findings are highlighted at the end of each hypothesis $\mathcal{H}_x$ and a summary of the results is available in Section 4.4.

4.2.1 Hardware analysis

The CardioMessenger T-Line is easy to open. It only takes a few minutes and no particular manipulation is required. Once opened, one can notice that the hardware components are all visible and that no obfuscation is used to make the identification of the electronic components harder (see Figure 4.6).

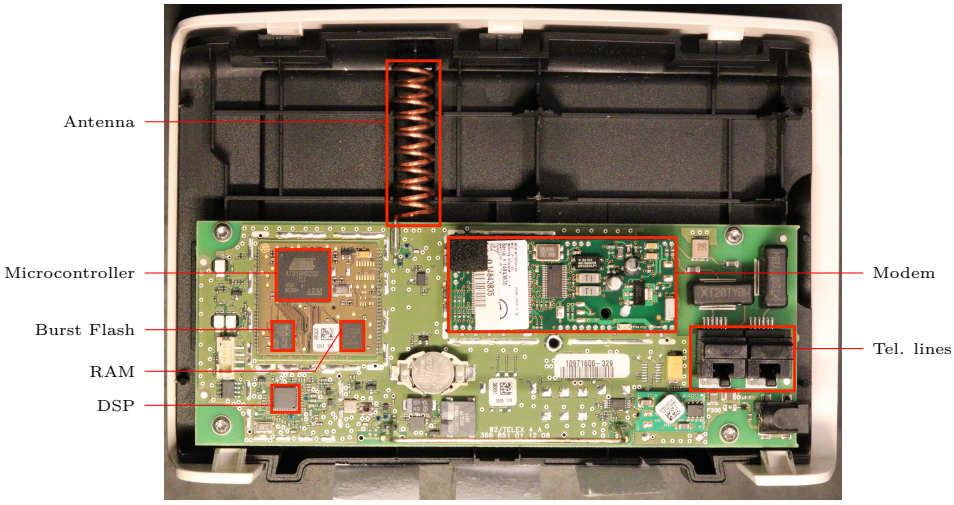


Figure 4.6: Inside of the CardioMessenger II-S T-Line

Multiple elements can be identified and searched on the internet for datasheets:

- The microcontroller which is based on an ARM Thumb Processor and featured with debug and UART interfaces
- A Burst Flash memory with a capacity of 4Mb
- An external Random Access Memory (RAM)
- A Digital Signal Processor (DSP)
- A modem that can provide an internet access
- An antenna, most likely to be used to communicate with the pacemaker
- Telephone line inputs

One can also notice that there are two PCB. A smaller one, containing the microcontroller, is on top of the main one. It looks like a development board which could be documented on the internet. However, we were not able to find any information about it. That board has also several testing points that might correspond to the debug interfaces.

Given the components on the board, we tried to determine what kind of Operating System (OS) is running on the board. Indeed, the box previously tested (see Section 4.1) were running a Linux system. However, in the case of the CardioMessenger

II-S, it is not a Linux OS. Given the microcontroller chosen, it is hard to believe that the developers of this system have developed their own firmware from scratch as it would represent a huge amount of work. That is why we made the hypothesis that some OS was running on the board. However, it was not easy to say if it was a Linux-based system or not given the fact that the microcontroller could run a Linux system. We finally made the hypothesis that it was not such a Linux system because of the size of the flash (4MB) which had several constraints in terms of features that can be included. Also, that hypothesis is reinforced by similarities found with the vendor's other devices (see the LLTv2's analysis in Section 4.5.2) which are not able to run a Linux-based system.

As a result of the hardware analysis, we got several elements to start our research with. The hypotheses inferred from those elements are presented below. We can also define a first finding regarding the hardware's obfuscation level.

Finding $\mathcal{F}_0$: An attacker can easily open the device and there is no obfuscation of the electronic components that would harden the identification of components.

4.2.2 Debugging interfaces

As highlighted in the previous section, multiple testing points are visible on the PCB. UART was the attack vector used to compromise the Merlin@Home device and is one of the most common way to compromise an IoT device. That is why our first hypotheses are about debugging interfaces.

UART interface ($\mathcal{H}_1$)

$\mathcal{H}_{1.1}$ Identification of the UART pins
Multiple pins and testing points are exposed on the PCB, but there are no labels indicating to which interface they belong or what they are. As one can see in red in Figure 4.7, multiple points are gathered together on the small PCB and look promising. Trying to plug one's computer directly without knowing the pin mapping could damage the equipment. That is why we followed the methodology described below to determine if there is a UART interface available.

The methodology we used to determine the pin mapping without any label is the following:

1. Determining the ground pin using a multimeter. A metal part of the device is usually a good ground reference if no other. If the multimeter is featured with a continuity test functionality, then one can hear a "Beep" when two points are connected together.
2. Determining V_{cc} without the continuity test is not straight forward and might often be guessed. With a continuity test, when connecting the ground and

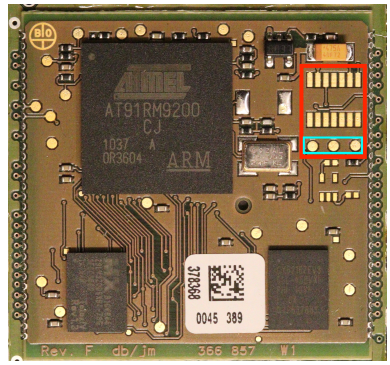


Figure 4.7: Possible debug ports on the board

V_{cc}, one can sometimes hear a short "Beep" due to the presence (often) of a capacitor.

3. Once the ground and power pins are known, one need to determine which pin is used for transmission and which one is used for reception. This could be guessed by simply connecting to it and switch if it is not working. However, it is also possible to monitor the activity on both pins using an oscilloscope and if data is sent on Tx, then it should be visible.
4. The last step is to determine the Baud rate in use. This is achieved using either the oscilloscope or a logic analyzer. The minimum symbol rate is measured and from that, one can deduce an approximation of the Baud rate using the relation given in Figure 2.2. Of course, it is also possible to guess with different Baud rates until the data seems to have a meaning.

As a result of the testing of all points on the PCB, we identified the UART interface. One can see the UART's pins framed in the cyan color on Figure 4.7, with the mapping detailed in Figure 4.8.

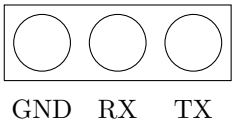


Figure 4.8: Mapping of the CardioMessenger II-S UART Pins

$\mathcal{H}_{1.2}$ Debug information sent on the UART interface

Having the UART pins mapping, the second hypothesis we made was that some debug information or boot information is sent on that interface like with the Merlin@Home or the Medtronic's HMU. To verify that assumption we connected a computer to the UART using a USB-to-TLL connector and monitored the activity during the

boot process. Some information is indeed sent by the device just after the boot. It displays the name of the bootloader (see Listing 4.3).

Listing 4.3: Output captured on the UART during the boot process

```
Bootloader TELEX4:
```

This is however the only information that is sent over the UART. Looking on the Internet for that specific string did not provide us with any interesting information either. Thanks to the firmware analysis performed in $\mathcal{H}_7$, we believe that the debug feature is disabled for commercial devices.

$\mathcal{H}_{1.3}$ Interaction is possible through the UART

The next logical step was to try to interact with the UART interface. Once the UART pins known, one only need to connect a UART-to-TLL adapter to it in order to interact. The `screen` utility can then be used on Linux and Mac OS. A software such as `Putty` can be used on Windows system.

Here are the `screen` commands to issue :

```
# Basic connection
$ screen /dev/tty.usbserial-XXXXX 115200

# Logging enabled (in screenlog.0)
$ screen /dev/tty.usbserial-XXXXX 115200

# Close an open session without using screen -ls manually
$ screen -X -S `screen -ls | grep "$(uname -n)" | awk '{print $1
    }'` quit
```

As we missed some functionalities with `screen` (such as advanced logging) and needed some tunable way to interact with the device, we have developed several scripts. Those scripts are written in Python 3 and make use of the pySerial[5] library. They are available in Appendix B.1.2.

Once connected to the CardioMessenger II-S, we tried to issue commands like "ENTER", "CTRL+C", "ESC", etc. but we do not manage to get any interaction with the device. The text sent is not echoed either.

Finding $\mathcal{F}_1$: An attacker can identify the UART interface, even though there are no labels on the PCB. It is enabled and the Bootloader's banner is visible during the boot process. We did however not succeed in interacting with this UART interface directly.

[5] Available at https://github.com/pyserial/pyserial

JTAG interface ($\mathcal{H}_2$)

Another common way to compromise an IoT device is to use the JTAG interface. In the case of the CardioMessenger II-S, it should have been the second hypothesis tested after the UART. Chronologically, however, this hypothesis was one of the last one tested because of its "complexity". Indeed, soldering a connector to the identified interesting pins was required to get the pins mapping. That soldering is not easy to do and could have broken the board if not done properly. Figure 4.9 shows in cyan the pins that are most likely to be the JTAG pins.

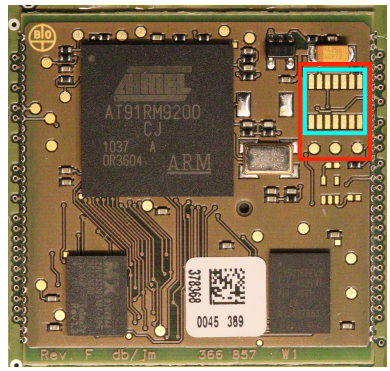
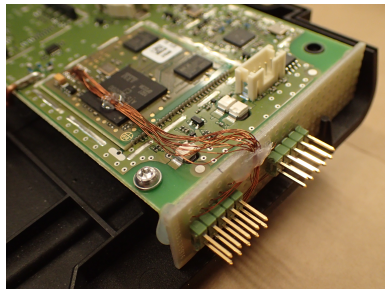
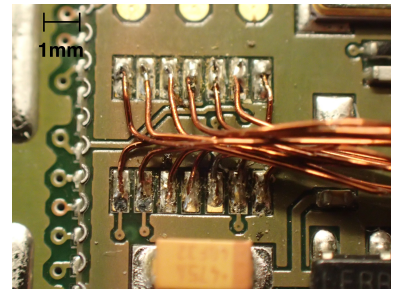


Figure 4.9: Possible JTAG pins on the board

Thanks to an engineer from the NTNU's Electronic Department, we were able to add a proper connector onto those pins. Figure 4.10 presents the connector that has been created along with the soldering performed on the board.



(a) JTAG connector added to the board



(b) Details of the soldering

Figure 4.10: Details of the JTAG connector added to the board

$\mathcal{H}_{2.1}$ Identification of the JTAG pins

Once the connector added to the board, we were able to first confirm the presence of JTAG and then to identify the pinout.

To determine the pinout of the JTAG interface, we used a special hardware tool called "JTAGulator". As explain by Grand Idea Studio, the distributer of the JTAGulator, it is an "open source hardware tool that assists in identifying OCD connections from test points, vias, or component pads on a target device." Basically, it can guess the pinout of the JTAG interface by trying all possible permutations. The JTAGulator[6] is an inexpensive commercial off-the-shelf equipment and is presented in Figure 4.11.

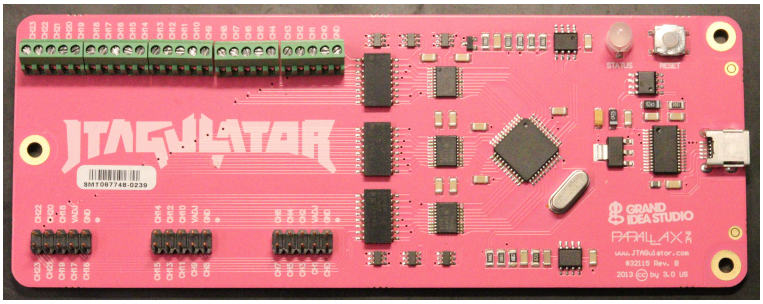


Figure 4.11: The JTAGulator from Grand Idea Studio

The device is easy to use. First, one needs to connect the interesting pins of the board to the JTAGulator (as presented in Figure 4.12). The JTAGulator is then connected to the computer with USB and one can interact with it through `screen`. The Baud rate is 115 200 Bd/s.

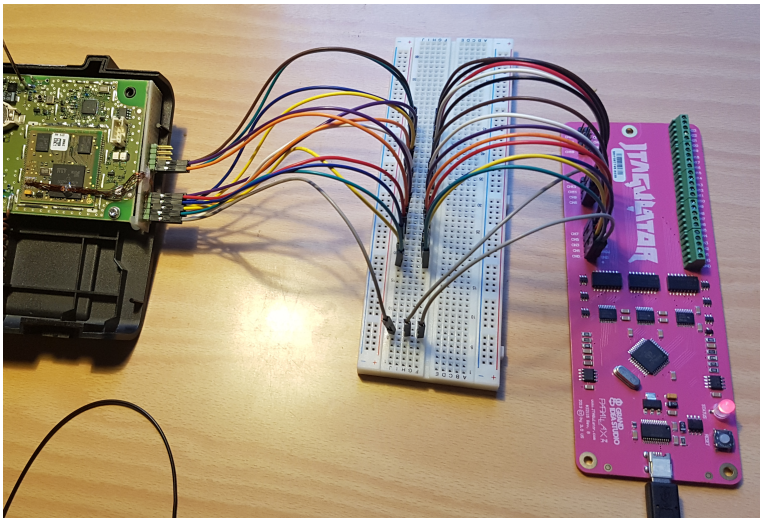


Figure 4.12: Connection of the JTAGulator to the device

[6]The version of the firmware of the JTAGulator we used was v1.6. The flashing process on Linux and Mac OS is presented in Appendix A.1.1

Once the device connected, the steps are the following:

1. Adjust the voltage of the JTAGulator (3.3V in the case of the CardioMessenger II-S)
2. Launch an IDCODE scan to determine TDO, TMS, TCK, TRST

Listing 4.4: Performing an IDCODE scan with the JTAGulator

```
JTAG> i

Enter starting channel [0]:

Enter ending channel [0]: 12

Possible permutations: 1716

Bring channels LOW between each permutation? [y/N]: y

Enter length of time for channels to remain LOW (in ms, 1 -
    1000) [100]:

Enter length of time after channels return HIGH before
    proceeding (in ms, 1 - 1000) [100]:

Press spacebar to begin (any other key to abort)...
JTAGulating! Press any key to abort...
-------------------------------------------
TDI: N/A
TDO: 11
TCK: 4
TMS: 12
Device ID #1: 0000 0101101100000010 00000011111 1 (0
    x05B0203F)
TRST#: 3
TRST#: 10

-------------------------------------------
IDCODE scan complete.
```

3. Launch a BYPASS scan to get TDI

Listing 4.5: Performing an BYPASS scan with the JTAGulator

```
JTAG> b

Enter starting channel [0]:

```

```
Enter ending channel [12]:

Are any pins already known? [Y/n]: y

Enter X for any unknown pin.
Enter TDI pin [0]: x

Enter TDO pin [11]:

Enter TCK pin [4]:

Enter TMS pin [12]:

Possible permutations: 10

Bring channels LOW between each permutation? [Y/n]: y

Enter length of time for channels to remain LOW (in ms, 1 -
    1000) [100]:

Enter length of time after channels return HIGH before
    proceeding (in ms, 1 - 1000) [100]:

Press spacebar to begin (any other key to abort)...
JTAGulating! Press any key to abort...
----
TDI: 5
TDO: 11
TCK: 4
TMS: 12
TRST#: 3
TRST#: 10
Number of devices detected: 1
------
BYPASS scan complete.
```

4. Read the device ID

Listing 4.6: Reading the device ID with the JTAGulator

```
JTAG> d

TDI not needed to retrieve Device ID.
Enter TDO pin [11]:

Enter TCK pin [4]:

```

```
8  Enter TMS pin [12]:
9
10
11 Device ID #1: 0000 0101101100000010 00000011111 1 (0
       x05B0203F)
12 -> Manufacturer ID: 0x01F
13 -> Part Number: 0x5B02
14 -> Version: 0x0
15 IDCODE listing complete.
```

The full session's transcript is available in Appendix C.1.2. Once the pinout is found, one can connect a JTAG adapter and start interacting with the microcontroller. In our case, we used a Raspberry Pi Zero as presented in Figure 4.13. The configuration of the Raspberry Pi Zero for that purpose is detailed in Appendix A.2.2.

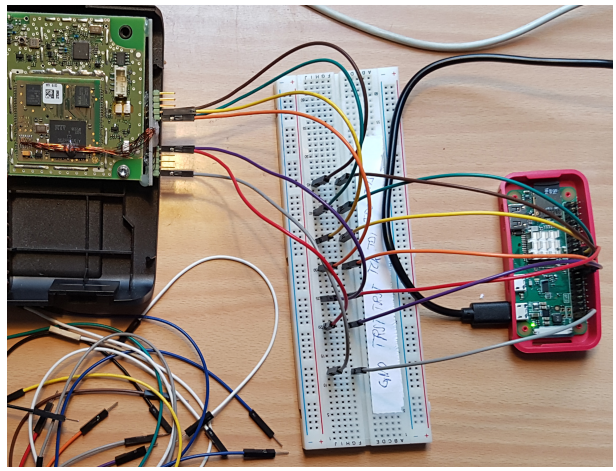


Figure 4.13: Using a Raspberry Pi as JTAG adapter

Figure 4.14 shows the pinout found using the JTAGulator.

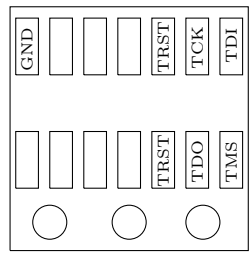


Figure 4.14: Mapping of the CardioMessenger II-S JTAG Pins

In some case, the JTAGulator might be useless, for instance if the JTAG interface has been disabled by the manufacturer. To confirm a hypothesis, an X-Ray imaging of the PCB can be performed. That way, by knowing the pinout of the microcontroller, one can follow the electric lines to find out which pins are part of the JTAG interface. This is particularly handy when the chip comes in a Ball Grid Array (BGA) package.

$\mathcal{H}_{2.2}$ Interaction with the JTAG pins
Knowing the JTAG pinout, an attacker can connect to the interface using low-cost equipment. Usually, to interact with a JTAG interface, proprietary connectors are used (such as the J-LINK adapters from Segger). However, connectors can be expensive. In the context of this thesis, we are trying to use off-the-shelves equipment and if possible, to use open-source project along with the cheapest hardware. That is why, OpenOCD is part our toolbox. Even though it could be more practical to use a proprietary interface, we present here a way to interact with a JTAG interface using OpenOCD along with a Raspberry Pi Zero W.

OpenOCD (Open On-Chip-Debugger) aims at providing debugging, in-system programming and boundary-scan testing for embedded target devices as explained by Rath [45]. Used along with a small adapter which provides the user with the required electrical signals, one can do on-chip-debugging or programming without the need of an external, and often expensive, programmer. As mentioned above, it can be used with a Raspberry Pi Zero, which costs around $10. An alternative to the Raspberry Pi, the Shikra, is presented in Appendix A.2.1. The drawback of using OpenOCD is that it can be unstable or contain bug. Furthermore, it is not necessarily straight forward to understand when used for the first time. Despite of this, it becomes a very powerful tool once the configuration's problems are solved. In our mind, the possibility to use "script" is also a plus.

The Raspberry Pi's configuration to work with OpenOCD is described in Appendix A.2. The pins of the Raspberry Pi Zero are connected to the HMU according to Table 4.1.

JTAG Pin	Raspberry Pi Pin[7]
TCK	23
TDI	19
TDO	21
TTMS	22
TRST (opt)	24

Table 4.1: Raspberry Pi's JTAG connection using OpenOCD

[7]Here the pins correspond to the GPIO's pins of the Raspberry and not to the microcontroller's pins.

From an attacker point of view, JTAG's debug functionalities are very interesting. With those features, it is possible to halt the CPU, read and write the memory but also to display the content of the registers. We wrote several OpenOCD configuration's scripts to help us automate certain tasks. They are available in Appendix B.3.

Figure 4.15 shows a session on OpenOCD's debug server, with the HMU's microcontroller as the target.

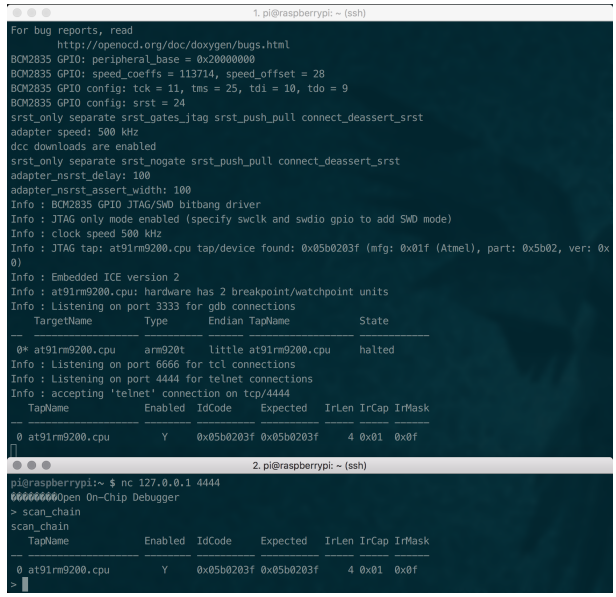


Figure 4.15: Interaction with the chip through JTAG

Once JTAG access granted, it is possible to gather very detailed information, including for instance, the content of the CPU's registers as demonstrated in Figure 4.16.

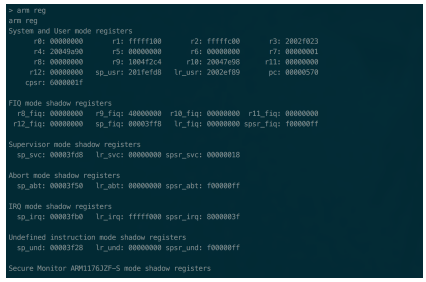


Figure 4.16: Available information through JTAG

$\mathcal{H}_{2.3}$ Dumping the memory

OpenOCD provides the user with a `dump_image` command which reads the memory.

By using the memory map of the microcontroller presented in Figure 4.17, it is easy to dump specific blocks of memory. Sections of interest include the bootloader, the content of the flash memory and of the external flash memory. Further memory analysis is performed in Subsection 4.2.4

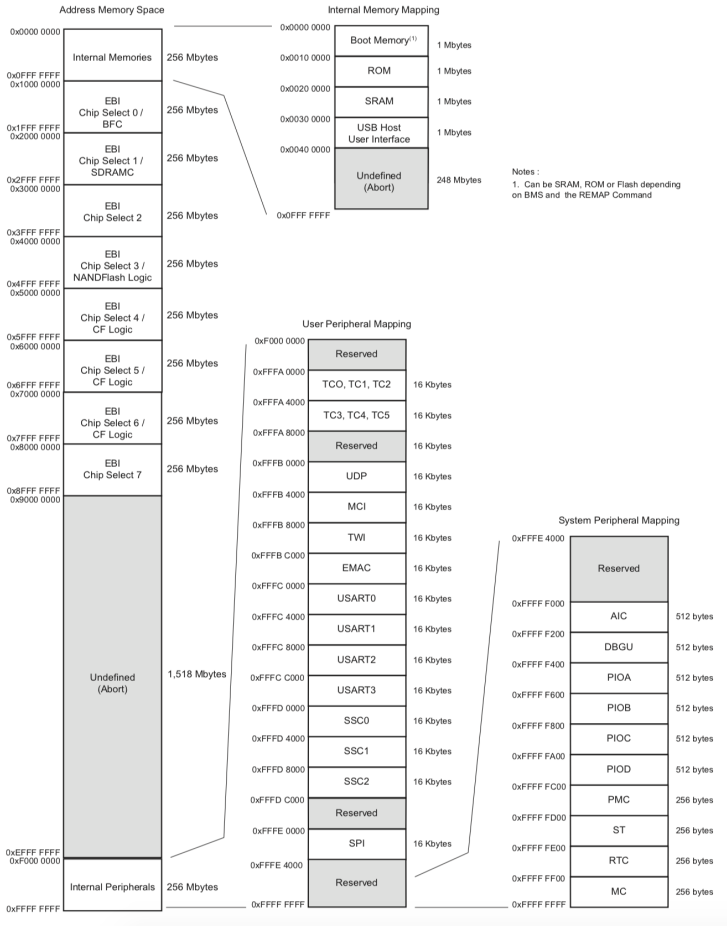


Figure 4.17: AT91RM9200 memory map [1]

To automatically dump the sections of the memory we were interested in, we used an OpenOCD configuration file, which is available in Appendix B.3. The results of the execution of that script is presented in Listing 4.7.

Listing 4.7: Dumping memory from the microcontroller

```
1      TargetName         Type     Endian TapName            State
2  --  --------------     -------  ------ --------------     -------
3   0* at91rm9200.cpu     arm920t  little at91rm9200.cpu     halted
```

```
Dumping bootloader...
dumped 1048576 bytes in 52.277569s (19.588 KiB/s)
Done!
Dumping SRAM...
dumped 104576 bytes in 5.229600s (19.528 KiB/s)
Done!
Dumping Flash content...
dumped 4194304 bytes in 208.740555s (19.622 KiB/s)
Done!
Dumping RAM...
dumped 2097152 bytes in 104.555344s (19.588 KiB/s)
Done!
```

$\mathcal{H}_{2.4}$ Alter the memory

Given a JTAG access, one can read and write the memory (both the RAM and the flash memory) along with the CPU's registers. That gives the ability to an attacker to alter the memory and thus to alter the data on it (for instance, the telephone number being used). It also means that an attacker can load her code in memory and execute it as she has control over the CPU's registers, and thus over the program execution. The `resume` command of OpenOCD allows the user to resume the execution with a specified value of `pc`.

As we did not want to brick the system in case the vendor implemented some checksum during the boot process, we did not try to write directly the flash memory. We have not reverse engineered enough of the firmware and bootloader to confirm that it is safe to do so. Nevertheless, we confirmed that it is possible to send data into the RAM by simply sending it through the modem's communication channel (see $\mathcal{H}_4$). That way we were able to write the string "NTNUNTNUNTNU" into the memory and to see it in the memory dump as one can see in Listing 4.8.

Listing 4.8: Visualizing strings sent in memory

```
$ cat ram.img | xxd | grep -i "NTNU"
0000b070: 554e 544e 554e 544e 554e 544e 550d 0a00  UNTNUNTNUNTNU...
000a9520: 0000 0000 0d0a 0d0a 4e54 4e55 4e54 4e55  ........NTNUNTNU
000a9530: 4e54 4e55 4e54 4e55 0d0a 3135 3237 0d0a  NTNUNTNU..1527..
```

Finally, it is possible, using OpenOCD, to directly load an image in RAM and to execute it. An attacker could then craft his own firmware and run it on the device.

Finding $\mathcal{F}_2$: The JTAG interface is enabled and available to anyone who has some soldering skills, giving an attacker the full control over the system. That includes reading and writing the memory but also, code execution.

The information gathered thanks to $\mathcal{F}_0$ is useful here as it allows the attacker to know that the microcontroller features JTAG which helps configuring OpenOCD's

target. However, it is worth noticing that, in fact, one does not really need it. Indeed, "*OpenOCD has a limited autoprobing ability to look at the scan chain, doing a blind interrogation and then reporting the TAPs it finds*".[8]

4.2.3 Eavesdropping and network emulation

Modem's communication channel ($\mathcal{H}_3$)

As highlighted in the hardware analysis, the CardioMessenger II-S has a modem on its board. The modem being the interface to the internet, patient's data are likely to pass through it to reach the vendor's servers. Moreover, on the T-Line version, that modem is on an external PCB that is plugged onto the main one, exposing its pins and allowing us to easily interact with it.

$\mathcal{H}_{3.1}$ Passive eavesdropping of the channel

Thanks to a guidance from Éireann Leverett who had already done some testing on the board, the first hypothesis was that it is possible to eavesdrop the communication channel between the microcontroller and the modem. By looking at the datasheet[9] of the modem, we identified the pins from the RS-232 standard, and particularly the UART ones. Those pins are shown on Figure 4.18. It is to be noted that the labels are here named from the DTE point-of-view which means TX correspond to the "input" of the modem's UART and RX to the "output".

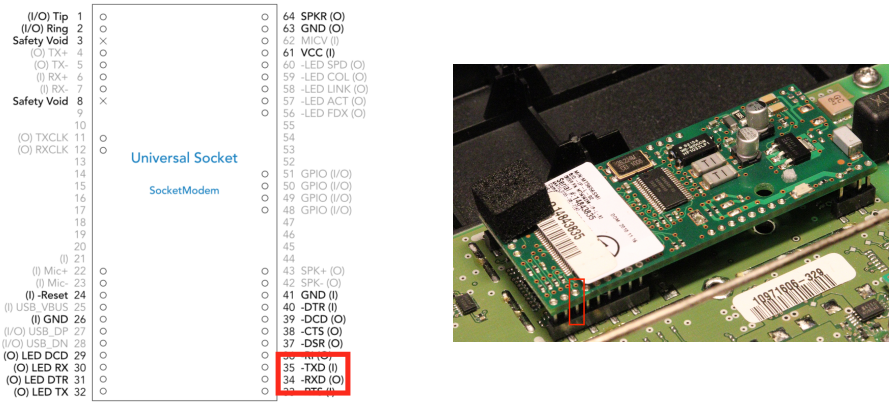


Figure 4.18: Modem RS-232 pins

Connecting to those pins with a Logic Analyzer and then with a USB-to-TLL adapter gave us the ability to monitor the exchanges between the modem and the

[8] According to OpenOCD's documentation, which can be downloaded at http://openocd.org/doc-release/pdf/openocd.pdf

[9] Available at http://www.multitech.com/documents/publications/data-sheets/86002071.pdf

microcontroller. Depending on which pin we were eavesdropping, we could either see what the microcontroller was sending to the modem, or the answer of the modem. Moreover, the modem was echoing all commands sent to it. Listing 4.9 presents the configuration of the modem by the microcontroller. This listing has been acquired through a script we developed and which is available in Appendix B.1.1.

Listing 4.9: Eavesdropping on the modem's pins

```
[2019-03-30 11:43:24] at
atii5#vversion
AT#MCOUNTRY?
at
AT+WOPEN=1
AT#DIALSELECT=1
AT#AUTHENT=PAP
at#atcmd=0,"-STE=7"
at#atcmd=1,"+A8E=6,5,0,1,0,0"
at#atcmd=2,"X3"
at
AT#DIALN1="T9W[REDACTED]"
AT#ISPUN="[REDACTED]@cm3-homemonitoring.de"
AT#ISPPW="[REDACTED]"
at#atcmd=0,"-STE=7"
[2019-03-30 11:43:26] AT#CONNECTIONSTART
[2019-03-30 11:43:28] at#connectionstop
```

The output provide several hints about the system and the communication patterns. One can notice that these commands are "AT commands", already described in Section 2.3.3. By looking at the "AT Commands for GSM/GPRS Wireless Modems with IP Connectivity"[10], we can learn that the modem is ordered by the microcontroller to connect to an Access Point Name (APN), which is the gateway between the mobile network and another network, usually the internet. Firstly, it starts the TCP/IP stack with the `WOPEN` command and then, selects the primary Dial number to be used with the `DIALSELECT` command. The authentication method is then set to Password Authentication Protocol (PAP). Finally, the number is set, along with the username and password required to connect to the APN. The way this type of connection works is described in Section 2.3.3. Username and password are sent in clear text. Also, we noticed that the username corresponds to the serial number of the device, which is printed on a label below it.

When connecting to "cm3-homemonitoring.de:80", an Apache welcome page is displayed. The Apache version is "Apache/2.2.15 (CentOS)". This version is affected by multiple vulnerabilities with a CVSS up to 7.8. Moreover, the certificate is not valid,

[10] Available at https://www.embeddedarm.com/documentation/third-party/ts-modem2_developerguide-gsm-gprs-ip-commands-s000333b.pdf

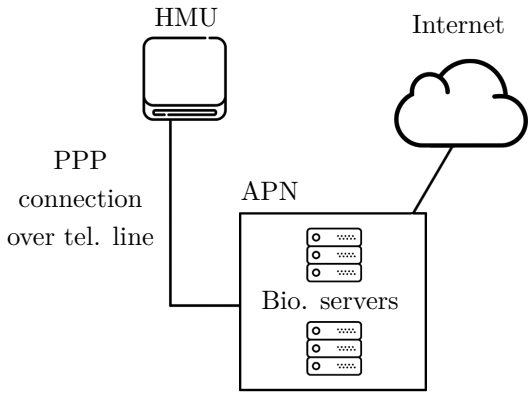


Figure 4.19: The HMU is accessing the internet using an APN

or there is no certificate. However, we do not know if that server is used for something or if the vendor is just using the domain name. No testings have been performed on that attack vector, as it is out of scope and would require the authorization of the vendor.

$\mathcal{H}_{3.2}$ **The credentials are still valid**
One can see that the command to close the connection is issued right after the command that asks for it to start. That is because the device was not plugged into a telephone line and the modem returned a "NO LINE" message to the microcontroller. As there were no telephone lines available, we wanted to verify whether or not the credentials were still valid. For that we tried two options:

1. Using a regular phone and trying to connect the APN
2. Trying to observe the connection on the CardioMessenger II-S GSM (see Section 4.3)

When trying to connect to the APN with the credentials using a regular phone, no access to the network is provided. The results being similar on the CardioMessenger II-S GSM, we believed that the credentials are revoked after a certain time or event (the patient is no longer using the service for example).

However, Lie confirmed that the credentials are still valid [34]. It is in fact the SIM card that is not registered anymore for the CardioMessenger II-S GSM. To do so, she took the SIM card of the CardioMessenger LLT2, whose PIN code was acquired using our APDU parser, and which seems to be able to connect to the vendor's server, and she used it with the CardioMessenger II-S GSM. The device was then able to obtain an IP, connect to the VPN and send its data.

$\mathcal{H}_{3.3}$ Dumping the modem configuration

In order to know what was the exact configuration of the modem, we dumped its full configuration. This was done thanks to the `VALL` command. By plugging our computer on the modem with the DTE convention (that means, RX on RX and TX on TX), we were able to block all commands sent by the microcontroller and to "spoof" it. We then dumped two times the modem's configurations. The first time, just after the start up to get its default values and a second one after the microcontroller has finished to configure it. The configuration confirms that PPP is used by the HMU but this testing did not bring any new information. Both configurations are included in Appendix C.1.1.

Finding $\mathcal{F}_3$: An attacker can gather APN's credentials by eavesdropping on the communication channel between the microcontroller and the modem. Those credentials are sent in cleartext and are still valid for at least some of the devices.

Network emulation ($\mathcal{H}_4$)

Since we did not have any available telephone lines, and to avoid interacting directly with the vendor's backend infrastructure, we decided to emulate the ISP's network. It would have been a lot of unnecessary work to configure a real modem to connect to the board, that is why we took advantage of the modem being removable from the PCB and plugged a USB-to-TTL connected to a computer instead. The goal was to see if it was possible to trick the microcontroller into thinking that the computer was the modem and to make it believe that the connection with the APN is established.

$\mathcal{H}_{4.1}$ Spoofing the modem

Emulating the modem manually worked, as we were able to get an answer from the microcontroller when responding "OK" to its first "at" command. To make it more practical we developed a script to act as the modem based on the communication we were able to eavesdrop. This script is available in Appendix B.1.2. The expected answers were guessed by looking at the "AT commands" reference. As a result, we noticed that the microcontroller is opening a Transmission Control Protocol (TCP) socket as soon as it got an IP, and is then sending data to a server with a fixed IP. Listing 4.10 presents an example of such an interaction with the microcontroller.

Listing 4.10: Spoofing the modem to interact with the microcontroller

```
1 < [2019-03-27 22:54:03] AT#DIALN1="T>W[REDACTED]"
2 > [2019-03-27 22:54:03] OK
3 < [2019-03-27 22:54:04] AT#ISPUN="[REDACTED]@cm3-homemonitoring.
    de"
4 > [2019-03-27 22:54:04] OK
5 < [2019-03-27 22:54:05] AT#ISPPW="[REDACTED]"
```

```
> [2019-03-27 22:54:05] OK
< [2019-03-27 22:54:06] at#atcmd=0,"-STE=7"
> [2019-03-27 22:54:06] OK
< [2019-03-27 22:54:09] AT#CONNECTIONSTART
> [2019-03-27 22:54:09] DIALING
> [2019-03-27 22:54:09] TW[REDACTED]
> [2019-03-27 22:54:09] CONNECT 115200
> [2019-03-27 22:54:09] 172.16.14.80
> [2019-03-27 22:54:09] OK_Info_PPP
< [2019-03-27 22:54:12] AT#TCPSERV=1,"172.16.14.1"AT#TCPPORT
   =1,4242AT#OTCP=1
> [2019-03-27 22:54:12] OK_Info_WaitingForData
---------- Switching to data mode -------------
< [2019-03-27 22:54:12] [REDACTED]@cm3-homemonitoring.de
[REDACTED]
< [2019-03-27 22:54:12] aa0005098e000000[REDACTED]0807204446
   d8b24981f356c69...
```

One can notice that the contacted server has a private IP. It means it is only accessible from inside the provider's network. This is generally a good practice as no one can access the server from the public internet. Also, the port in use here, 4242 is not reserved for any specific protocol. It is known as a port used by trojan, mostly because it repeats the number "42".[11] We believe that the connection established is a "telnet-like" connection. Indeed, one can notice that the first data that is sent after the switch to data mode is the same username and password used to connect to the APN. After having sent the data, nothing else happens. The microcontroller remains quiet and does not order the modem to quit the data mode. Eventually, it will reboot and the whole process will start again. The data sent is new all the time and its varies between 1kB and 3kB. It was also confirmed that nothing is happening on the other pins of the RS-232 connector. They can indeed be used to control the modem with hardware signals.

$\mathcal{H}_{4.2}$ Sending data

As we were able to spoof the modem and interact with it, we then wondered if it is possible to have more advanced interactions with it.

The fact that nothing happens after the data is sent to the modem does not seem normal. The modem is not order to return to the command mode and nothing else is sent before the whole device reboots. One hypothesis could be that the device is waiting for a response from the server. It was then tried to reply to what the device sends with first random data, and then by sending back the data. The goal was to generate a new flow of data that could be error messages for instance. However, we did not get any result. The only thing noticed is that the device crashed when too

[11] As explained on the following page: https://www.speedguide.net/port.php?port=4242.

much data were sent too quickly (sending the whole 3kB back). We further observed that the data sent is written in RAM (see $\mathcal{H}_{2.4}$).

$\mathcal{H}_{4.3}$ Scanning

The second hypothesis was that the device could be "scanned". Indeed, even though the TCP/IP stack is handled by the modem, PPP still defines a way for the microcontroller to be alerted in case an external device is trying to establish a connection with it: the `RING` string is sent by the modem to the microcontroller to notify it of an incoming call. It is then supposed to issue a command to answer or reject the call. In order to verify that, the modem script was adapted to send a `RING` string to the microcontroller at different moments (before the configuration starts, during the configuration and after data has been sent). None of those notifications gave a result. We also responded by the negative to the `WOPEN` command and sent the `RING` string instead. The device then stopped the connection and started all over again.

Finding $\mathcal{F}_4$: As demonstrated by our ability to spoof the connection, there is no mutual authentication between the microcontroller and the backend server, allowing an attacker to perform a Man in the Middle (MitM) attack.

Finding $\mathcal{F}_5$: The data is sent using telnet (or an equivalent program running over TCP) and the credentials are sent in cleartext over the communication channel allowing an attacker to eavesdrop the communication or to perform a MitM attack.

Finding $\mathcal{F}_6$: Credentials used to connect to the APN are reused to connect to the "telnet-like" service. An attacker can then get an access to the VPN and interact with the backend server using the same credentials.

Even though these three findings have a significant impact on the system, we have to highlight the fact that the vendor followed some good security practices. Firstly, the backend server is not exposed on the internet and a VPN access is required, which is a good practice. Secondly, the data is not sent in cleartext over the communication channel, but is encrypted with "strong" encryption (as shown by other findings, see $\mathcal{H}_8$) confirming the claim of the vendor in the manual of the CardioMessenger II-S, saying that data is encoded between the HMU and the backend server. "Encrypted" being here erroneously referred to as "encoded".

Data analysis ($\mathcal{H}_5$)

$\mathcal{H}_{5.1}$ Data is encrypted / encoded

Given the gathered data, our guess was that it is encrypted, encoded or compressed in

some way. To verify that assumption, we first used the `strings`[12] utility to confirm that there were no ASCII texts in it. Then, we used `binwalk`[13] on it to check for any known header but there was no results. Finally, we calculated its entropy to get an idea of whether it was encrypted or compressed. The entropy being close to 8 bits per byte (> 7.9), it is more likely to be encrypted or random data. Listing 4.11 shows the entropy of the data sent by the microcontroller, the username and password being excluded.

Listing 4.11: Entropy of the data gathered

```
$ cat hex_data.txt | xxd -r -p | ent
Entropy = 7.926409 bits per byte.

Optimum compression would reduce the size
of this 3379 byte file by 0 percent.

Chi square distribution for 3379 samples is 356.45, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 126.0648 (127.5 = random)
    .
Monte Carlo value for Pi is 3.175843694 (error 1.09 percent).
Serial correlation coefficient is 0.003429 (totally uncorrelated
     = 0.0).
```

While one can wonder which kind of data is being sent, we believe that there are three possibilities:

- Patients' data
- Communication related to software updates or HMU configuration
- Logs (errors, etc.)

$\mathcal{H}_{5.2}$ Basic reverse Engineering of the protocol used

Even though the entropy of the gathered data seems to show that is is either encrypted or random data, it does not mean that all the data is. Hence, we collected several samples and analyzed them together. It is worth mentioning that, before handling the data, one needs to "sanitize" it. Indeed, the characters `0x10` and `0x03` have to be escaped when sent to the modem. `0x03` is indeed the "End of text" character and "0x10" the escape character. They are then sent as `0x1003` and `0x1010`.

The data has been obtained by monitoring the HMU during 48h while our computer was emulating the modem. As a result the modem sent multiple times data on a

[12] As explained on the man page, the strings utility "finds the printable strings in a object or other binary."

[13] According to its man page, binwalk is a "tool for searching binary images for embedded files and executable code."

same “session” (that means, without being shutdown). We also gathered data from different sessions. Listing 4.12 presents the first bytes of two different data flows that have been gathered.

Listing 4.12: Hex dump of the first bytes of four data chunks gathered

```
05072e00000d[REDACTED]08090d909787d0c71d     # session 1
0502fe000000[REDACTED]0802b5742b63f5343d     # session 2
05030e000001[REDACTED]080d4caa019dee6028     # session 2
05030e000002[REDACTED]080db29ad304f90ffe     # session 2
```

A pattern exists in these data, giving a header as presented in Table 4.2.

Byte(s)	Value
0	Always `0x05`
1-2	Corresponds approximately to the size of the data sent
3-5	A counter that increments each time a packet is sent
6-9	The same 4 bytes sequence is repeated [REDACTED]. That sequence, converted to an integer corresponds to [REDACTED]which is the HMU’s serial number
10	Always `0x08`, which we think is the start of the encryption packet
11	Always between `0x00` and `0x0e`, which could be the padding

Table 4.2: Pattern of the gathered data

That pattern does not follow any known protocol header. It is most likely a proprietary protocol created by the vendor to communicate with its servers. At that point we had a guess on the structure of the header (see Figure 4.20) but no other information.

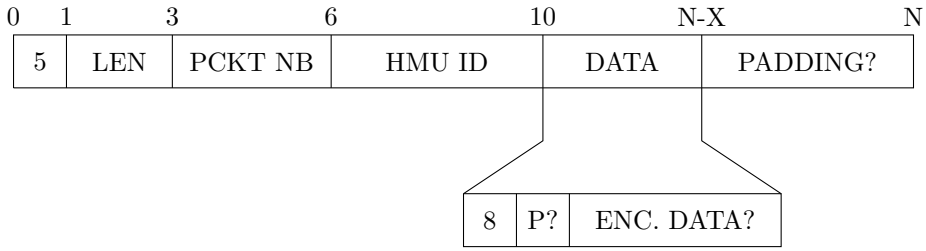


Figure 4.20: Structure of the protocol’s header

We also noticed that the length of the packet was always congruent to 14 modulo 16 (and by extension to 6 modulo 8). For instance, with the four samples from Listing 4.12:

$$\begin{array}{lcllll} 72e_{16} & = & 1838_{10} & \equiv & 14 & [16] \\ 2fe_{16} & = & 766_{10} & \equiv & 14 & [16] \\ 30e_{16} & = & 782_{10} & \equiv & 14 & [16] \end{array}$$

Even though the encryption algorithm used here is not known, this could be a good indication of the usage of a block cipher such as Data Encryption Standard (DES), Triple DES, Advanced Encryption Standard (AES) or Blowfish, which all have 64bits or 128bits key.

Finding $\mathcal{F}_7$: A proprietary protocol is used to send the data to the backend server. The HMU ID is available in the header, allowing an attacker to link a given packet to a given HMU (and with a given patient if she has the corresponding data).

4.2.4 Memory analysis and reverse engineering

Our analysis of the memory, and more specially of the firmware is divided into three main steps. First, we analyze the RAM and gather useful information that help in the reverse engineering of the firmware, i.e. debug strings, the memory mapping, etc. In a second step, we load the binary files into Ghidra[14] with the correct architecture and base addresses, which allows us to decompile it. Finally we perform some reverse engineering to understand the structure of the protocol used and attempt to decrypt the data sent by the HMU to the server.

RAM analysis ($\mathcal{H}_6$)

$\mathcal{H}_{6.1}$ The memory is not encrypted

The first operation performed after having dumped the RAM was to use `strings` to have an idea of what strings can be found in memory. There is no RAM encryption mechanism and one can gather interesting information. For instance, by looking for the string "src" in the output of the `strings` utility, we can get a list of the source files. There is a special formatting, which seems to match the "keyword substitution" feature of the Concurrent Versions System (CVS). We believe that they were using this feature to generate the logging messages automatically.

```
$ cat sdram.img | strings | grep -i "src"
...
$Source: src/syshelper.s $ $Revision: 1.2 $
$Source: src/Ulpamipair.c $ $Revision: 1.11 $
$Source: src/ulpami.h $ $Revision: 1.13 $
$Source: src/Utimer.c $ $Revision: 1.5 $
$Source: src/command.c $ $Revision: 1.3 $
```

[14] Describe in Section 4.2.4.

```
$Source: src/command.h $ $Revision: 1.2 $
$Source: src/crc.c $ $Revision: 1.2 $
$Source: src/crc.h $ $Revision: 1.2 $
$Source: src/fifo.c $ $Revision: 1.10 $
$Source: src/fifo.h $ $Revision: 1.2 $
$Source: src/flash.c $ $Revision: 1.27 $
$Source: src/flash.h $ $Revision: 1.6 $
$Source: src/fram.c $ $Revision: 1.11 $
$Source: src/fram.h $ $Revision: 1.9 $
$Source: src/gprs.c $ $Revision: 1.40 $
$Source: src/gprs.h $ $Revision: 1.13 $
$Source: src/gsm.c $ $Revision: 1.47 $
$Source: src/gsm.h $ $Revision: 1.12 $
...
```

$\mathcal{H}_{6.2}$ Credentials can be gathered from memory

We also confirm that the username and password used to connect to the VPN/APN are stored in memory and in cleartext as highlighted in Listing 4.2.4.

```
$ cat sdram.img | strings | grep -i "monitoring" -C 2
...
[REDACTED]@cm3-homemonitoring.de
[REDACTED]
...
```

$\mathcal{H}_{6.3}$ Usage of debug strings

Other interesting information gathered are the names of functions and error strings. More specifically, one can get the names of the functions sending the messages received while emulating the modem. Some error messages presented in Listing 4.2.4 contain the expected encryption algorithms. All this information is useful when reverse engineering the communication protocol as further explained in $\mathcal{H}_8$.

```
$ cat sdram.img | strings | grep -i "get"
...
Error: GetContainerFromGroup: sanity check failed
Error: GetContainerFromGroup: CRC error in msg container
GetDataFromMessageLayer sanity: status not OK
Wrong frame ID in GetDataFromMessageLayer
CRC check in GetDataFromMessageLayer
GetDataFromCompressionLayer sanity
GetDataFromEncryptionLayer: too many padding bytes
GetDataFromTransportLayer sanity
CRC check in GetDataFromTransportLayer
GetDataFromTransportLayer:start
TransportLayerToFifo: GetDataFromTransportLayer()
```

```
14 GetDataFromEncryptionLayer:start
15 TransportLayerToFifo: GetDataFromEncryptionLayer()
16 GetDataFromCompressionLayer:start
17 TransportLayerToFifo: GetDataFromCompressionLayer()
18 GetDataFromMessageLayer:start
19 TransportLayerToFifo: GetDataFromMessageLayer()
20 Get Container from Group:start
21 ...
22 GetDataFromEncryptionLayer: wrong ID byte (%02Xh): expected
       TRIPLE_DES_CBC (%02Xh) or AES_CBC (%02Xh)!
23 GetDataFromEncryptionLayer: wrong ID byte (%02Xh): expected DES
       (%02Xh), TRIPLE_DES_CBC (%02Xh) or AES_CBC (%02Xh)!
24 ...
```

$\mathcal{H}_{6.4}$ Files can be gathered from memory

Along with those findings we also found what seems to be a log file. It contains only the error message `NO PROVIDER DETECTED` as well as what is probably the name of the application and its version: `T4APP 1.20`.

`binwalk` has also been used on the RAM dump to see if any known file lies in it. On some of the RAM dumps performed, `binwalk` recognized a compressed file but not on all dumps. That file, once extracted and decompressed, is actually the same "log file" containing `NO PROVIDER DETECTED` multiples time. After having analyzed the firmware and the encryption protocol, we confirm that this is the file that is sent by the HMU. The data sent to the server is then available in cleartext in memory.

$\mathcal{H}_{6.5}$ Memory mapping

The last analysis performed on the memory dumps is the analysis of the entropy distribution. Indeed, understanding the structure of the memory is important to prepare the firmware's analysis. From the microcontroller datasheet [1], we expected the bootloader to repeat on itself due to the Memory Controller: "Within the Internal Memory address space, the Address Decoder of the Memory Controller decodes eight more address bits to allocate 1-Mbyte address spaces for the embedded memories. The allocated memories are accessed all along the 1-Mbyte address space and so are repeated n times within this address space, n equaling 1 Mbyte divided by the size of the memory." That is indeed what can be observed on the entropy graph presented in Figure 4.21.

We also expected the firmware to be loaded in RAM, which is the case when looking at their entropy distribution (see Figure 4.22). The common area are framed.

Finding $\mathcal{F}_8$: The memory is unencrypted, allowing an attacker to access in cleartext credentials, debug strings and all the data that is sent to the backend server.

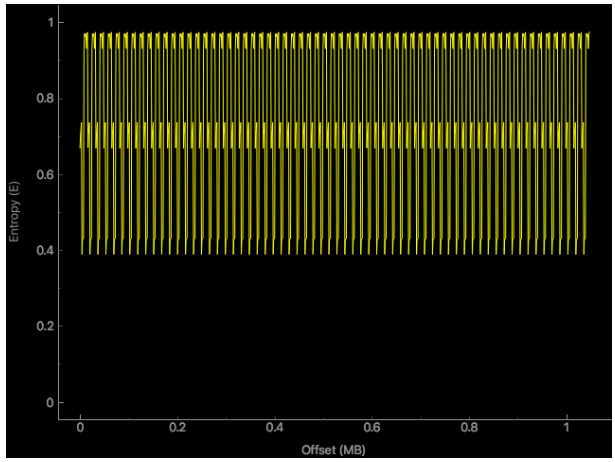
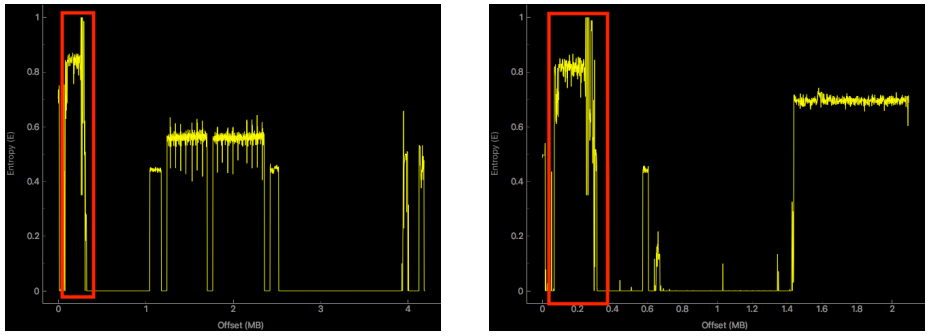


Figure 4.21: Entropy distribution of the bootloader's dump



(a) Burst Flash Memory **(b)** Random Access Memory

Figure 4.22: Entropy distribution of the ram and of the flash memory

Firmware analysis ($\mathcal{H}_7$)

$\mathcal{H}_{7.1}$ Mapping the firmware in Ghidra

As the National Security Agency (NSA) released its own reverse engineering tool in open source at the end of 2018, we decided here to try it as it is considered by some as the open source version of IDA Pro, which is a reference in the reverse engineering area. We installed it from the sources available on GitHub.[15]

The next step was to create a project in Ghidra and to import the dump files. For that, Ghidra needs some information such as the microcontroller's architecture and endianess, along with the base address for each file. All these information can be

[15] Available at https://github.com/NationalSecurityAgency/ghidra

found in the AT91RM9200's datasheet [1]. The mapping of the different files is the following:

- bootloader.img: ARM v4t little endian, base address: 0x0000 0000
- sdram.img: ARM v4t little endian, base address: 0x1000 0000
- flash.img: ARM v4t little endian, base address: 0x2000 0000

Even if Ghidra is easy to understand, part of the documentation written by Travis Goodspeed for the project md3801tools on GitHub[16] helped us applying a basic reverse engineering methodology to the firmware. Ghidra is able to perform basic analysis on the binaries and to provide us with functions, strings, disassembly code, etc. It might, however, get confused and not find the interrupt vector table at the beginning of the flash though, but that is not a problem for us.

$\mathcal{H}_{7.2}$ Locating the pacemaker's communication code in the firmware
The HMU communicates with both the pacemaker and the backend server. In the next section, the reverse engineering process of the communication protocol with the server is detailed. Even if it is not in our scope to analyze the communication link with the pacemaker, we still confirmed that the code is indeed available in the firmware. When looking at the strings obtained in $\mathcal{H}_{6.3}$, we find a lot of references to "ulpami". Ulpami is the acronym for Ultra Low Power Active Medical Implant. Those strings are then likely to be related to the communication with the pacemaker. When looking at their usage in the code, one can see the code in charge of dealing with the pacemaker communication, as presented in Figure 4.23.

$\mathcal{H}_{7.3}$ The firmware is common to both versions of the HMU
Several strings that are related to GSM and SMS were encountered during the firmware analysis. Now, the CardioMessenger II-S TLine does not have the required hardware to use the mobile network, which led us think that the firmware is common to both HMU. Listing 4.13 shows an example of the strings related to the mobile network that can be gathered in the firmware.

Listing 4.13: Gathered strings related to the GSM versions

```
READSMS_PARSE
READSMS_DELETE
SMS_OFF
SMS_COMMUNICATION
READ_SMS
DELETE_SMS
PARSE_INCOMING_SMS
LIST_SMS
[SKIPPED]
```

[16] Available at https://github.com/travisgoodspeed/md380tools/wiki/GHIDRA

```
else {
  if (*(char *)(iVar1 + 1) != 0) {
    FUN_1003145c(s_UlpamiInit:_Automatic_pairing_fo_1001488c);
  }
  if (*(ushort *)(iVar1 + 2) >> 8 != 2) {
    if (*(char *)(iVar1 + 1) != 0) {
      FUN_10031454(s_UlpamiInit:_Softpaired_/_Tent._P_100148bc);
      FUN_100314d4(*(undefined4 *)(iVar1 + 0x18));
      FUN_1003146a();
      FUN_1003145c(DAT_100148fc);
    }
    auStack40[0] = FUN_1001542a();
    puVar5 = *ppuVar3;
    uVar6 = *puVar5;
    uStack48 = auStack40[0] - uVar6;
    uStack44 = (extraout_r1_00 - puVar5[1]) - (uint)(auStack40[0] <
    if ((uStack48 >> 0xf | uStack44 * 0x20000) < uVar2) {
      *(ushort *)(iVar1 + 2) = *(ushort *)(iVar1 + 2) & 0x700;
      iVar4 = FUN_10033e66(*(undefined4 *)(iVar1 + 0x18));
      if (iVar4 == 0) {
        FUN_100140aa();
        return;
      }
      if (*(char *)(iVar1 + 1) != 0) {
        FUN_1003145c(s_UlpamiInit:_WM_restart_failed_10014844);
      }
```

```
void FUN_10013e4c(void)

{
  int iVar1;
  undefined4 uVar2;
  undefined4 uVar3;

  FUN_10031454(0x10013f94);
  iVar1 = DAT_10013f3c;
  FUN_100314d4(**(undefined4 **)(DAT_10013f3c + 0x44));
  FUN_1003146a();
  FUN_10031454(s_UlpamiNextStartTime2:_10013fac);
  FUN_100314d4(**(undefined4 **)(iVar1 + 0x60));
  FUN_1003146a();
  FUN_10031454(s_UlpamiNextStartTime0:_10013fc4);
  FUN_100314d4(**(undefined4 **)(iVar1 + 0x7c));
  FUN_1003146a();
  FUN_10031454(s_UlpamiTrialcount:_10013fdc);
  FUN_100314d4((uint)**(ushort **)(iVar1 + 0x98));
  FUN_1003146a();
  FUN_10031454(s_UlpamiPM2go:_10013ff0);
  FUN_100314d4((int)**(short **)(iVar1 + 0xb4));
  FUN_1003146a();
  FUN_10031454(s_UlpamiWMactive:_10014000);
  FUN_100314d4((int)**(short **)(iVar1 + 0xd0));
  FUN_1003146a();
  FUN_10031454(s_UlpamiWMsuccess:_10014014);
  FUN_100314d4((uint)**(byte **)(iVar1 + 0xec));
  FUN_1003146a();
  FUN_10031454(s_UlpamiPER:_10014028);
  FUN_100314d4(**(undefined4 **)(iVar1 + 0x108));
  FUN_1003146a();
  FUN_10031454(s_UlpamiPmsgsuccess:_10014034);
  FUN_100314d4((uint)**(byte **)(DAT_10013f44 + 0x94));
  FUN_1003146a();
  FUN_10031454(s_UlpamiLastRTC:_10014048);
  FUN_100314d4(**(undefined4 **)(iVar1 + 0x124));
  FUN_1003146a();
  uVar2 = (*(undefined4 **)(iVar1 + 0x140))[1];
  uVar3 = **(undefined4 **)(iVar1 + 0x140);
  FUN_10031454(s_UlpamiLasttime:_10014058);
  FUN_100314d4(uVar2);
  FUN_100314d4(uVar3);
  FUN_1003146a();
  return;
```

Figure 4.23: Decompiled code - Pacemaker communication

Finding $\mathcal{F}_9$: The firmware is not encrypted nor obfuscated, easing the reverse engineering task for an attacker.

Reverse Engineering the communication protocol ($\mathcal{H}_8$)

$\mathcal{H}_{8.1}$ Locating the code in charge of packets' creation

Our main objective with the reverse engineering of the firmware was to understand what is the data sent and what is the protocol structure. To do so, we used the strings found in the RAM, such as "GetDataFromMessageLayer". Indeed, once we knew where those strings are located RAM, we used the "find references" feature of Ghidra. That provides us with the functions where are referenced the strings. As a result, we got a function which seemed to be the packing function we were looking for so we name it "PackData". Part of that function's decompiled code is shown in Figure 4.24. All the functions and variables that have an english name in the rest of this thesis have been renamed by us during the reverse engineering process. The same applies to comments in code.

```
else {
                    /* // ************************************************************
                       //               START PACK TO MESSAGE LAYER
                       // ************************************************************ */
  if (1 < **(byte **)(puVar1 + 0xc)) {
    thunk_FUN_2002eb7c();
    printLog(s_PackToMessageLayer:_start_2002a458);
  }
  uVar3 = PackToMessageLayer(frame);
  if (uVar3 == 0) {
                    /* // ************************************************************
                       //            START PACK TO COMPRESSION LAYER
                       // ************************************************************ */
    if (1 < **(byte **)(puVar1 + 0xc)) {
      thunk_FUN_2002eb7c();
      printLog(s_PackToCompressionLayer:start_2002a4a4);
    }
    if (((((logicalPort == 1) || (logicalPort == 0)) || (logicalPort == 4)) ||
       ((logicalPort == 7 || (uVar3 = PackToCompressionLayer(frame,*iVar9), uVar3 == 0)))) {
                    /* // ************************************************************
                       //               START PACK TO ENCRYPTION LAYER
                       // ************************************************************  */
      if (1 < **(byte **)(puVar1 + 0xc)) {
        thunk_FUN_2002eb7c();
        printLog(s_PackToEncryptionLayer:start_2002a4f4);
      }
      if ((logicalPort == 3) || (logicalPort == 2)) {
        uVar3 = PackToEncryptionLayer
                          (frame,8,*(undefined4 *)PTR_PTR_LAB_2002a510,logicalPort,*iVar9);
      }
      else {
        if (((logicalPort == 1) || (logicalPort == 0)) || (logicalPort == 7)) {
          uVar3 = PackToEncryptionLayer(frame,6,PTR_DAT_2002a514,logicalPort,*iVar9);
        }
        else {
          if (logicalPort == 4) {
            uVar3 = PackToEncryptionLayer
                              (frame,6,*(undefined4 *)(PTR_PTR_LAB_2002a510 + 0x38),4,*iVar9);
          }
        }
      }
      if (uVar3 == 0) {
                    /* // ************************************************************
                       //               START PACK TO TRANSPORT LAYER
                       // ************************************************************ */
        if (1 < **(byte **)(puVar1 + 0xc)) {
          thunk_FUN_2002eb7c();
          printLog(s_PackToTransportLayer:start_2002a548);
        }
        puVar6 = PTR_DAT_2002a568 + logicalPort * 0x10;
        *(int *)(PTR_LAB_2002a564 + (int)&local_20 + iVar4 + 0x7c) = logicalPort * 0x10;
        *(undefined4 *)(&stack0x00000114 + iVar4) = *(undefined4 *)(**(int **)(puVar6 + 8) + 4);
        iVar2 = FUN_20017278(&stack0x000000ec + iVar4.0x28.*(undefined4 *)(puVar6 + 8).
```

Figure 4.24: Partial decompiled code of the PackData function

We were able to make sense of most of the code creating the packets (and also receiving the packets) by looking at the disassembly and by doing hypotheses on the role of some functions. Notably, we identified:

- The general function in charge of packing the data (whose code is partially exposed in Figure 4.24)
- The functions handling the encapsulation for each layer:
  - PackToMessageLayer
  - PackToCompressionLayer
  - PackToEncryptionLayer
  - PackToTransportLayer
- The compression format used is “deflate”, which is a “lossless compressed data format” defined to RFC1951.

– The encryption algorithms used: DES, 3DES CBC and AES CBC

When looking at the `PackToEncryptionLayer` we see the three algorithms used for encryption, as presented in Figure 4.27 at the end of this Section.

$\mathcal{H}_{8.2}$ Understanding the packets' structure
Thanks to $\mathcal{H}_{8.1}$, we confirmed most of our findings regarding the communication packets' structure exposed in $\mathcal{H}_{5.2}$. From there we updated our diagram on the communication protocol's structure. The full layers' encapsulation is presented in Figure 4.25.

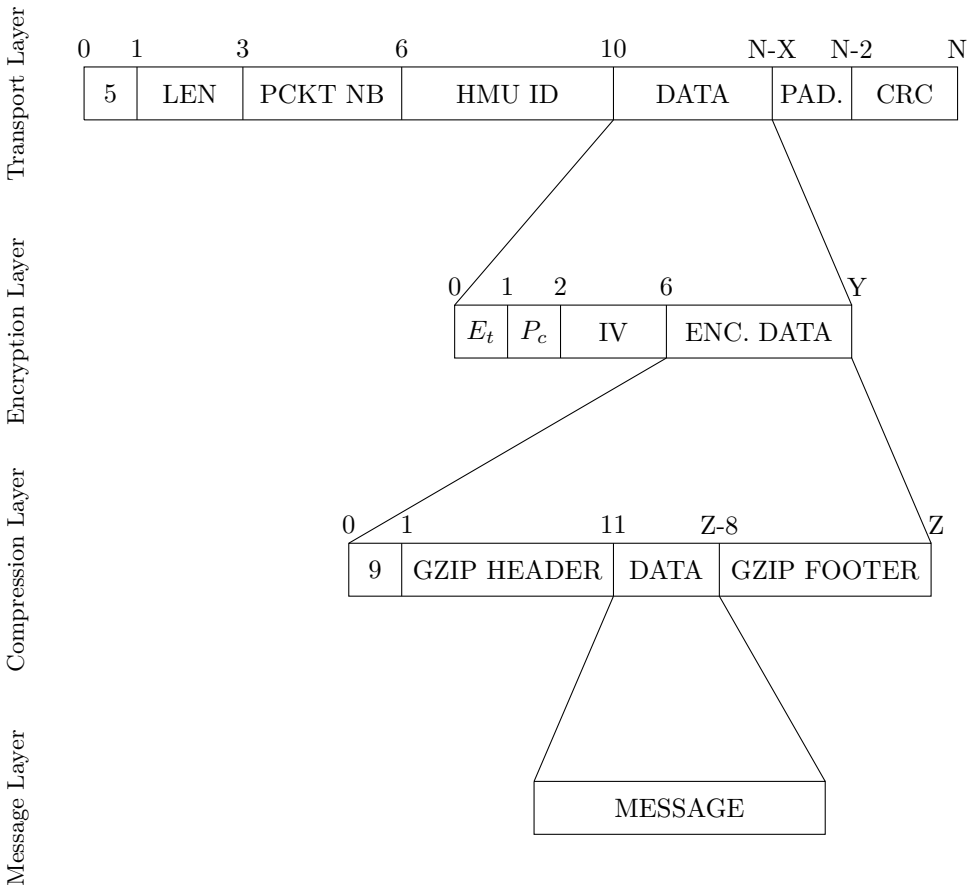
Based on our new understanding of the protocol, our initial script, that was only parsing the transport layer based on our guesses, has been updated. It is now able to parse all layers. However, for the decryption to be fully functional, the decryption key is still required.

$\mathcal{H}_{8.3}$ Finding the AES key
In order to find the key, we first thought that we could reverse engineer the code and understand how the key was obtained. From our first analysis, we believed that the key was derivated from a key derivation function. However, after trying to make sense of the code of that function, decrypting the data was still not possible and none of the possible key was working. Reproducing the "key derivation function" would have been time consuming and we decided to try other methods.

Having the disassembled code of the PackToEncryptionLayer function, one knows the address of the derivation function. Hence, we took advantage of having JTAG access to set hardware breakpoints at those specific addresses (before and after the call). Our idea was simple: the key is or is not in RAM before this call and has to be in RAM after as it is used for the encryption, which starts right after. Dumping the RAM at both this breakpoints and doing a diff on them should then give us the key (among other data). However, none of the 16 bytes sequences was clearly identified in the bytes that changed between the two dumps.

For the second try, the objective was to know what is the data that is supposed to be encrypted. We then reproduced the idea of dumping the RAM before a function's execution, but for each function of the layer. We could then see the evolution of the data in RAM at each layer. We noticed that data is modified in place during the packing process. The file is copied in RAM, then compressed and encrypted in place and finally the transport layer header and footer are added. At that point, we had the compressed data that is supposed to be encrypted, and its structure `0x9F + 0x1F8B + ...`, which is the same structure as a `gzip` file. The differences between the two files is presented in Figure 4.26.

With the knowledge of both the plaintext and the ciphertext, we then thought about finding the key by brute forcing the AES key. Of course, brute forcing the whole key space was not an option. However, the key is in RAM just before the encryption



E_t: Encryption type (8 = AES CBC; 7 = 3DES CBC; 6 = DES)

P_c: Padding from the compression layer

Gzip header: 10 bytes starting with `0x1F8B` (`0x1F`: compressed file; `0x8`: deflate)

Gzip footer: CRC and length of original data

Figure 4.25: Detailed structure of the communication protocol's packet

starts. The RAM being a file of 2Mb, we included a feature in our script to brute force the key by using the RAM file as the key space: we are taking a 16 bytes

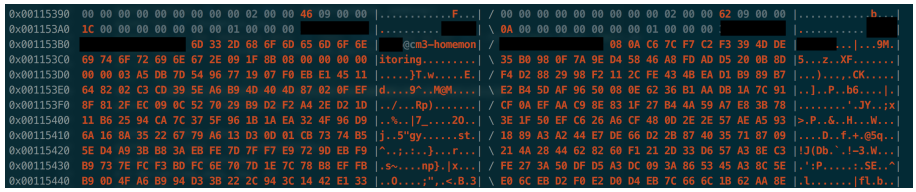


Figure 4.26: Content in RAM before and after PackToEncryptionLayer

windows and then shift it from one byte each try,[17] generating a new key.

Our python's script takes in average 1.9×10^{-5} s to decrypt the data with a given key. Based on that, we calculated that brute forcing the key would take in the worst case scenario[18] around

$$\frac{2\ 097\ 152}{\frac{1}{1.9} \times 10^5} \approx 40\text{s}$$

The flash dump's file can also be used, as discovered later. In fact, the key is stored at the beginning of that file. An example of the script's output is presented in Listing 4.14. The full script is available in Appendix B.2.

Listing 4.14: Brute forcing the AES key

```
$ python3 cm-decrypt.py  validation-tests/file_modem.bin -b
    validation-tests/ram_dump/ram_b_PackToEncryptionLayer.img
** CM DECRYPT v1.0 **

[*] Opening validation-tests/file_modem.bin...

-- FILE INFORMATION --

File size: 2514 bytes (hex: 9D2)
File created: Sat May 18 00:42:41 2019
File modified: Sat May 18 00:42:41 2019
File entropy: 7.91 bits per byte

[*] Data sanitized!
[*] Brute force mode
[*] Binary: validation-tests/ram_dump/
    ram_b_PackToEncryptionLayer.img

** Transport Layer **

```

[17] Here we were taking the whole RAM file without any other considerations. Of course, it would be possible to reduce the size of the key space even more by removing the null sections and the sections we know the key is not in.

[18] This value is not constant, but the time remains below 10 minutes.

```
Type of packets: 5 (hex: 05)
Length: 2478 bytes (hex: 09ae)
Unknown: 2 (hex: 02)
Packet ID: 0 (hex: 0000)
CM ID: [REDACTED] (hex: [REDACTED])
Checksum: 47070 (hex: b7de)

[*] Brute forcing the AES key...
[*] 2097152 keys to try
............................................................
............................................................
................
[*] Key Found in 37.69s!

Key: [REDACTED]
Addr: 0x0015d180
```

Any data sent by the HMU to the server can now be accessed. An example is presented in Listing 4.15.

Listing 4.15: Decrypting the data sent by the HMU

```
$ python3 cm-decrypt.py -k [REDACTED] validation-tests/
    file_modem.bin
** CM DECRYPT v1.0 **

[*] Opening validation-tests/file_modem.bin...

-- FILE INFORMATION --

File size: 2514 bytes (hex: 9D2)
File created: Sat May 18 00:42:41 2019
File modified: Sat May 18 00:42:41 2019
File entropy: 7.91 bits per byte

[*] Data sanitized!
[*] Decrypt mode
** Transport Layer **

Type of packets: 5 (hex: 05)
Length: 2478 bytes (hex: 09ae)
Unknown: 2 (hex: 02)
Packet ID: 0 (hex: 0000)
CM ID: [REDACTED] (hex: [REDACTED])
Checksum: 47070 (hex: b7de)

```

```
** Encryption Layer **

Length of the packet: 2466 (hex: 9a2) (div by 16? No)
Type of packet: 8 (hex: 08) => AES_CBC
Padding: 15 (hex: 0f)
IV: 5c8aff9dd3a7bb54f04915bc18e96fbb
Key: [REDACTED]

** Compression Layer **

Compression packet: Yes
Magic header: 0x1f8b  => gzip compressed data, from Unix
Entropy: 7.72

** Message Layer **

Size of the recovered data: 28538 bytes
Number of packets: 105
Entropy: 3.94
```

Finding $\mathcal{F}_{10}$: The usage of debug strings, even if debug is not enabled, ease the reverse engineering of the communication protocol.

Finding $\mathcal{F}_{11}$: The AES key is hard coded in the memory, allowing an attacker who accesses it to decrypt all outgoing communication from the HMU, and to forge her own packets.

```
                    /* DES Encryption */
if (enc_type == 6) {
  iVar3 = 0;
  puStack36 = data_to_encrypt;
  iStack32 = enc_type;
  passphrase = enc_password;
  while (iVar3 <= (int)(((uint)((int)*data_to_encrypt >> 0x1f) >> 0x1d)
  {
    mem_n_cpy(auStack264,passphrase,8);
    DES_EncryptBlock(0,remain_to_encrypt,auStack264);
    remain_to_encrypt = remain_to_encrypt + 8;
    iVar3 = iVar3 + 1;
  }
  iVar3 = (int)*data_to_encrypt % 8;
  if (iVar3 != 0) {
    iVar3 = 8 - iVar3;
  }
  uVar2 = data_to_encrypt[1];
  data_to_encrypt[1] = uVar2 - 1;
  *(undefined *)((uVar2 - 1) + (int)data_start) = (char)iVar3;
  uVar2 = data_to_encrypt[1];
  data_to_encrypt[1] = uVar2 - 1;
  *(undefined *)((uVar2 - 1) + (int)data_start) = 6;
  *data_to_encrypt = *data_to_encrypt + iVar3 + 2;
}
```

(a) DES

```
                    /* 3DES_CBC Encryption */
if (enc_type == 7) {
  puStack36 = data_to_encrypt;
  iStack32 = enc_type;
  passphrase = enc_password;
  read_random(init_vector,8);
  uVar2 = data_to_encrypt[1];
  data_to_encrypt[1] = uVar2 - 8;
  mem_n_cpy((uVar2 - 8) + (int)data_start,init_vector,8);
  iVar3 = 0;
  while (iVar3 <= (int)(((uint)((int)*data_to_encrypt >> 0x1f) >> 0x1d)
        ) {
    iVar1 = 0;
    do {
      iVar3 = iVar1;
      *(byte *)(remain_to_encrypt + iVar3) =
           *(byte *)(remain_to_encrypt + iVar3) ^ init_vector[iVar3];
      iVar1 = iVar3 + 1;
    } while (iVar3 + 1 < 8);
    mem_n_cpy(auStack264,passphrase,8);
    DES_EncryptBlock(0,remain_to_encrypt,auStack264);
    mem_n_cpy(auStack264,passphrase + 8,8);
    DES_EncryptBlock(1,remain_to_encrypt,auStack264);
    mem_n_cpy(auStack264,passphrase + 0x10,8);
    DES_EncryptBlock(0,remain_to_encrypt,auStack264);
    mem_n_cpy(init_vector,remain_to_encrypt,8);
    remain_to_encrypt = remain_to_encrypt + 8;
    iVar3 = iVar3 + 2;
  }
  iVar3 = (int)*data_to_encrypt % 8;
  if (iVar3 != 0) {
    iVar3 = 8 - iVar3;
  }
  uVar2 = data_to_encrypt[1];
  data_to_encrypt[1] = uVar2 - 1;
  *(undefined *)((uVar2 - 1) + (int)data_start) = (char)iVar3;
  uVar2 = data_to_encrypt[1];
  data_to_encrypt[1] = uVar2 - 1;
  *(undefined *)((uVar2 - 1) + (int)data_start) = 7;
  *data_to_encrypt = *data_to_encrypt + iVar3 + 2;
}
```

(b) 3DES CBC

```
                    /* AES_CBC Encryption */
if (enc_type == 8) {
  puStack36 = data_to_encrypt;
  iStack32 = enc_type;
  passphrase = enc_password;
  read_random(init_vector,0x10);
  size_data_iv = data_to_encrypt[1];
  data_to_encrypt[1] = size_data_iv - 0x10;
  mem_n_cpy((size_data_iv - 0x10) + (int)data_start,init_vector,0x10);
  unknown_key_operation((byte *)passphrase,key_aes_cbc);
  while (0xf < (int)sizeToEncrypt) {
    counter = 0;
    do {
      dataToEncrypt[counter] =
           ((undefined *)remain_to_encrypt)[counter] ^ init_vector[counter];
      counter = counter + 1;
    } while (counter < 0x10);
    uVar2 = AES_EncryptBlock(dataToEncrypt,(undefined *)remain_to_encrypt,key_aes_cbc);
    data_to_encrypt[2] = uVar2;
    if (uVar2 != 0) {
      printLogWithParam(PTR_s_Error:_%s_20029d38,s_Block_encryption_error_20029d20);
      return data_to_encrypt[2];
    }
    mem_n_cpy(init_vector,remain_to_encrypt,0x10);
    sizeToEncrypt = sizeToEncrypt - 0x10;
    remain_to_encrypt = (int)((undefined *)remain_to_encrypt + 0x10);
  }
  if (sizeToEncrypt != 0) {
    counter2 = 0;
    while (uVar2 = sizeToEncrypt, counter2 < (int)sizeToEncrypt) {
      dataToEncrypt[counter2] =
           ((undefined *)remain_to_encrypt)[counter2] ^ init_vector[counter2];
      counter2 = counter2 + 1;
    }
    while ((int)uVar2 < 0x10) {
      dataToEncrypt[uVar2] = init_vector[uVar2];
      uVar2 = uVar2 + 1;
    }
    uVar2 = AES_EncryptBlock(dataToEncrypt,dataToEncrypt,key_aes_cbc);
    data_to_encrypt[2] = uVar2;
    if (uVar2 != 0) {
      printLogWithParam(PTR_s_Error:_%s_20029d38,s_Block_encryption_error_20029d20);
      return data_to_encrypt[2];
    }
    mem_n_cpy(remain_to_encrypt,dataToEncrypt,0x10);
    padding = 0x10 - (char)sizeToEncrypt;
  }
```

(c) AES CBC

Figure 4.27: Different encryption algorithms used in PackToEncryptionLayer

4.3 Security Analysis of the CardioMessenger II-S GSM

4.3.1 Hardware analysis

As one can notice in Figure 4.28, the CardioMessenger II-S GSM is very similar to the TLine version. The only differences stand in the modem and its surroundings components. Similarly, there is no obfuscation of hardware components.

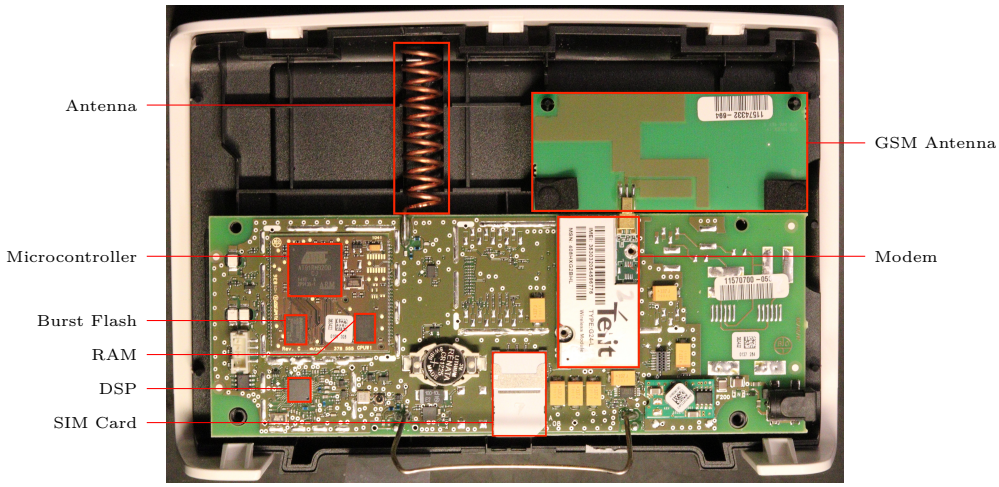


Figure 4.28: Inside of the CardioMessenger II-S T-Line

As with the CardioMessenger II-S, multiple elements can be identified and searched on the internet for datasheets. Apart from the modem, they are in fact the same.

- The microcontroller which is based on an ARM Thumb Processor and featured with debug and UART interfaces
- A Burst Flash memory with a capacity of 4Mb
- An external RAM
- A DSP
- A modem that can provide an internet access using the mobile network
- An antenna that is most likely to be used to communicate with the pacemaker
- A second antenna that is used for the mobile broadband
- A SIM card from “t-mobile”, a german company

The same PCB is used for both CardioMessenger II-S, that is why one can see the circuit where the TLine modem is supposed to be. Due to these similarities, most of the hypotheses tested on the TLine version gave the same results on the GSM version. Hence, we only present in this section the differences that have been noticed along with the new hypotheses related to the mobile network. We have not been able to assert the hypotheses regarding the JTAG interface and the memory analysis due to the complexity of the soldering, but we are confident that these findings are repeatable.

4.3.2 Eavesdropping and network emulation

Modem's communication channel ($\mathcal{H}_3$)

$\mathcal{H}_{3.1}$ Passive eavesdropping of the channel

On the CardioMessenger II-S TLine, the modem was on its own PCB, which was plugged on the main board, exposing its pins and easing the eavesdropping task. On the GSM version, the modem is also plugged on the main board but is not exposing its pins. Even if it would be possible to remove it and connect our computer in between, we took instead advantage of the fact that both CardioMessenger have the same PCB. Indeed, by eavesdropping on the circuit where the modem of the TLine version is supposed to be plugged, one can get the same results. It is thus possible to access to the communication between the microcontroller and the modem and to get the "AT commands". Such an eavesdrop session is presented in Listing 4.16

Listing 4.16: AT commands sent by the microcontroller to the GSM modem

```
1  [2019-03-06 14:25:12] AT
2  [2019-03-06 14:25:13] AT+CGMR
3  [2019-03-06 14:25:13] AT+MBAND?
4  [2019-03-06 14:25:13] AT+CPIN?
5  [2019-03-06 14:25:13] AT+CPIN="[REDACTED]"
6  [2019-03-06 14:25:14] AT+CPIN?
7  [2019-03-06 14:25:14] AT+CGSN
8  [2019-03-06 14:25:14] AT+CIMI
9  [2019-03-06 14:25:14] AT+CRSM=176,242
10 [2019-03-06 14:25:14] ATS24=0
11 [2019-03-06 14:25:15] ATS100=0
12 [2019-03-06 14:25:15] ATS102=0
13 [2019-03-06 14:25:15] AT+MSCTS=0
14 [2019-03-06 14:25:20] AT+CREG?
15 [2019-03-06 14:25:22] AT+CREG?
16 [2019-03-06 14:25:24] AT+CREG?
17 [2019-03-06 14:25:24] AT+COPS?
18 [2019-03-06 14:25:29] AT+CGPRS
```

```
19 [2019-03-06 14:25:29] AT+CGMI
20 [2019-03-06 14:25:29] AT+COPS=3,2
21 [2019-03-06 14:25:29] AT+COPS?
22 [2019-03-06 14:25:29] AT+COPS=3,0
23 [2019-03-06 14:25:29] AT+CSQ
24 [2019-03-06 14:25:32] AT+CSQ
25 [2019-03-06 14:25:32] AT+CREG=2
26 [2019-03-06 14:25:32] AT+CREG?
27 [2019-03-06 14:25:32] AT+CREG=0
28 [2019-03-06 14:25:38] AT
29 [2019-03-06 14:25:38] AT+CPMS="SM"
30 [2019-03-06 14:25:38] AT+CMGL=4
31 [2019-03-06 14:25:38] AT+MIPCALL=1,"[REDACTED]","[REDACTED]@cm3-
       homemonitoring.de","[REDACTED]"
32 [2019-03-06 14:30:38] AT+MRST
```

The operations performed are the same than the one on the TLine version. The difference here, is that the configuration is for a GSM modem. One can notice that is again possible to gather cleartext credentials to connect to the VPN, along with the PIN code of the SIM card.

The `MIPCALL` command is asking the modem to set up the IP stack and is supposed to provide the HMU with an IP in the VPN network. However, that command seems to fail which is why the HMU resets the modem after five minutes and starts the whole process again.

Even though it is possible to eavesdrop the communication on that circuit, it is not possible to reproduce all hypotheses tested against the CardioMessenger II-S TLine. It is indeed not possible to eavesdrop on the "RX" pin (DTE convention).

$\mathcal{H}_{3.2}$ Credentials are still valid
As already explained for the TLine, credentials seems revoked, as it is not possible to use them to connect to the APN with a regular phone and the CardioMessenger II-S GSM is not able to connect anymore to the server. However, after Lie used the SIM card from the CardioMessenger LLT2 in the CardioMessenger II-S GSM, she confirmed that the credentials are still valid and it is in fact the SIM card which is not valid anymore. As the PIN codes of the two SIM cards were not the same, she used the information provided by the APDU script (described in Section 4.5) to obtain the PIN code of the SIM card of the CardioMessenger LLT2 and then reprogram it with the PIN code of the CardioMessenger II-S GSM.

Finding $\mathcal{F}_{12}$: The PCB design of the CardioMessenger II-S is the same for both versions, exposing its circuit to an attacker who can then eavesdrop the communication between the microcontroller and the modem.

Network emulation ($\mathcal{H}_4$)

Given the results of the previous hypotheses, the next step was to emulate the network to trick the HMU into thinking it can send the information to the server. However, spoofing the modem is this time more complex as one cannot simply unplug it and plug a computer instead (without a specific connector). Consequently, we collaborated with Lie to use the Fake Base Station they set up and emulate the VPN connection. When provided the HMU with a internet access, it sends another command, after the `MIPCALL` command, to open a TCP socket to a specific IP, which is private and the same as the one used in the TLine version. This is presented in Listing 4.17.

Listing 4.17: The HMU trying to contact the server

```
1 [2019-03-04 10:31:20] AT+CMGL=4
2 [2019-03-04 10:31:20] AT+MIPCALL=1,"[REDACTED]","[REDACTED]@cm3-
      homemonitoring.de","[REDACTED]"
3 [2019-03-04 10:31:24] AT+MIPOPEN=1,3953,"172.16.14.1",2323,0
```

From there, one can guess the overall network architecture that is normally used by the HMU to communicate with Biotronik's servers. Figure 4.29 shows what we guess is expected by the HMU.

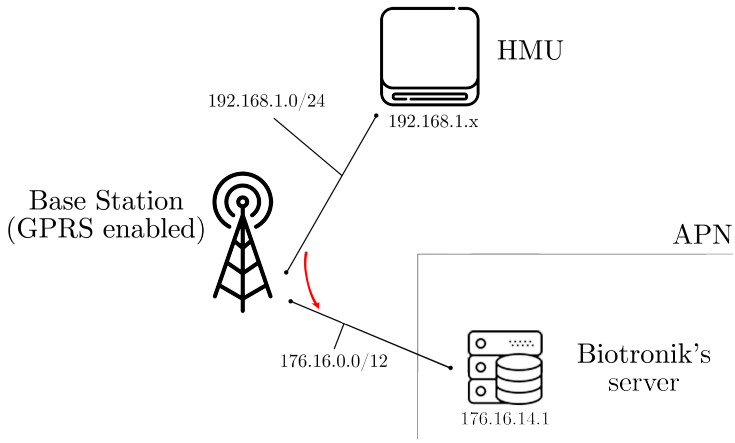


Figure 4.29: Guess of the network architecture between the HMU and the servers

To emulate that architecture, we added an Ubuntu machine in a private network with the IP address requested by the HMU. The routing table are then configured so that the HMU can contact the server. We opened a TCP socket on that server, on port `2323` to see if the HMU sends something. Figure 4.30 presents the network diagram of the emulated network.

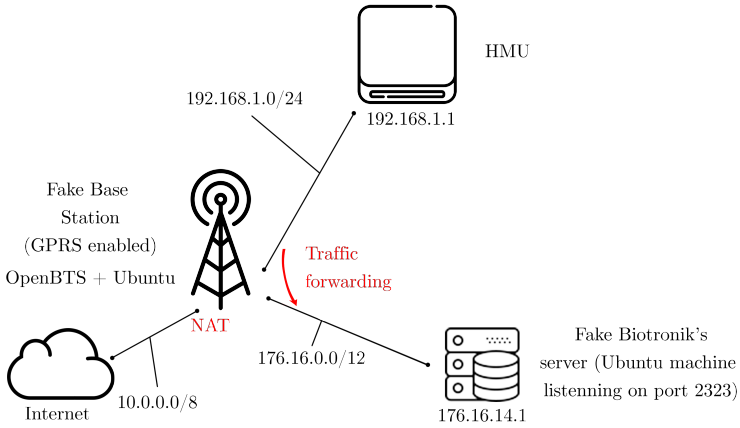


Figure 4.30: Network diagram of the emulated network

As a result of the emulation, the HMU contacted the server and after having sent the credentials (which are again the same as the one used for the VPN), it sends data (see Listing 4.18).

Listing 4.18: The HMU contacting the fake server

```
[2019-03-01 13:35:35] AT+CMGL=4
[2019-03-01 13:35:35] AT+MIPCALL=1,"[REDACTED]","[REDACTED]@cm3-
    homemonitoring.de","[REDACTED]"
[2019-03-01 13:35:36] AT+MIPOPEN=1,6407,"172.16.14.1",2323,0
[2019-03-01 13:35:41] AT+MIPSETS=1,1372
[2019-03-01 13:35:41] AT+MIPSEND=1,"[REDACTED]"
[2019-03-01 13:35:41] AT+CSQ
[2019-03-01 13:35:41] AT+MIPPUSH=1
[2019-03-01 13:35:46] AT+MIPSEND=1,"050DFE000000[REDACTED]080BE6
E2F7AA0FCD96EC016D65E2A28E09F23E5AB69B404957EAB038A3EC4472ABF6
9C50F282C0A45C8F9C578315E97871828432DFD777921451993950F40FCB1E
F9"
[2019-03-01 13:35:46] AT+MIPSEND=1,"941E44938B37B27EC1E99F6772
93A2A08A784C4A2AEDD44873812D44183D9FD896687105326ABFAC7213347A
D7398BA48D86E8A555AFA868C2862F893B7C46B9A43B01103B0A17C7C692F0
34"
[Cropped]
```

Data analysis ($\mathcal{H}_5$)

As one can see from Listing 4.18, the data sent follows the same scheme as the one sent by the TLine version. First, the HMU also sends the credentials to connect to

the service and then, the data themselves.

The protocol being the same, we wondered if the AES key is also the same. However, as shown in Listing 4.19, the key found in the TLine HMU does not decrypt the data successfully.

Listing 4.19: Using the AES key from the TLine version on the data

```
$ python3 cm-decrypt.py -n -k [REDACTED] files_extracted/data.bin
** CM DECRYPT v1.0 **

[*] Opening files_extracted/data.bin...

-- FILE INFORMATION --

File size: 3582 bytes (hex: DFE)
File created: Mon May 13 11:19:37 2019
File modified: Mon May 13 11:19:37 2019
File entropy: 7.94 bits per byte

[*] Decrypt mode
** Transport Layer **

Type of packets: 5 (hex: 05)
Length: 3582 bytes (hex: 0dfe)
Unknown: 0 (hex: 00)
Packet ID: 0 (hex: 0000)
CM ID: [REDACTED] (hex: [REDACTED])
Checksum: 15713 (hex: 3d61)

** Encryption Layer **

Length of the packet: 3570 (hex: df2) (div by 16? No)
Type of packet: 8 (hex: 08) => AES_CBC
Padding: 11 (hex: 0b)
IV: e6e2f7aa0fcd96ec016d65e2a28e09f2
Key: [REDACTED]

** Compression Layer **

[ERROR] Not a compression layer packet!
```

Given the fact the firmware is the same, our second hypothesis was that the key of the GSM version was different but still included in the firmware. So we tried to

brute force using the firmware file. However it did not work either as highlighted by Listing 4.20.

Listing 4.20: Bruteforcing the AES key using the firmware file

```
$ python3 cm-decrypt.py -n -b validation-tests/ram_dump/
    flash_content_start.img files_extracted/data.bin
** CM DECRYPT v1.0 **

[*] Opening files_extracted/data.bin...

-- FILE INFORMATION --

File size: 3582 bytes (hex: DFE)
File created: Mon May 13 11:19:37 2019
File modified: Mon May 13 11:19:37 2019
File entropy: 7.94 bits per byte

[*] Brute force mode
[*] Binary: validation-tests/ram_dump/flash_content_start.img

** Transport Layer **

Type of packets: 5 (hex: 05)
Length: 3582 bytes (hex: 0dfe)
Unknown: 0 (hex: 00)
Packet ID: 0 (hex: 0000)
CM ID: [REDACTED] (hex: [REDACTED])
Checksum: 15713 (hex: 3d61)

[*] Brute forcing the AES key...
[*] 4194304 keys to try
............................................................
............................................................
............................................................
............................................................
............................................................
............................................................
..........................................
Key not found! Time elapsed: 147.8s
```

In conclusion, we believe the vendor is not reusing the same AES key for all devices (at least not for both the GSM and TLine versions). Given the way these specific variables are stored in memory (AES key, HMU ID, credentials), it is likely that the firmware are generated automatically for each device and that the tuples

(HMU ID, password, AES key) are also stored on a server in the VPN.

4.3.3 Mobile network

This section presents the results obtained when looking at the SMS communication with the HMU. We are only analyzing the SMS from the point of view of the HMU, i.e. as a payload sent to the modem to be sent via SMS as communication channel. What happens outside the HMU is out of our scope and is not presented in this thesis. More information can be found in the thesis from Lie [34].

Collecting SMS ($\mathcal{H}_9$)

$\mathcal{H}_{9.1}$ SMS gathering
When monitoring the device for several hours, one can notice that it is sending data to the server using a TCP socket, but it is also sending SMS as it can be observed in Listing 4.21 (the `CMGS` is the command used to send SMS).

Listing 4.21: SMS are sent by the HMU

```
[2019-03-09 02:55:11] AT+CPMS="SM"
[2019-03-09 02:55:11] AT+CMGL=4
[2019-03-09 02:55:11] AT+CMGF=0
[2019-03-09 02:55:11] AT+CMGS=120
[2019-03-09 02:55:11] 07919[REDACTED]011000C919[REDACTED]00F6
C86As0604FEB0555B4374D1C5EF24287FA3954CA200AFBBDC44C170DB1A8C4
52DC03891BD43302DAC8DA33CE225FF99E976F9B066CA00AA5A25AB8A47218
D21F232ED41AED4E0F22F42E6189968D7CF6B965B73A768D7CF6B965B73A76
8D7CF6B965B73A7FE059B460232575656
```

Plotting the timestamps of each SMS did not reveal any pattern, as shown in Figure 4.31.

$\mathcal{H}_{9.2}$ SMS analysis
When analyzing the structure of the data sent, one can see that it is not random data. There is indeed a header. However, it seems to be the header of an SMS PDU (one can indeed see the destination phone number in Big endian: [REDACTED], i.e. +47[REDACTED]). The packet we are interested in starts at `06`. It indeed seems to be a packet from the encryption layer, `06` being the code for the DES algorithm. We are however not certain of that assumption.

Finding $\mathcal{F}_{13}$: SMS are used as a mean of communication.

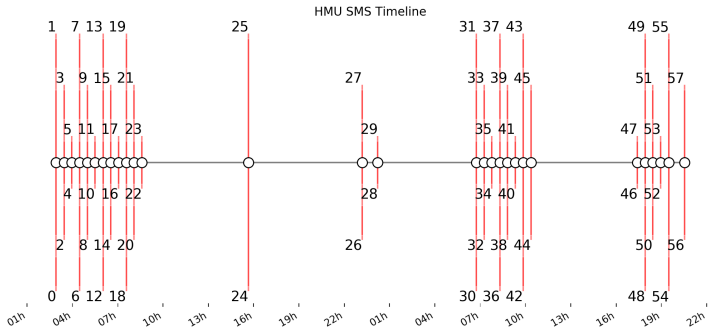


Figure 4.31: Timeline of the SMS sent by the HMU

SMS as an attack vector ($\mathcal{H}_{10}$)

$\mathcal{H}_{10.1}$ Reception of the SMS by the HMU

Using the Fake Base Station, SMS were sent to the HMU to see if it would trigger something. It is indeed reading and deleting them from the SIM straight after as it can be observed in Listing 4.22.

Listing 4.22: SMS are read by the HMU

```
[2019-03-06 09:34:01] AT+MIPCALL=0
[2019-03-06 09:34:01] AT+CPMS="SM"
[2019-03-06 09:34:01] AT+CMGL=4       # List all SMS
[2019-03-06 09:34:02] AT+CMGD=1       # Delete SMS at index 1
[2019-03-06 09:34:08] AT+MRST
```

$\mathcal{H}_{10.2}$ Parsing of the SMS by the HMU

Even if the HMU is listing and deleting the SMS, we do not know if it is parsing them. Given the fact that the firmware is the same for both versions (see $\mathcal{H}_{7.3}$ in the analysis of the CardioMessenger II-S TLine), we looked at the code and got references to what seems to be the function parsing the SMS. Part of its code is presented in Figure 4.32.

It might be possible to fuzz the HMU with SMS to make it crash.. This is however out of our scope and has not been tried in this thesis. SMS fuzzing has, however, already been presented by Gorenc and Molinyawe, at Defcon 22 for example [21].

Finding $\mathcal{F}_{14}$: SMS are processed by the HMU and could be used as an attack vector.

```
void ProcessReceivedSMS(int iParm1,undefined4 uParm2,undefined4 uParm3,undefined4 uParm4)

{
  undefined *puVar1;
  undefined *puVar2;
  int iVar3;
  undefined4 uVar4;
  int iVar5;

  puVar2 = PTR_DAT_2001fac4;
  iVar5 = iParm1 * 0x90;
  iVar3 = FUN_2002bb20(4,*(undefined4 *)(PTR_DAT_2001fac4 + iVar5),PTR_DAT_2001fac4 + iVar5 + 4,
                       uParm4,uParm2,uParm3,uParm4);
  puVar1 = PTR_DAT_2001fa88;
  if (iVar3 == 0) {
    uVar4 = FUN_2002f0fa();
    FUN_2002f474(PTR_LAB_2001f78a+1_2001fad0,iParm1,*(undefined4 *)PTR_DAT_2001faa8,DAT_2001fac0,
                 DAT_2001facc,uVar4);
  }
  else {
    if (*(int *)(puVar2 + iVar5) == iVar3) {
      *(undefined4 *)(puVar2 + iVar5) = 0;
      if (iParm1 == 0) {
        if (**(char **)(puVar1 + 0xb0) != 0) {
          printLogWithParam(s_Incoming_SMS_processing_finished_2001fad4);
        }
        FUN_2002fb58(0x4b0);
      }
      else {
        uVar4 = FUN_2002f0fa();
        FUN_2002f474(PTR_LAB_2001f78a+1_2001fad0,iParm1 + -1,*(undefined4 *)PTR_DAT_2001faa8,
                     DAT_2001fac0,DAT_2001facc,uVar4);
      }
    }
    else {
      if (**(char **)(PTR_DAT_2001fa88 + 0xb0) != 0) {
        printLogWithParam(PTR_s_Error:_PdhmAddInputDataToTranspo_2001fac8,iVar3,
                          *(int *)(puVar2 + iVar5),iParm1);
      }
      *(undefined4 *)(puVar2 + iVar5) = 0;
      if (iParm1 == 0) {
        if (**(char **)(puVar1 + 0xb0) != 0) {
          printLogWithParam(s_Incoming_SMS_processing_finished_2001fad4);
        }
        FUN_2002fb58(0x4b0);
      }
```

Figure 4.32: Decompiled code of the function processing SMS

4.4 Summary of the findings on the CardioMessenger II-S

Tables 4.3 and 4.4 present the summary of our results regarding the security analysis of the CardioMessenger II-S in its Telephone Line and GSM versions. The findings are analyzed in two steps:

- Technical evaluation: do they have an impact on the CIANA criteria? Answered by yes or no.
- Ecosystem evaluation: what is the impact on the patient's safety and privacy? Answered by none, low, medium or high.

The evaluation is based on the fact that findings can be combined to launch an attack. Also, we considered that the "high" risk on the patient safety means that the patient could be injured or worse, killed.

Distinction is made between the two devices by the version column (V):

- T: CardioMessenger II-S TLine version
- G: CardioMessenger II-S GSM version
- X*: Finding that has not be verified by a proof of concept on that version but which is likely to be true

Findings			Impact						
ID	Description	V	C	I	A	N	A	Safety	Conf.
$\mathcal{F}_0$	An attacker can easily open the device and there is no obfuscation of the electronic components that would harden the identification of components.	B	×					None	None
$\mathcal{F}_1$	An attacker can identify the UART interface is even though there are no labels on the PCB. It is enabled and the Bootloader's banner is visible during the boot process. We did however not succeed in interacting with this UART interface directly.	B	×					None	None
$\mathcal{F}_2$	The JTAG interface is enabled and available to anyone who has some soldering skills, giving an attacker the full control over the system. That includes reading and writing the memory but also code execution.	T G*	×	×	×	×	×	High	High
$\mathcal{F}_3$	An attacker can gather APN's credentials by eavesdropping on the communication channel between the microcontroller and the modem. Those credentials are sent in cleartext and are still valid for at least some of the devices.	B	×				×	Low	High
$\mathcal{F}_4$	There is no mutual authentication between the microcontroller and the backend server, allowing an attacker to perform a MitM attack.	B	×	×		×	×	Medium	Medium
$\mathcal{F}_5$	The data is sent using telnet (or an equivalent program running over TCP) and the credentials are sent in cleartext over the communication channel allowing an attacker to eavesdrop the communication or to perform a MitM attack.	B	×	×		×	×	Low	Medium

Table 4.3: Findings summary (1)

Findings			Impact						
ID	Description	V	C	I	A	N	A	Safety	Conf.
$\mathcal{F}_6$	Credentials used to connect to the APN are reused to connect to the "telnet-like" service. An attacker can then get an access to the VPN and interact with the backend server using the same credentials.	B	×	×		×	×	Low	Low
$\mathcal{F}_7$	A proprietary protocol is used to send the data to the backend server. The HMU ID is available in the header, allowing an attacker to link a given packet to a given HMU (and with a given patient if she has the corresponding data).	B	×					None	Low
$\mathcal{F}_8$	The memory is unencrypted, allowing an attacker to access in cleartext credentials, debug strings and all the data that is sent to the backend server.	T G*	×				×	Medium	Medium
$\mathcal{F}_9$	The firmware is not encrypted nor obfuscated, easing the reverse engineering task for an attacker.	T G*	×	×		×	×	Medium	High
$\mathcal{F}_{10}$	The usage of debug strings, even if debug is not enabled, ease the reverse engineering of the communication protocol.	T G*	×	×		×		Low	Low
$\mathcal{F}_{11}$	The AES key is hard coded in the memory, allowing an attacker who accesses it to decrypt all outgoing communication from the HMU, and to forge her own packets.	T G*	×	×		×	×	High	High
$\mathcal{F}_{12}$	The PCB design of the CardioMessenger II-S is the same for both versions, exposing its circuit to an attacker who can then eavesdrop the communication between the microcontroller and the modem.	G	×	×				Low	Medium
$\mathcal{F}_{13}$	SMS are used as a mean of communication.	G	×					None	Low
$\mathcal{F}_{14}$	SMS are processed by the HMU and could be used as an attack vector.	G*			×			Low	None

Table 4.4: Findings summary (2)

4.5 Additional results

This section presents analysis results that were obtained but that are not included in our main scope, i.e. which do not concern directly the CardioMessenger II-S. The first subsection presents the analysis performed on the CardioMessenger 3G and that could constitute a base for further research. The second subsection presents the hardware analysis of the CardioMessenger LLT2 which was performed mainly to provide Lie with the PIN code of the SIM card [34]. That testing then helped her to confirm that the credentials to access the VPN are still valid.

4.5.1 Hardware analysis of the CardioMessenger 3G

The CardioMessenger 3G is the most recent HMU by Biotronik at the time of this writing. It uses the mobile network like the CardioMessenger II-S GSM. We present here the short hardware analysis we made before focusing on the CardioMessenger II-S, as it can be useful for further research.

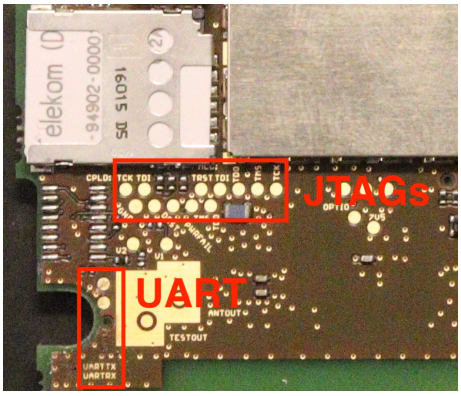


Figure 4.33: Labelled pins on the back of the CardioMessenger 3G

The first step, as for the other devices, was to test the UART interface. The CardioMessenger 3G is pretty hard to open without breaking it, but once opened the pins are labelled. As presented on Figure 4.33, the UART and JTAG interfaces are available on the back side of the device. However, even though the pins are labelled, testing the UART interface did not lead to any result. Not even the Bootloader message as it could have been expected when looking at the older devices.

Connecting to the JTAG interface requires soldering as on the CardioMessenger II-S and we did not try it ourselves to prevent breaking the board. It seems, however, that two interfaces are available. Given the fact that Biotronik seems to care about security and was already taking it into account when designing the older devices, it is not impossible that they have disabled those interfaces. That would explain

why they are labelled. However, these are suppositions and one need to solder to the pins to have a confirmation. A STM32 microcontroller is controlling that board, which means the firmware could be very similar to the one we gathered on the CardioMessenger II-S, opening the door to further research and attacks.

This board is featured with a micro-USB port, used to charge the device. However, it could be that the USB port is also an attack vector. That is why we tried to connect it to a host computer and see what kind of interaction is possible. The device is visible in the output of an `lsusb` command but no more interaction is possible. We then tried to use a FaceDancer[19] to emulate different USB devices. The FaceDancer is an external device, plugged between the computer and the board, which allows an attacker to emulate multiple type of device, like a keyboard, a usb key, etc. Those devices are usually accepted by a computer. None of the tests we performed with the FaceDancer were successful. We have not tried to fuzz the USB interface, as it could have broken the device and that was not the goal. Fuzzing is one of the Facedancer's main feature though.

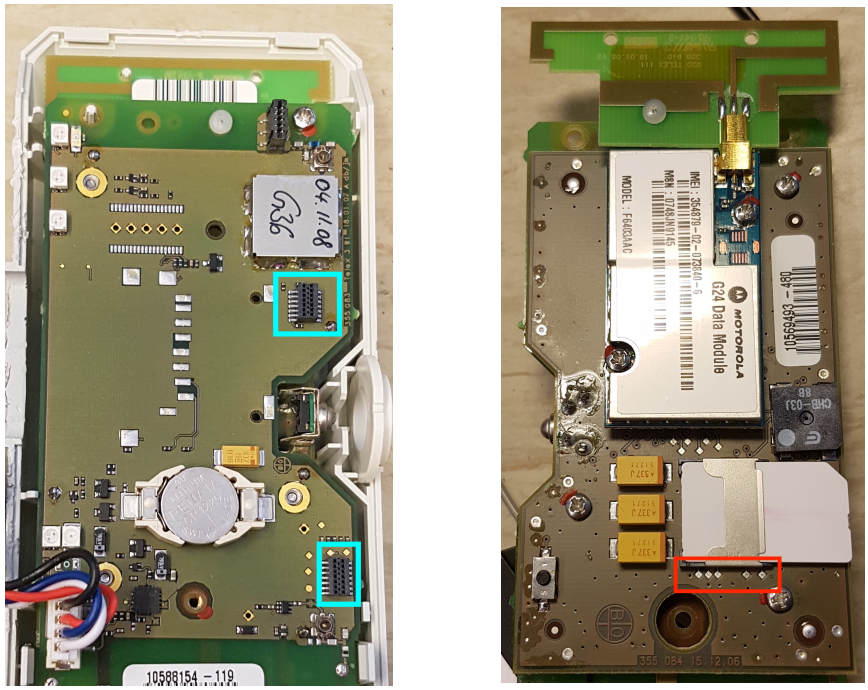
4.5.2 Hardware analysis of the CardioMessenger LLT2

Part of the work of Lie on her project was to analyze the kind of data that can be gathered through the mobile network [34]. To do so, having the PIN code of the SIM Card inside the HMU is useful. That is why, we were asked to reproduce on the LLT2, if possible, the eavesdropping of the communication channel between the microcontroller and the modem. Indeed, as one can see from Figure 4.34, there are plenty of testing points on the board. We then reproduced the testing to find a UART interface.

As a result, we located the UART interface which, similarly to the CardioMessenger II-S, displays the Bootloader's banner and that is all. No testings for the JTAG interface have been performed on this board, but the behavior being really close to the one observed on the CardioMessenger II-S, we believe that the JTAG interface could be available on one of the pin framed in cyan on Figure 4.34. Both of those connectors also expose the UART interface.

However, we did not find the communication channel with the modem exposed as it is likely to be inside the PCB. That is why we decided to look at another communication channel: the one between the modem and the SIM card. It appears that several testing points were sitting next to the SIM card's slot (in red in Figure 4.34). Probing them revealed serial communication. The Baud rate and the type of encoding was different than the standard one. The Baud rate was around 105 000 Bd/s and a parity bit was also used. The data received is not directly understandable.

[19] Available at https://int3.cc/products/facedancer21



(a) Front **(b)** Back

Figure 4.34: Inside of the CardioMessenger LLT2

It corresponds in fact to Application Protocol Data Unit (APDU). The details of the protocol are not discussed here as it is not part of the main results[20]. To extract relevant information from these APDU such as PIN code and phone numbers, we wrote a script that parses the packets and outputs a summary of the findings, as presented in Listing 4.23. The script itself is available in Appendix B.2.

Listing 4.23: Information's summary output by our APDU parser

```
** ANALYSIS RESULTS **

PIN found: [REDACTED]

5 german's phone numbers found in records:
+49[REDACTED]
+49[REDACTED]
+49[REDACTED]
+49[REDACTED]
```

[20]The interested reader can find information about that standard at https://cardwerk.com/smart-card-standard-iso7816-4-section-5-basic-organizations

```
+49[REDACTED]

7 german's phone numbers found in total in the hex dump:
/!\ This is raw data and can be misleading, be careful when
    dealing with it.
+49[REDACTED]
+49[REDACTED]
+49[REDACTED]
+49[REDACTED]
+49[REDACTED]
+49[REDACTED]
+49[REDACTED]

Plain texts found in the hex dump:
deen
[REDACTED]
[Cropped]
Special
SMS & MMS News
Downloads
Mail & Fax
Chat & Dating
[Cropped]
Eigene Rufnummer
```

4.6 Conclusion of the analysis

Several security issues leading to the compromise of the device have been discovered as a result of the security analysis we performed on both versions of the CardioMessenger II-S. This full compromise has been proved on the TLine version and is likely to be repeatable on the GSM version.

Assessing the impact of the findings on the patient's safety and privacy led us conclude that both these assets are impacted, answering our research questions **Q1** and **Q2**. Scenarios that can threaten patients' privacy and safety are described in Chapter 6.

When comparing Biotronik's HMU with other vendors' equivalent HMU such as the Merlin@Home, which is from 2008 as well, one can notice that Biotronik's system implements more security measures on the hardware level than St Jude Medical. However, when physical access is granted, both devices can be compromised without special skills.

Chapter 5
Mitigation

5.1 Mitigation of our findings

As a result of the security analysis conducted on the CardioMessenger II-S, several vulnerabilities were discovered, impacting both the patient's safety and privacy. Even if the vendor has implemented some security mechanisms, they failed to protect the physical layer which leads to the HMU being vulnerable to our attacks. It is not possible to mitigate most of the vulnerabilities as they lie in the hardware layer, which would require a new board design. That does not make sense due to the age of the device. The best mitigation is here to replace the CardioMessenger II-S still used in production with more recent versions which include better security measures. We shortly described security measures that could have been implemented to harden our task while analyzing the security of the device. Details on our recommendations and more advanced mitigation solutions are explained in the next Section.

5.1.1 Mitigating the physical tampering

Tampering mechanisms

No tampering mechanisms are implemented in the CardioMessenger II-S. Opening the device takes less than five minutes and does not require special tools nor to break anything. Adding tampering detection and tampering response mechanisms (as described in the next Section) would have hardened our task, as we did not have many devices at our disposal. Sacrificing one device was not an option. HMUs can be easily acquired using the internet, on eBay for example, but they are sold by individuals and not by companies. Consequently, the stock of a specific device is usually limited and response mechanisms which destroy the device could be a good measure to discourage "hobbyist" attackers with a low budget.

Obfuscation

Once opened, the components on the board are easily identified. Wiping out all information from the chips could constitute an additional difficulty for an attacker. The first step of an adversary is indeed to know what components are on the board and what interfaces to look for. In the case of the CardioMessenger II-S, knowing that the microcontroller is featured with JTAG led us to look for the corresponding testing points. Obfuscation should not be used as a standalone security practice, but more as an additional layer which goal is to harden the task to the attacker.

Hiding critical traces

An attacker should not have easy access to traces that can be used to gather information. In the case of the CardioMessenger II-S, the communication channel between the microcontroller and the modem can be eavesdropped using one of this trace, even on the GSM version due to PCB reuse. Hiding these traces inside the PCB is a good practice. This is, for instance, the case on the CardioMessenger 3G.

Disabling programming interfaces

The flaw that really resulted in the compromisation of the device was the discovery of the JTAG interface. The vendor should not let the JTAG interface available on production devices if possible. If not possible, the interface should at least be disabled. This can be done either on the software side or on the microcontroller after having flashed the device, and also on the circuit by using fuses to break the connection.

5.1.2 Mitigating the network emulation

Strong transport layer encryption

The HMU is sending its data directly over a TCP socket, which provides absolutely no security. Secure transport layer protocols such as SSL/TLS or Secure Shell (SSH) should be used instead, to prevent an attacker from eavesdropping the communication.

Mutual authentication

There is no mutual authentication between the microcontroller and the server, only the microcontroller authenticating itself to the server using a username and a password. If possible, one should use protocols featuring mutual authentication such as SSL/TLS or SSH, which allow authentication using both credentials and public key infrastructure.

No credentials reuse

Finally, the device is reusing the same credentials to connect to both the APN and to the backend service. That means that even though a solution such as SSH is implemented with credentials as the authentication method, an attacker can still connect to both the APN and to the backend service only by knowing the APN's credentials.

Hardening the backend infrastructure

The credentials gathered during the assessment are still valid and allow an attacker to access the APN, and thus the backend server. We truly recommend the vendor to harden the available servers on the backend infrastructure and also to implement a proper decommissioning procedure with a user management policy to prevent unused HMU that can be gathered through the internet to be used to access the main infrastructure and potentially exploit vulnerabilities on the server.

5.1.3 Mitigating the reverse engineering

Removing the debug function

One element that helped us start the reverse engineering process was the debug function included in the firmware and which seems disabled on production. However, the strings still being referenced, it gives an attacker a piece of information about the function he is looking at. This function should not be included in the commercial firmware.

Encrypting the firmware

The firmware is not encrypted which eases the reverse engineering process. We recommend to encrypt the firmware and to use a special flash memory with encryption and authentication features along with a secure way to store the cryptographic keys.

Storing the keys securely

In the case of the CardioMessenger II-S, not only the firmware is not encrypted, but the AES key is hard-coded in the memory and not even obfuscated. The vendor should use an additional chip, which goal would be to store the cryptographic keys used for secure booting and for other cryptographic functionalities.

5.2 Best practices

5.2.1 Embedded security pyramid

Even though embedded devices were not believed to be real targets by engineers, many attacks against IoT devices such as Mirai malware and its variants, have taken advantage of the lack of security in IoT devices to "overwhelm several high-profile targets with massive distributed Distributed Denial of Service (DDoS) attacks" [9, 32]. The IoT devices often being endpoints of the infrastructure, they constitute a potential attack vector and weak points in the whole architecture. Since those devices are mass produced, finding a vulnerability in one usually means finding a vulnerability in all the same devices. In addition, IoT devices are designed to last for several years and are not necessarily designed to be secure ten years from their production year, leading to breaches in infrastructures.

Securing embedded devices is challenging. One needs to address security at every level of the system and not only on the protocol layer. To help to address those problems in a structured manner, Hwang et al. [27] describe the "embedded security pyramid", which is reproduced in Figure 5.1. It is well known that security has a cost. Also, the bigger the scope to secure, the costlier it is. That is why the notion of "security partitioning" explained by Hwang et al. [27] is particularly interesting when designing an embedded device. They define "security partitioning" as an application of Kerckhoffs' principle: the security of a system must only rely on its secrets. They explain that the goal is to reduce the number of secrets in the system, without reducing the security of the device. If the device is "physically compromised, it remains secure as long as the secure module is intact". Reducing the area to protect also reduces costs.

The fact that an attacker can easily have physical access to the device is usually forgotten or ignored. Now, vendors cannot neglect one of the layers in IoT security without compromising the whole device. Indeed, the strongest security mechanisms at the upper layers (algorithm and protocol) are useless if an attacker has access to the data in cleartext in the flash memory. This problem is well highlighted in the security analysis of the CardioMessenger II-S: even though data are encrypted using AES, the key can be recovered from memory, allowing the attacker to decrypt all communications. Obviously, implementing security at every layer cannot be done at the end of the production. It is then highly recommended to include security in the development cycles of the product, thus the expression "security by design".

5.2.2 Security requirements for embedded devices

Listing all existing security measures in an exhaustive manner is neither possible nor the goal of this chapter. However, we describe in this Subsection some security

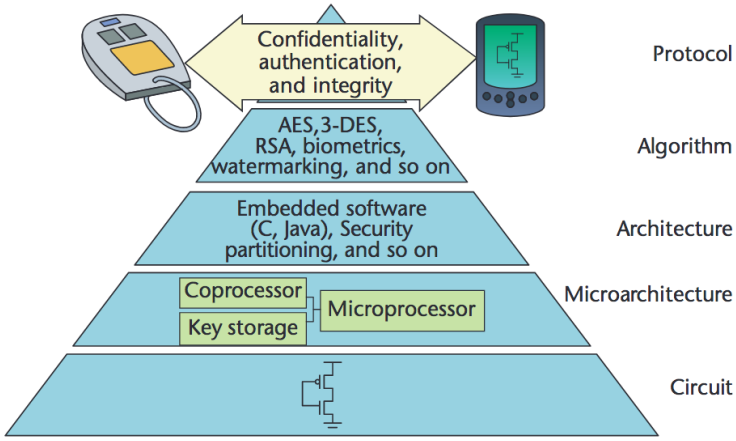


Figure 5.1: Embedded security pyramid [as depicted in [27]]

requirements for embedded devices to counter vulnerabilities and attack vectors described in Section 5.1.

Weingart [57] categorizes the attackers into three different classes. This classification is also used by Grand [22] in his guide for securing embedded devices, and consists of the following:

Class I A clever outsider, who has limited knowledge about the system and a low budget and equipment. This could be a curious attacker that is targeting the system mostly for prestige and as a hobby.

Class II A knowledgeable insider, who has advanced knowledge and/or specialized education and experience in the area. This category has access to sophisticated tools. Typically, this class corresponds to academics.

Class III A funded organization categorized by its high budget and its ability to recruit class II attackers to attack the system. This corresponds to organized crime or to a government.

In our thesis, we, as the attackers, would be classified as Class I attackers. The findings described in Chapter 4 do not require any advanced knowledge (class II attackers) nor extended funding (class III attackers), which is one of the biggest issues. The practices described below aim at preventing class I and discouraging class II attackers from attacking the device. Implementing enough security measures could also increase the cost of the attack and make the class III attackers choose

another attack vector. Indeed, it is unlikely that organized crime will try to attack a system if the costs are superior to the profits.

Physical level

On the physical level, the measure that can be implemented is a secure enclosure for the device. A device that can be opened with a basic screwdriver or a knife is not likely to discourage class I attacker [22]. Grand advises manufacturers to implement tampering solutions, some of those mechanisms being described in *Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defenses* [57]. The tampering mechanisms are classified into four categories:

- resistance: making the device more difficult to open, the circuit more difficult to understand. Protecting the circuit with epoxy[1] to prevent an attacker to tamper with them.
- evidence: having a proof of the tampering (by logging the event for instance), which can be useful for forensic analysis.
- detection: detecting the tampering (using a magnetic switch for example)
- response: reacting to an attempt of tampering by, for instance, destroying the device and its data.

Having several tampering mechanisms implemented is likely to discourage a class I attacker, as it will then require acquiring several devices in order to understand the mechanisms (which often involves destroying the device).

Access to the most targeted components on the board such as the microcontroller, the ROM, the RAM or the flash memory should be difficult. Using BGA packages and putting the chip where it cannot be desoldered and put in socket easily can prevent an attacker from having access to a rework station to do so [22]. As already highlighted in our result chapter, wiping out the information written on the components is not a security measure either. It can, however, slow down an attacker who tries to understand the overall system.

Circuit level

Regarding the circuit level, one should not expose programming or debugging interfaces on an operational device. Indeed, these interfaces can be of huge help to an attacker who has physical access to the device as highlighted by our analysis in

[1] Epoxy is some sort of resin.

Chapter 4. Grand also advises not to try to obfuscate the programming and debugging interface with some proprietary connectors, as they are likely to be understood by an attacker with basic probing tools.

Unnecessary test points should be removed from the final board and critical traces should be hidden in the inner layout of the PCB when possible. This is to prevent an attacker from eavesdropping directly the communication or to perform more advanced attacks such as differential power analysis which allows an attacker to recover cryptographic's keys [31]. Even though these types of attack seem to require advanced equipment, a presentation at Troopers 19 shows that Side Channel Attacks can be executed with very cheap equipment, available to class I attackers [51].

Among other functionalities, JTAG should not be included in the finished product if possible. Usually, one can disable JTAG on the microcontroller directly which would require the memory to be reset to enable it again. One can also disconnect the interface physically by using fuses. However, Grand [22] highlights that this is not enough as an attacker could repair the connection and then use JTAG again. Solutions to secure JTAG interface have been proposed by researchers and are presented by Vishwakarma and Lee [56]. Public Key Cryptography, challenge/response, credentials are propositions that are given by the author.

Microcontroller security

When it comes to securing the device at the level of the microcontroller, one should implement a secure boot (also called "trusted boot" or "verified boot") [16]. This requires the firmware to be run on the microcontroller to be signed by the manufacturer.

The method used to update the device should as well include a strong verification process, involving code signature. That can prevent an attacker from tampering with the new firmware or to replace the firmware by its own (if she is able to act as a Man in the Middle).

As explained by Grand [22], persistent memory should be encrypted. For that one can use (a)synchronous flash memory with authentication, password and encryption features. Encryption keys used for the secure boot but also the other cryptographic operations should not be stored in flash memory, nor appear in RAM. Indeed, it has been proven by Gutmann [23], that erasing data from RAM, and non-volatile memory in general, is complicated. Cryptographic chips such as the ATECC508A[2] can be used for that purpose.

[2]More details available at https://www.microchip.com/wwwproducts/en/ATECC508A.

In the case of medical devices, we highly recommend not to store any patient's data on the device in a persistent manner. An attacker having access to an old device will then not be able to recover any useful information about the patient.

Architecture level

Regarding the firmware running on the device, applying good practice can harden the reverse engineering task to an attacker who would have dumped a non-encrypted firmware. As explained by Grand [22], production's firmware should be compiled with:

- only the required features (no debug functions)
- no symbol tables and debug information (such as DWARF[3] for Executable and Linkable Format (ELF) files[4])
- compiler optimizations which harden the task of identifying common codes

Using source code auditing tools is also recommended by specialists. Open source tools like Lynis[5] can help harden a UNIX-based system.

Algorithm level

When it comes to cryptographic's algorithms, it is good practice not to implement the cryptographic functions from scratch and instead use well standardized and reviewed algorithms such AES or (Rivest Shamir Adleman (RSA) to cite only those two. Indeed, as well summarized by Schneier, "Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break."[6]

Embedded devices usually suffer from power and resource constraints which prevent the manufacturers from implementing strong cryptography mechanism such as Public Key Infrastructure (PKI). However, in the case of a monitoring device which is not using a battery, one can implement asymmetric cryptography. Also, the usage of integrity and authenticity mechanisms should be considered.

Protocol level

At the highest level, we do recommend the usage of a secure protocol such as SSH or SSL/TLS.[7] These protocols allows strong encryption along with authentication

[3] DWARF is a debugging format originally developed for the ELF.
[4] ELF is a file format for executables, defined in http://refspecs.linuxbase.org/elf/elf.pdf.
[5] Available at https://github.com/CISOfy/lynis.
[6] Full text available at https://www.schneier.com/crypto-gram/archives/1998/1015.html#cipherdesign.
[7] Secure Sockets Layer (SSL) and Transport Layer Security (TLS).

through different means. Asymmetric encryption can be used for both SSH and SSL/TLS, or a pair of username/password can also be used with SSH.

As a general recommendation, and for the same reasons developed in the "Algorithm level" section, one should not craft its own communication protocol, but rather use industry standard protocol. It is bad practice to have a proprietary protocol deal with the security, as it is most likely to be insecure against some attacks and also to be reverse engineered by the attacker. It is acceptable to use proprietary protocols over an already existing secure communication channel (established using SSH for instance).

Infrastructure level

Even though the infrastructure level is not included in the "embedded security pyramid" defined by Hwang et al. [27], we do believe it is important to take this level into considerations when designing an IoT infrastructure. In the situation of an end device being compromised, the whole infrastructure should include some securities and not leak information or allow the attacker to contact other devices as it has been shown by Borgaonkar et al. [12] in the study of femtocells' security.

Intrusion detection systems could be used along with monitoring of the devices to detect unusual activity. A device is only supposed to connect to a given number of ports on the server and any other attempt of connection should be suspicious and reported. That way, the system can also defend itself by blocking temporarily (or definitively) the suspicious device. Accounts management should also be implemented. In the case of medical devices, whose distribution is supposed to be monitored and regulated, the account of a patient who is not using the service anymore should not remain active. Otherwise, an attacker who acquires an old device (using eBay for instance) can gather valid credentials with potentially a huge impact on the vendor's systems if no protection is implemented on the server side.

Obfuscation

As already mentioned several times in this thesis, "security by obscurity" is a bad practice often believed as a good security measure for proprietary devices. This is definitively true. Obfuscation, however, should still be considered by manufacturers. It is essential to understand that securing an embedded device to be resistant to the three classes of attackers is most likely to be very costly and time-consuming, without having the absolute certainty that the device is secure against an attacker that has no time nor money limitation. That said, the goal of a manufacturer is to establish the risk along with a threat model to then design what level of security to implement. Should a connected temperature sensor be resistant to class I attackers? The response is most likely to be "yes". Should it be resistant to a government level

organization who has the ability to attack the device with state-of-the-art technics? Most likely not. Implementing obfuscation technics on top of other *real* security measures is, as explained by Grand [22], a good way to raise the difficulty of the attacks and to discourage most of the class I attackers.

5.2.3 Recommended guidelines

This subsection aims at providing the reader with more complete and detailed guidelines issued by companies and organizations.

GSMA The GSMA is an organization who represents the interest of mobile operators worldwide. Part of their work includes IoT security, and they provide detailed guidelines on their website.[8] Notably, best practices are given in the "GSMA IoT Security Guidelines" along with a set of resources to assess an IoT ecosystem in the "GSMA IoT Security Assessment". These guidelines and set of resources cover all the steps of the development of an IoT ecosystem, that is why they are included in this section.

OWASP The OWASP Foundation, famous for its "TOP 10" vulnerabilities' list also provides guidance on how to secure IoT environments.[9] We particularly recommend this guide as a beginning. It indeed provides three different top ten, targeting the manufacturers, the developers and the consumers. Each of these top ten provides the vulnerabilities from the most common to the least, which allows to prioritize the mitigation and/or the implementation of the security features.

Microsoft Microsoft provides security guideline for the implementation of an IoT architecture on Azure. Companies do not always have the skills neither the infrastructure required to use IoT and have to rely on a third party, such as Microsoft Azure. It is then important to follow the guidelines[10] Microsoft is providing them with, as they ensure a good configuration of the infrastructure along with a secure design of the whole infrastructure in general.

Manufacturers' guidelines Finally, when designing a product, it is also important to look for the guidelines written by the manufacturers of a given product. The security guidelines for a given microcontroller is an example. Often, microcontrollers and hardware chips are featured with several security functionalities which are not taken advantage of by the engineers.

[8] Available at https://www.gsma.com/iot/iot-security/iot-security-guidelines/.
[9] Available at https://www.owasp.org/index.php/IoT_Security_Guidance.
[10] Available at https://docs.microsoft.com/en-us/azure/iot-fundamentals/iot-security-architecture.

As a conclusion of this chapter, it is important when designing an embedded device to include security in the development cycles and to apply "security by design" to all abstraction layers, from the physical layer to the infrastructure one. A vulnerability in one of the layers can lead to the device being fully compromised and that scenario should be kept in mind by the engineers when defining what security measures will be included. When designing embedded medical devices, engineers must also consider the assets they are trying to secure, which are in most cases the patient's safety and privacy.

Chapter 6
Discussion

6.1 Implications of this work

6.1.1 Attack scenarios

As a result of the security analysis of the CardioMessenger II-S, several vulnerabilities have been found, both hardware and software. Alone, these vulnerabilities have each a small to medium impact on the ecosystem. When chained, however, the impact is more severe, and we established several scenarios based on our threat model that could threaten both the patient's privacy and safety. We assume in those scenarios that the attacker is organized crime whose goal is to blackmail a politician in order to influence some decisions, the politician or someone in her family having a pacemaker and an HMU.

In the first scenario, the goal of the attackers is to be able to perform extortion attacks, threaten the politician's life, and get hold of his personal data. In order to accomplish that, they could bribe the housekeeper or the concierge to get access to the politician's flat and thus to the HMU. They would then install a Raspberry Pi Zero in the HMU, connected to the JTAG ports. That part requires to solder four wires on very small connectors and two wires on the ground and the power. The device would then be closed again (the Raspberry Pi Zero is small enough to fit inside the device). Having configured the Raspberry Pi as an Access Point (using, for instance, `hostapd`) the attackers are then provided with a "remote physical address" to the HMU. The attackers are now in a position to do a MitM attack and have access to all the data that are going through the HMU along with the encryption key used to encrypt the data sent to the server. They also have the credentials to connect to the VPN and to the backend server, and can thus send forged packets to the service, tricking the doctor into thinking that there is a problem or on the contrary that everything is alright even though it is not. Indeed, although the HMU is not lifesaving equipment as explained by the vendor, it can still be used to detect malfunctions on already implemented pacemakers such as batteries' defects and

preventing that information to be reported could lead to the death of the patient [19].

The second scenario does not require to install a Raspberry Pi inside the HMU but still requires physical access for 5 to 10 minutes. In that amount of time, we think it is possible to open the device, connect to the JTAG interface (using a special connector crafted for that specific interface) and dump the memory. With the memory, the attackers are provided with the encryption key and can then decrypt all data that are sent to the backend server, breaking the privacy of the patient. That data can be accessed by installing a Fake Base Station near the device. The signal is more powerful than the one from the normal antenna so the device will connect to it and try to send its information. Indeed, there is no authentication of the backend server from the HMU, which means that it sends its data to a pre-configured IP as soon as it is provided with an IP itself.

Both those scenarios do not require a lot of expensive equipment. All it takes is a Raspberry Pi Zero, which costs around $10, and soldering skills. The hardest part of this scenario is in fact to get physical access to the HMU. However, this is definitively possible for the organized crime for instance.

6.1.2 Hypothetical attack scenarios

The scenarios that are described below have not been tested and uncertainty exists regarding their feasibility. However, we do think they are worth being reported to the vendor for it to verify its systems against those hypotheses. They could indeed have a huge impact.

During our research, we got access to the credentials used to connect to the VPN and to the backend services, which still seem valid. We have been able to contact the backend server with one of the HMU but have not tried to interact directly using a computer, as it goes outside our scope and we do not have the authorization from the vendor. Our hypothesis is however, that it should be possible by using one of the valid SIM cards at our disposal. It would then be possible for an attacker to get an IP address inside the vendor's VPN and to interact with the server that lies there. These servers not being exposed on the internet, we do not know what level of security is implemented. If for instance the patient's data are stored unencrypted on the "collect server", that could lead to a huge data breach if an attacker were to get access to it. That is why we do recommend the vendor to assess the security of that network and to harden the servers that are in that network. Indeed, the vulnerabilities leaking the credentials seem challenging to fix by a software update which means that several credentials will remain available.

Given the research about automotive and femtocell's security described in Chapter 2, the hypothesis of a remote attack using the Mobile Broadband should be

considered. This kind of attack can impact not only one specific patient but the patients in a whole geographic area. Our study showed that information can be gathered and attacks performed with physical access to the device. If code execution is possible using a flaw in the modem or using the mobile network, that could motivate organized crime or worse, terrorist organizations to exploit that attack vector and then reproduce our findings remotely.

Another scenario that could be possible is on the communication link with the pacemaker. We have not conducted research on this interface but now that we have the firmware, it seems possible to reverse engineer the code connecting to the pacemaker. Assessing that link for insecure features is also important, as it has already been demonstrated in other studies that performing a battery-draining attack on this interface might be possible [10, 11]. Such attacks use the HMU to constantly interrogate the pacemaker in order to make it perform cryptographic and communication operations which are battery-consuming. Researchers were able to drain the battery of a pacemaker "at a rate of approximately three percent per 24-hour period" using a Merlin@Home device [10]. At that rate, they calculated that draining the full battery of the pacemaker would take three months.[1] A good practice would be for the pacemaker to establish the communication. However, Rios and Butts [46] demonstrated that the HMU is establishing the communication for all the four vendors studied, vendor four being Biotronik.

6.1.3 Ethical considerations

One could argue that the probability of such an attack vector being chosen to attack someone's privacy and safety is very low. However, this threat is taken really seriously by high-valuable targets such as the former Vice President of the United States [43], which led us to think that the zero risk does not exist. That is why the main reason why we have chosen to wait with disclosing our findings. Instead, we will contact the vendor as explained in Chapter 3 and follow a coordinated vulnerability disclosure process.

In addition, many of the vulnerabilities found are due to PCB design or hardware problem in general which means that they are unlikely to be fixed by the vendor. Furthermore, this device not being the most recent one, it might not be worth investing in mitigation of our results. We do hope however, that disclosing our findings to the vendor will help them increase the security of their future devices.

[1] Assuming that the patient is sleeping eight hours per night.

6.2 Remaining problems in the IoT and Medical Devices

Security in the IoT area has been a hot topic for several years now. But it is clear that the situation is not really evolving. We had the opportunity to attend to the *Troopers* conference in Heidelberg, Germany which had a "Breaking the IoT" track, along with a round table about the IoT and medical devices' security. We also attended conferences at *Sikkerhet og Sårbarhet* in Trondheim where concerns about that topic were also raised. In this section, we try to summarize the main problems existing in the IoT area and more specifically in the medical devices based on our experience working on that topic for the past year and on the different contents proposed at the conferences.

6.2.1 Best practices are not applied

All the participants of the round table entitled "IoT / Medical Device Security" agreed on the fact that best practices are not applied in that area. The observations made by several researchers in the Automotive and IoT areas show that vulnerabilities found in devices today are the same as ten years ago. Different reasons were raised to explain that phenomenon:

- A lack of awareness from both the manufacturers and the customers
- A lack of IT security knowledge from manufacturers
- Negligence from the manufacturers
- A requirement from the market to quickly develop a product
- A lack of regulation

Another interesting point raised is that medical devices' users do not have the choice. Indeed, they often rely on their medical devices for their survival and thus cannot refuse to use them. That means manufacturers can impose higher prices as well as poor security as they have the monopoly on medical devices interacting with their implants. Security could, in fact, be a justification for the price of the products, but severe security flaws are still discovered nowadays.

In the case of Biotronik, our assessment is that it is indeed better at implementing security than other manufacturers in some aspects. On one side, they do have encryption implemented and use a VPN to connect to their backend server. On the other side, they seem to have underestimated the physical security and send credentials in clear text over different communication channels. Also, attacks exploiting the JTAG interface are not new and were already presented at BlackHat Europe 2006

by Barnaby Jack [28]. But the CardioMessenger II-S being from 2008, it is not surprising that we found that vulnerability.

6.2.2 The medical security trade-offs

Developing secure IoT and medical devices is hard. Multiple challenges have to be overcome, both technically and ethically. Indeed, as explained by Zheng et al., developing a secure embedded medical device is a real technical challenge and three main trade-offs exist [59]:

Security vs. Accessibility This is maybe one of the most complex challenges to overcome when designing an IMD: creating a device that is both secure and accessible in case of emergency. Indeed, in case the patient needs surgery or any other emergency treatment that could require to disable the implant, the doctors should be able to do so, even in a hospital on the other side of the earth. Ensuring security while having a "backdoor" access is however quite counter-intuitive.

Emergency vs. Checkup Patients sometimes require checkup when having some implants (such as a pacemaker or an insulin pump). The challenge here is to implement security measures for the doctor to access the device in a secure way (not in an emergency context) but without consuming too much power, as the implants have limited battery and strong authentication mechanisms could shorten their life significantly.

Security vs. Resources Similarly to the second trade-off, security schemes implemented in an implant have to be "light" in terms of battery consumption which often means weak security. That is also why it is challenging to conceive a monitoring unit for such a device, as sending the data is already power consuming. Sending them with authentication, encryption, etc. is even more.

In addition, when dealing with medical devices security, and security in the medical area in general, one has to face ethical issues. Indeed, as emphasized by Beau Woods during his talk at *Sikkerhet og Sårbarhet*, one has to consider the assets that are really at stake when securing an IT environment. In a medical environment, it is often the patients' safety. One striking example he gave was about implementing a strong password policy on the computers that are used in an emergency service. He said: "The time a doctor is spending resetting his password, he is not spending it saving lives." According to him, that is where the difference between financial companies and critical industries stands: "companies can deal with a financial loss, but what about humans' lives?" This dilemma is the case in the medical devices' industry but also in the automotive industry.

In a nutshell, designing devices that could put humans' lives at stake is not an easy task. While the question of which medical device is the most secure should not be asked, it is a utopia to believe that all manufacturers are putting security at their top first priority. We can, however, hope that manufacturers start to be more aware of the security problems that exist, and start include security in the design cycle of their products.

6.2.3 Scoring systems

Even though we are not using any of the two scoring systems described below, we do believe that assessing a medical device using a defined and standardized scoring systems is important. This subject still being a discussion we have not included such a rating in our thesis. However, a more complete assessment of our findings involving interviews along with an adapted scoring system is left for future work as explained in Section 6.3. That justifies the discussion of the scoring systems' problem in this section.

When assessing the security of their IT systems, companies need a way to know the severity of the vulnerabilities. That way, they can remediate quickly to the most critical problems. The solution is to use a scoring system that allows the comparison between the vulnerabilities based on several criteria. One of the most commonly used scoring systems is the Common Vulnerability Scoring System (CVSS). It evaluates the severity based on several metrics and gives a score to the vulnerability ranging from 0 to 10. Metrics include impact, environment or temporality [38].

Manufacturers are required to assess the risk associated with their devices during the development process. However, CVSS does not fit well to assess the security of medical devices. As explained by Penny Chase and Steve Christey Coley [42], CVSS was "developed for enterprise information technology systems and do not adequately reflect the clinical environment and potential patient safety impacts." According to the same authors, a scoring system suited for medical devices should be of "minimal complexity", "usable by practitioners", "flexible" and also be accepted by the different actors including the "manufacturers, the hospitals, the researchers, the patients and the regulators."

Designing such a scoring system is a challenging task given the fact that all the actors have a different perspective on the subject. However, they do all agree on the fact that priority should be given to the patients' safety and security [15]. Penny Chase and Steve Christey Coley proposed a rubric that guides CVSS users into assessing medical devices [42]. Their idea is there to keep using CVSS but to adapt it for medical devices. The rubric provides for each metric of the CVSS, decisions trees with questions and examples. Also, they explain that the exploitability and the impact should lead to two different scores, in order for the impact not to

hide the exploitability. Adapting CVSS is however not the only solutions that have been proposed. QED Secure Solutions and WhiteScope have come with a brand-new scoring system specifically designed for medical devices: Risk Scoring System for Medical Devices (RSS-MD). As explained by the creators of that system, it remains consistent with the objectives of the CVSS while providing the medical community with factors they care about such as a focus on the impact to the patient's safety [7]. The system differentiates two scores: the functional impact score and the vulnerability score. The functional impact aims at reflecting the impact of an exploited vulnerability on the medical devices capability to assure its primary function of taking care of the patient while the goal of the vulnerability characterization is more similar to the CVSS and gives more understanding of the vulnerability itself along with its exploitability.

However, these different solutions are still work in progress and we did not use them for assessing our findings.

6.2.4 Certifications

As explained in Chapter 1, medical devices are classified into several types by regulatory agencies in Europe and America. Devices that have a direct impact on the patients' safety require their hardware and software to be certified. This is a good measure.

However, when a bug or a security issue is reported to a vendor, mitigating that bug can induce changes in the software which then needs to be certified again. That certification process is both costly and time-consuming. Also, what happens if the mitigation and re-certification process last three months and that a new vulnerability is disclosed to the vendor every month or every two months as it can happen in the automotive industry? Manufacturers are then more likely to leave the bug unfixed if they estimate that there is no risk.

This is another situation where vendors are facing a dilemma. Indeed, the money and time that are invested in obtaining a new certification for a product are not invested in the creation of the next generation of saving life devices. Vendors have to answer questions such as "should we focus on saving the life of a few individuals or work on the next generation of devices which could save many more people?". This can be seen as a modern version of the Trolley problem introduced by Philippa Foot in 1967 and further discussed by Judith Thomson in 1976 [20, 54]. This problem raises the question of whether or not it is ethical to switch a lever to kill one person and save five that would have died otherwise. In other words, that raises the dilemma of "killing" versus "letting die". The exact same dilemma is now faced by the automotive industry when designing the autonomous driving systems. Should the car choose to save its passengers and kill a pedestrian, or on the contrary choose to let its

passengers die? Those questions are the ones manufacturers should have in mind when designing new devices and objects that can have a direct impact on humans' lives. It is however out of our scope to try to answer them.

6.3 Future work

Even though both Anniken Wium Lie's thesis and our thesis present new vulnerabilities in the pacemaker's ecosystem, there are still plenty of elements that need to be investigated. We try in this section to give some leads for future work on the Biotronik's pacemaker ecosystem.

We have described in the first section of this chapter two scenarios that could possibly threaten both the patients' privacy and safety. However, without having a working pacemaker connected to our HMU it was not possible to have a proof of concept of our claims. One of the further research that could be conducted could use a working pacemaker and our findings to study the exact impact of the vulnerabilities discovered. We do not know what kind of data is transmitted by the pacemaker to the HMU and what is the impact of faking this data for the practitioner. A more detailed risk analysis using the scoring systems developed currently along with interviews of the actors of the ecosystem could be performed.

As we got the firmware of the CardioMessenger II-S, more studies can be performed on it. Notably, one can take a look at the following points:

- the code in charge of receiving data from the server. We do not know what kind of data they are supposed to receive. It could be update related data or other data. Some flaw might exist in this code and could lead to remote exploitation of the device, as we already know that the data is indeed copied in RAM when sent to the device through that communication channel.

- the update mechanism. It is a common way for attackers to obtain the firmware and to then compromise the device. Apart from the receiving functions, we have not been able to confirm that such an "update over the air" system exist here.

- the way the SMS are handled. We know from the firmware source code that the HMU is parsing the received SMS. It could be as well an attack vector if vulnerabilities exist in those functions.

- the communication link with the pacemaker. The code to interact with the pacemaker can be reverse engineered to understand what is happening on that interface and what is doable. Testing which device between the pacemaker and

the HMU is in charge of establishing the communication can be an indication of whether or not it is possible to drain the battery of the pacemaker.

The CardioMessenger II-S is not the last device from the vendor and is thus not expected to be the most secure today. That is why focusing on the latest version of the HMU, the CardioMessenger 3G can be interesting. We have done very little work on that device. Soldering a connector to the JTAG interface is the way to go in our mind. One should, however, be careful as it is also complex soldering. Doing more investigation on the USB interface is also possible, notably fuzzing. However, without a way to monitor the microcontroller, it might not be really of any interest.

It is most likely that the vendor is not starting a firmware from scratch for each new device. Indeed, similarities between devices exist as we noticed with the Bootloader's messages on the UART. It could probably be useful to spend some time investigating the older HMU versions and to try to get their firmwares as it could reveal interesting findings for the most recent devices.

Finally, the last lead would be to work with the vendor's authorization to assess their cloud-based infrastructure and see what is indeed doable when VPN's credentials are gathered.

Chapter 7
Conclusion

Improving the security level of IoT devices, and more specifically of medical devices is still an ongoing research topic involving researchers, manufactures, doctors, regulations authorities and patients. Part of researchers' work involves assessing the security of existing devices. In our thesis, we focused on Biotronik's pacemaker ecosystem with one main hypothesis:

> *The Biotronik's HMUs contain security vulnerabilities that might help an attacker with physical access to the device to get patients' personal data or to be a threat to patients' safety.*

To answer this question, we defined a testing methodology based on the Black Box Testing Methodology, which we applied to the Biotronik's Home Monitoring Unit. As a result of our testing, we discovered several vulnerabilities that can impact both the patient's safety and privacy, thus confirming our main hypothesis. We provided mitigations that could be used to make the device more secure along with general guidelines for designing more secure embedded devices. Finally, we discussed the implications of our findings along with the remaining problems in the IoT area.

Overall, our results demonstrate that the Biotronik's CardioMessenger II-S contains vulnerabilities that can have a substancial impact on the patients and on the ecosystem in general. Our work opens the door to further research on the CardioMessenger II-S but also on the latest HMUs of Biotronik.

References

[1] Atmel AT91RM9200 Datasheet. Datasheet, Atmel. URL https://www.microchip.com/wwwproducts/en/AT91RM9200.

[2] The Point-to-Point Protocol (PPP). RFC, Internet Engineering Task Force, July 1994. URL https://tools.ietf.org/html/rfc1661.

[3] Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange. Standard, Telecommunications Industry Association, October 1997.

[4] IEEE Standard Test Access Port and Boundary-Scan Architecture. IEEE Standard, IEEE Computer Society, February 2013. URL https://standards.ieee.org/standard/1149_1-2013.html.

[5] *Collaborative Approaches for Medical Device and Healthcare Cybersecurity*. US Food and Drug Administration, 2014. URL https://www.fda.gov/downloads/MedicalDevices/NewsEvents/WorkshopsConferences/UCM419427.pdf.

[6] *Medical Device Cybersecurity - Regional Incident Preparedness and Response Playbook*. MITRE, 2018.

[7] Risk Scoring System For Medical Devices RSS-MD. Technical report, QED Secure Solutions and WhiteScope, 2018. URL https://www.riskscoringsystem.com/medical/techspecmedical.pdf.

[8] Lawrence K. Altman. Arne h. w. larsson, 86; had first internal pacemaker. *The New York Times*. URL https://www.nytimes.com/2002/01/18/world/arne-h-w-larsson-86-had-first-internal-pacemaker.html.

[9] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.

[10] Carson C. Block. Muddy waters report - st. jude medical, inc. Technical report, Muddy Waters Capital LLC, August 2016. URL http://www.muddywatersresearch.com/research/stj/mw-is-short-stj/.

[11] Carson C. Block. Muddy waters report - st. jude medical, inc. Technical report, Muddy Waters Capital LLC, August 2016. URL http://www.muddywatersresearch.com/research/stj/mw-is-short-stj/.

[12] Ravishankar Borgaonkar, Kevin Redon, and Jean-Pierre Seifert. Security analysis of a femtocell device. In *Proceedings of the 4th International Conference on Security of Information and Networks*, SIN '11, pages 95–102, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1020-8. doi: 10.1145/2070425.2070442. URL http://doi.acm.org/10.1145/2070425.2070442.

[13] Carmen Camara, Pedro Peris-Lopez, and Juan E. Tapiador. Security and privacy issues in implantable medical devices: A comprehensive survey. *Journal of Biomedical Informatics*, 55:272 – 289, 2015. ISSN 1532-0464. doi: https://doi.org/10.1016/j.jbi.2015.04.007. URL http://www.sciencedirect.com/science/article/pii/S153204641500074X.

[14] James Carlson. *PPP design, implementation, and debugging*. Addison-Wesley Reading, 2000.

[15] Penny Chase and Steve Christey Coley. Using cvss in medical device security risk assessment. 2017. URL https://www.fda.gov/media/105995/download.

[16] Derek L Davis. Secure boot, August 10 1999. US Patent 5,937,063.

[17] European Commision. Medical Devices - Regulatory Framework, 2017. URL https://ec.europa.eu/growth/sectors/medical-devices/regulatory-framework_en.

[18] European Parliament and Council. REGULATION (EU) 2017/745 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 5 April 2017 on medical devices, amending Directive 2001/83/EC, Regulation (EC) No 178/2002 and Regulation (EC) No 1223/2009 and repealing Council Directives 90/385/EEC and 93/42/EEC, 2017. URL https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32017R0745&from=EN.

[19] US Food and Drug Administration. Firmware update to address cybersecurity vulnerabilities identified in abbott's (formerly st. jude medical's) implantable cardiac pacemakers: Fda safety communication, August 2017. URL https://www.fda.gov/MedicalDevices/Safety/AlertsandNotices/ucm573669.htm.

[20] Philippa Foot. The problem of abortion and the doctrine of double effect. 1967.

[21] Brian Gorenc and Matt Molinyawe. Blowing up the Celly! Building Your Own SMS/MMS Fuzzer, 2014. URL https://www.defcon.org/images/defcon-22/dc-22-presentations/Gorenc-Molinyawe/DEFCON-22-Brian-Gorenc-Matt-Molinyawe-Blowing-Up-The-Celly.pdf.

[22] Joe Grand. Practical secure hardware design for embedded systems. In *Proceedings of the 2004 embedded systems conference*, volume 23, 2004.

[23] Peter Gutmann. Data remanence in semiconductor devices. In *USENIX Security Symposium*, pages 39–54, 2001.

[24] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 129–142, May 2008. doi: 10.1109/SP.2008.31.

[25] Pete Herzog. Open source security testing methodology manual. *ISECOM, USA*, 2010.

[26] Hjerteforum. Norsk pacemaker og icd statistikk for 2017, 2018. URL https://legeforeningen.no/Fagmed/Norsk-cardiologisk-selskap/Hjerteforum1/b-2018/Hjerteforum-nr-2-2018/Artikler/Norsk-pacemaker--og-ICD-statistikk-for-2017/.

[27] David D Hwang, Patrick Schaumont, Kris Tiri, and Ingrid Verbauwhede. Securing embedded systems. *IEEE Security & Privacy*, (2):40–49, 2006.

[28] Barnaby Jack. Exploiting embedded systems. Black Hat EU, 2006. URL https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Jack.pdf.

[29] Randy Johnson and Stewart Christie. JTAG 101 - IEEE 1149.x and Software Debug. White Paper, Intel Corporation, January 2009.

[30] Richard L. Kissel. Glossary of key information security terms. http://nist.gov/, Published on 05-06-2013. URL https://www.nist.gov/publications/glossary-key-information-security-terms-1.

[31] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[32] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[33] Eivind S. Kristiansen and Anders B. Wilhelmsen. Security Testing of the Pacemaker Ecosystem. Master's thesis, NTNU Trondheim, 2018.

[34] Anniken Wium Lie. Wireless Security analysis of the Home Monitoring Unit in the Pacemaker Ecosystem. Master's thesis, NTNU Trondheim, 2019.

[35] ARM LTD. Arm920t technical reference manual. *ARM Ltd.*

[36] Eduard Marin. *Security and Privacy of Implantable Medical Devices*. PhD thesis, KU Leuven, 2018. Bart Preneel and Dave Singelée (promotors).

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

[38] Peter Mell, Karen Scarfone, and Sasha Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6):85–89, 2006.

[39] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015:91, 2015.

[40] National Heart, Lung, and Blood Institute. Arrhythmia. URL https://www.nhlbi.nih.gov/health-topics/arrhythmia.

[41] UK Parliament. Cyber-attack on the nhs, March 2018. URL https://publications.parliament.uk/pa/cm201719/cmselect/cmpubacc/787/78702.htm.

[42] Penny Chase and Steve Christey Coley. Rubric for Applying CVSS to Medical Devices. Technical report, MITRE, 2017. URL https://www.mitre.org/sites/default/files/publications/pr-18-2208-CVSS-medical-device-rubric-v0.11.04.pdf.

[43] Andrea Peterson. Yes, terrorists could have hacked dick cheney's heart, October 2013. URL https://www.washingtonpost.com/news/the-switch/wp/2013/10/21/yes-terrorists-could-have-hacked-dick-cheneys-heart/?utm_term=.c78a265d407e.

[44] Ransdell Pierson, Jim Finkl. St. Jude sues short-seller over heart device allegations. URL https://www.cnbc.com/2016/09/07/st-jude-sues-short-seller-over-heart-device-allegations.html.

[45] Dominic Rath. *OpenOCD: Open on-chip debugging*. PhD thesis, Diploma Thesis, FH Augsburg, 2005.

[46] Billy Rios and Jonathan Butts. Security evaluation of the implantable cardiac device ecosystem architecture and implementation interdependencies, 2017.

[47] Martin Sauter. *From GSM to LTE: an introduction to mobile networks and mobile broadband*. John Wiley & Sons, 2010.

[48] A. Shostack. *Threat Modeling: Designing for Security*. Wiley, 2014. ISBN 9781118810057. URL https://books.google.no/books?id=YiHcAgAAQBAJ.

[49] Zheng Huang Shupeng Gao, Haiku Xie and Zhang Ye. All the 4G Modules Could be Hacked. Black Hat USA, 2019. URL https://www.blackhat.com/us-19/briefings/schedule/#all-the-4g-modules-could-be-hacked-16187.

[50] Nasjonal sikkerhetsmyndighet. Yes, terrorists could have hacked dick cheney's heart, 2018. URL https://www.nsm.stat.no/aktuelt/datainnbrudd-helse-sor-ost/.

[51] Albert Spruyt and Alyssa Milburn. Hardware side channel attacks.. on the cheapiest! 2019. URL https://troopers.de/downloads/troopers19/TROOPERS19_NGI_RT_Hardware_Side_Channels.pdf.

[52] Andrew Sun. *Using and Managing PPP*. O'Reilly Media, Inc., 1999.

[53] Texas Heart Institute. Implantable Cardioverter Defibrillator (ICD). URL https://www.texasheart.org/heart-health/heart-information-center/topics/implantable-cardioverter-defibrillator-icd/.

[54] Judith Jarvis Thomson. Killing, letting die, and the trolley problem. *The Monist*, 59(2):204–217, 1976.

[55] US Food and Drug Administration. Medical device overview, 2018. URL https://www.fda.gov/industry/regulated-products/medical-device-overview#What%20is%20a%20medical%20device.

[56] Gopal Vishwakarma and Wonjun Lee. Exploiting jtag and its mitigation in iot: A survey. *Future Internet*, 10(12):121, 2018.

[57] Steve H. Weingart. Physical security devices for computer subsystems: A survey of attacks and defenses. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 302–317. Springer, 2000.

[58] Roelf J. Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[59] G. Zheng, G. Zhang, W. Yang, C. Valli, R. Shankaran, and M. A. Orgun. From wannacry to wannadie: Security trade-offs and design for implantable medical devices. In *2017 17th International Symposium on Communications and Information Technologies (ISCIT)*, pages 1–5, Sept 2017. doi: 10.1109/ISCIT.2017.8261228.

Appendix

Methodologies and procedures

A.1 Discovering the JTAG interface

A.1.1 Updating the JTAGulator firmware

JTAGulator being a tool under development, new functionalities and bug fixes are added on each new release of its firmware. The firmware used during our research is the version 1.6. It has been upgraded on a Mac OS system by following the procedure described below.[1] The procedure on Windows systems is slightly different as it takes advantage of Windows-only tools.

1. Download the `propellergcc` that corresponds to the OS from https://code.google.com/archive/p/propgcc/downloads
2. Download the desired version of the JTAGulator from https://github.com/grandideastudio/jtagulator/releases
3. Extract both archives
4. Connect the Board to your computer
5. Execute the following command (ensure that there is no session on the board):

```
$ ./parrallax/bin/propeller-load \
        -e jtagulator-1.6/JTAGulator.eeprom
```

6. The result should be similar to:

```
Propeller Version 1 on /dev/cu.usbserial-A601YNUN
Loading .../jtagulator-1.6/JTAGulator.eeprom to EEPROM via
    hubmemory
```

[1] This procedure is based on the documentation available at https://sites.google.com/site/propellergcc/documentation/tutorials/load-propgcc-code-images

```
32768 bytes sent
Verifying RAM ... OK
Programming EEPROM ... OK
Verifying EEPROM ... OK
```

7. Disconnect the board

Once the board is reconnected, the new firmware should have been flashed correctly. One can confirm that by issuing the "I" command on the firmware 1.6.

```
 JJJ  TTTTTTT AAAAA  GGGGG           UUU LLL    AAAA    TTTTTT    OOOOO   RRRRRR
 JJJJ TTTTTTT AAAAAA GGGGGGG        UUUU LLL   AAAAAA TTTTTTTT OOOOOOO  RRRRRRRR
 JJJJ  TTTT  AAAAAAA GGG       UUU  UUUU LLL   AAA AAA    TTT   OOOO OOO  RRR RRR
 JJJJ  TTTT  AAA AAA GGG  GGG UUUU UUUU LLL AAA  AAA    TTT   OOO  OOO  RRRRRRR
 JJJJ  TTTT  AAA  AA GGGGGGGGG UUUUUUUU LLLLLLLL AAAA   TTT OOOOOOOOO  RRR RRR
  JJJ  TTTT AAA   AA GGGGGGGGG UUUUUUUU LLLLLLLLL AAA   TTT OOOOOOOOO  RRR RRR
  JJJ  TT                GGG              AAA                        RR RRR
 JJJ                      GG              AA                             RRR
JJJ                       G              A                                RR


          Welcome to JTAGulator. Press 'H' for available commands.
        Warning: Use of this tool may affect target system behavior!
```

A.2 Interacting with a JTAG interface

To interact with a JTAG interface, proprietary connectors are usually used (as the J-LINK adapters from Segger). However those connectors can be expensive. In this thesis we tried to performed attacks with the lowest cost possible. That is why, even though it could be more practical to use a proprietary interface, we present here a way to interact with a JTAG interface using OpenOCD along with the Shikra and a Raspberry Pi Zero W.

A.2.1 Using the Shikra as the interface

The Shikra from Xipiter is already the tool we used when we wanted to interact with RS-232 or more generally to UART as it as revealed itself as being very stable compared to regular USB to TTL connectors. OpenOCD does not include any specific[2] configuration file for the Shikra but Xipiter is giving one on their website:

```
#shikra.cfg
interface ftdi
ftdi_vid_pid 0x0403 0x6014

ftdi_layout_init 0x0c08 0x0f1b
```

[2]The configuration file for the Shikra could actually be deducted form the one of the JTAGkey for instance, as the chip used on the board is the same.

```
adapter_khz 1000
#end shikra.cfg
```

One need to create a basic configuration file “config.cfg” with the following content to interface with the microcontroller:

```
jtag newtap at91 tap -irlen 8 -expected-id 0x05B0203F
adapter_khz 1000
transport select jtag
```

The connection can then be checked using the `scan_chain` command.

```
> scan_chain
scan_chain
  TapName         Enabled IdCode     Expected   IrLen IrCap IrMask
- -------------- ------- ---------- ---------- ----- ----- ------
0 at91rm9200.cpu    Y    0x05b0203f 0x05b0203f     4 0x01  0x0f
```

A.2.2 Using a Raspberry Pi as the interface

An even cheaper option than the Shikra is to use a Raspberry Pi Zero as an interface. Indeed, those boards cost $10 from official resellers and have everything needed to act as a JTAG interface. Some project exist as well to use Raspberry board to perform JTAG enumeration.[3]

As explained above, using OpenOCD is not always straight forward and having a functional configuration can take several trials and errors before it does what one can expect.

Establishing a basic connection

The installation and configuration process from a Raspberry Pi Zero that is already a Linux distribution installed (Raspbian GNU/Linux 9 (stretch) in our case) is the following:

1. Connect to the Raspberry Pi (using SSH, a serial connection or other).
2. Execute the following commands line to build OpenOCD from sources and including the required dependencies for the Raspberry Pi. An error regarding the lack of “make info” might be raised but it is not an issue.

[3] JTAGenum is available at https://github.com/cyphunk/JTAGenum

```
$ 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
```

3. Connect the pins according Table A.1.

JTAG Pin	Raspberry Pi Pin[4]
TCK	23
TDI	19
TDO	21
TTMS	22
TRST (opt)	24

Table A.1: Raspberry Pi's JTAG connection using OpenOCD

4. Issue the following command to copy the configuration file of the Raspberry Pi

```
$ cp /path/to/openocd/scripts/interface/raspberrypi-native.
    cfg rpi.cfg
```

5. Reuse the same configuration file created for the Shikra ("config.cfg").

6. Issue the following commands in two different sessions (or on another terminal if you decide to allow remote access):

```
$ sudo openocd -f rpi.cfg -f config.cfg.        # terminal 1
$ nc 127.0.0.1 4444                             # terminal 2
```

7. The connection can be checked using the `scan_chain` command that should return the following:

```
> scan_chain
scan_chain
  TapName         Enabled  IdCode     Expected   IrLen IrCap IrMask
- -------------- ------- ---------- ---------- ----- ----- ------
0 at91rm9200.cpu    Y    0x05b0203f 0x05b0203f     4 0x01  0x0f
```

[4]The pins correspond to the GPIO's pins of the Raspberry and not to the microcontroller's pins.

Appendix B
Tools developed

B.1 Serial line communication scripts

B.1.1 Extended monitoring script for serial line communication

Listing B.1: CardioMessenger Serial Monitor Script

```
#!/usr/bin/env python3

"""
    File name: cm-serial-monitor.py
    Version: 1.0
    Author: Guillaume Bour
    Last modified: 2018/05/29
    License: GNU General Public License v3.0
    Description: A script to monitor activity on a UART channel.
        This script logs the received both in raw and decoded format.
"""


import serial
import os
import sys
import time
import datetime
import signal


DISPLAY_NEW_LINE = False
VERSION = 1.0
TIMESTAMP_FORMAT = "%Y-%m-%d %H:%M:%S"
LOGGING = True
WAITING_TIME = 0.04 # 0.04 for CM as response time is set to 30ms

BASIC_FILE_NAME = datetime.datetime.now().strftime("%Y%m%d_%H%M%S.txt")
RAW_FILE_NAME = datetime.datetime.now().strftime("%Y%m%d_%H%M%S_raw.txt
    ")

```

```
31 file_basic = None
32 file_raw = None
33
34
35 def usage():
36     print("Usage: python3 cm-serial-monitor.py SERIAL_PORT BAUDRATE")
37
38
39 def main():
40     if len(sys.argv) < 3:
41         usage()
42         sys.exit(0)
43
44     S_PORT = sys.argv[1]
45     B_RATE = sys.argv[2]
46
47     print("*** CM Serial Monitor v{} ***".format(VERSION))
48     print("[+] Port: {}".format(S_PORT))
49     print("[+] Baudrate: {}".format(B_RATE))
50     print("[+] Display new line: {}".format("True" if DISPLAY_NEW_LINE
           else "False"))
51     print("[+] Current directory is {}".format(os.getcwd()))
52
53     with serial.Serial(S_PORT, B_RATE, timeout=1) as ser:
54         signal.signal(signal.SIGINT, signal_handler)
55
56         file_basic = open(BASIC_FILE_NAME, "a+")
57         file_raw = open(RAW_FILE_NAME, "wb")
58
59         print("[*] Logging raw input in {}".format(RAW_FILE_NAME))
60         print("[*] Logging decoded input in {}".format(BASIC_FILE_NAME)
               )
61         print("[*] Starting serial communication, press CTRL+C to
               terminate...\n")
62
63         while True:
64             raw_line = ser.readline()
65             currentDT = datetime.datetime.now()
66
67             timestamp = datetime.datetime.now().strftime(
                   TIMESTAMP_FORMAT)
68
69             if raw_line:
70                 line = ""
71                 try:
72                     line = raw_line.decode("ascii")
73                 except UnicodeDecodeError:
74                     line = "Decoding error"
75
76                 final_line = "[{}] {}".format(timestamp, line)
77
78                 file_raw.write(raw_line)
```

```
79                  file_basic.write(final_line)
80                  file_raw.flush()
81                  file_basic.flush()
82
83                  if DISPLAY_NEW_LINE:
84                      final_line = final_line.replace("\r","/r").replace(
                            "\n", "/n")
85                  else:
86                      final_line = final_line.replace("\r", "").replace("
                            \n", "")
87
88                  print(final_line)
89
90              time.sleep(WAITING_TIME) # default answer time is 30ms for
                    the G24 modem
91
92
93  def signal_handler(sig, frame):
94      print('[+] Terminating...')
95      try:
96          print("[+] Files closed!")
97          file_basic.close()
98          file_raw.close()
99      finally:
100         sys.exit(0)
101
102
103 if __name__ == "__main__":
104     main()
```

B.1.2 Modem emulation script

Listing B.2: CardioMessenger Serial Modem Script

```
1   #!/usr/bin/env python3
2
3   """
4       File name: cm-serial-modem.py
5       Version: 1.0
6       Author: Guillaume Bour
7       Last modified: 2018/05/29
8       License: GNU General Public License v3.0
9       Description: A script to emulate a modem via a UART connection
10          to the microcontroller.
11  """
12
13  import serial
14  import os
15  import sys
16  import time
17  import datetime
18  import signal
```

```
19
20
21  DISPLAY_NEW_LINE = False
22  VERSION = 1.0
23  TIMESTAMP_FORMAT = "%Y-%m-%d %H:%M:%S"
24  LOGGING = True
25  WAITING_TIME = 0.05 # 0.05 for CM as response time is set to 30ms
26  ECHO = False
27
28  BASIC_FILE_NAME = datetime.datetime.now().strftime("mod-%Y%m%d_%H%M%S.
        txt")
29  RAW_FILE_NAME = datetime.datetime.now().strftime("mod-%Y%m%d_%H%M%S_raw
        .txt")
30  FILES_FORMAT = datetime.datetime.now().strftime("file_%Y%m%d_%H%M%S_{}.
        bin")
31
32  file_basic = None
33  file_raw = None
34
35  def send_command(ser, timestamp, command):
36      print("> [{}] {}".format(timestamp, command))
37      command = "\r\n{}\r\n".format(command)
38      ser.write(bytearray(command, 'utf-8'))
39
40
41  def usage():
42      print("Usage: python3 cm-serial-modem.py SERIAL_PORT BAUDRATE")
43
44
45  def main():
46      if len(sys.argv) < 3:
47          usage()
48          sys.exit(0)
49
50      S_PORT = sys.argv[1]
51      B_RATE = sys.argv[2]
52
53      print("*** CM Serial Modem v{} ***".format(VERSION))
54      print("[+] Port: {}".format(S_PORT))
55      print("[+] Baudrate: {}".format(B_RATE))
56      print("[+] Display new line: {}".format("True" if DISPLAY_NEW_LINE
            else "False"))
57      print("[+] Current directory is {}".format(os.getcwd()))
58
59      with serial.Serial(S_PORT, B_RATE, timeout=1) as ser:
60          signal.signal(signal.SIGINT, signal_handler)
61
62          file_basic = open(BASIC_FILE_NAME, "a+")
63          file_raw = open(RAW_FILE_NAME, "wb")
64
65          print("[*] Logging raw input in {}".format(RAW_FILE_NAME))
```

```
66          print("[*] Logging decoded input in {}".format(BASIC_FILE_NAME)
                )
67          print("[*] Starting serial communication, press CTRL+C to
                terminate...\n")
68
69          DECODE_ERROR = False
70          FILE_BUFFER = b""
71          FILE_NB = 0
72          FILE_MODE = False
73
74          while True:
75              if FILE_MODE:
76                  raw_line = ser.read(ser.in_waiting)
77              else:
78                  raw_line = ser.readline()
79
80              timestamp = datetime.datetime.now().strftime(
                    TIMESTAMP_FORMAT)
81              time.sleep(WAITING_TIME) # default answer time is 30ms for
                    the G24 modem
82
83              if raw_line:
84                  line = ""
85                  try:
86                      line = raw_line.decode("ascii")
87                      DECODE_ERROR = False
88                      FILE_MODE = False
89                  except UnicodeDecodeError:
90                      line = raw_line.hex()
91                      DECODE_ERROR = True
92
93                  if (not FILE_MODE) and not (FILE_BUFFER == b""):
94                      with open(FILES_FORMAT.format(FILE_NB), "wb") as
                            ofile:
95                          ofile.write(FILE_BUFFER)
96                          ofile.flush()
97                      FILE_BUFFER = b""
98                      FILE_NB += 1
99
100                 final_line = "< [{}] {}".format(timestamp, line)
101
102                 file_raw.write(raw_line)
103                 file_basic.write(final_line)
104                 file_raw.flush()
105                 file_basic.flush()
106
107                 if DISPLAY_NEW_LINE:
108                     final_line = final_line.replace("\r","/r").replace(
                            "\n", "/n")
109                 else:
110                     final_line = final_line.replace("\r", "").replace("
                            \n", "")
```

```
111
112                 print(final_line)
113
114                 if (not FILE_MODE) and ECHO and (not DECODE_ERROR):
115                     send_command(ser, timestamp, raw_line.decode('ascii
                            '))
116
117                 if "atii5#vversion" in line:
118                     send_command(ser, timestamp, "OK")
119                 elif "AT#CONNECTIONSTART" in line:
120                     send_command(ser, timestamp, "DIALING")
121                     send_command(ser, timestamp, "TW[REDACTED]")
122                     send_command(ser, timestamp, "CONNECT 115200")
123                     send_command(ser, timestamp, "172.16.14.80")
124                     send_command(ser, timestamp, "OK_Info_PPP")
125                 elif "AT#OTCP=1" in line:
126                     send_command(ser, timestamp, "
                            OK_Info_WaitingForData")
127                     print("{} Switching to data mode {}".format("-"*5,
                            "-"*5))
128                     FILE_MODE = True
129                 elif "AT#" in line or "at#" in line:
130                     send_command(ser, timestamp, "OK")
131                 elif "AT+" in line or "at+" in line:
132                     send_command(ser, timestamp, "OK")
133                 elif "at" in line:
134                     send_command(ser, timestamp, "OK")
135                 else:
136                     if not DECODE_ERROR:
137                         if line == "\x00":
138                             print("Host sent NULL Byte - (Reset)")
139                             send_command(ser, timestamp, "OK")
140                         elif line == "\x03":
141                             print("Host sent ETX, back to command mode"
                                    )
142                             send_command(ser, timestamp, "OK")
143                             FILE_MODE = False
144                         else:
145                             print(line.encode('ascii').hex())
146                         print("INFO: {} Command not registered".format(
                                line.replace("\r","").replace("\n", "")))
147                         FILE_MODE = True # Trick to switch back to
                                command mode on RST
148                     elif FILE_MODE:
149                         FILE_BUFFER += raw_line
150                     else:
151                         print("Here but should not happened")
152
153
154 def signal_handler(sig, frame):
155     print('[+] Terminating...')
156     try:
```

```
157         print("[+] Files closed!")
158         file_basic.close()
159         file_raw.close()
160         if not (FILE_BUFFER == b""):
161             with open(FILES_FORMAT.format(FILE_NB), "wb") as ofile:
162                 ofile.write(FILE_BUFFER)
163                 ofile.flush()
164     finally:
165         sys.exit(0)
166
167
168 if __name__ == "__main__":
169     main()
```

B.2 Analysis scripts

Listing B.3: CardioMessenger Data Decryption Script

```
1  #!/usr/bin/env python3
2
3  """
4      File name: cm-decrypt.py
5      Version: 1.0
6      Author: Guillaume Bour
7      Last modified: 2018/05/25
8      License: GNU General Public License v3.0
9      Description: A script to decrypt the data sent by the HMU to
10         the backend server, given an AES key. Can also be used to
11         brute force the AES key given a binary dump of either the
12         RAM or the flash memory.
13 """
14
15 import os
16 import sys
17 import time
18 import argparse
19 import struct
20 import math
21 import zlib
22 from stat import *
23 from Crypto.Cipher import AES
24
25
26 NAME = "CM DECRYPT"
27 VERSION = "1.0"
28
29 GZIP_MAGIC_HEADER = "1f8b"
30 ESC_ELT = 0x10
31
32 ENC_DES = 6
33 ENC_3DES_CBC = 7
```

```
34 ENC_AES_CBC = 8
35
36
37 def print_file_info(filename):
38     print("** {} v{} **\n".format(NAME, VERSION))
39     print("[*] Opening {}...".format(filename))
40
41     with open(filename, "rb") as bin_file:
42         print("\n-- FILE INFORMATION --\n")
43         try:
44             st = os.stat(filename)
45         except IOError:
46             print("Failed to get information about {}".format(filename)
                   )
47             sys.exit(-1)
48         else:
49             print("File size: {} bytes (hex: {:02X})".format(st[ST_SIZE
                   ], st[ST_SIZE]))
50             print("File created: {}".format(time.asctime(time.localtime
                   (st[ST_MTIME]))))
51             print("File modified: {}".format(time.asctime(time.
                   localtime(st[ST_MTIME]))))
52             print("File entropy: {} bits per byte\n".format(H(bin_file.
                   read(), True)))
53
54
55 def sanitize(bin_data):
56     binary = b""
57     skip = False
58     for b in bin_data:
59         if not skip and b == ESC_ELT:
60             skip = True
61         else:
62            binary += bytes([b])
63            skip = False
64
65     # Fix weird bug
66     binary = bytes.fromhex(binary.hex().replace('aa01', 'aa'))
67     return binary
68
69
70 def getKeys(file_name):
71     key = b"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
         x00\x00"
72     with open(file_name, "rb") as bin_file:
73         next_byte = bin_file.read(1)
74         while next_byte != "":
75             key = key[1:] + next_byte
76             next_byte = bin_file.read(1)
77             yield key
78
79
```

```
80  # Adapted from http://blog.dkbza.org/2007/05/scanning-data-for-entropy-
        anomalies.html
81  def H(data, round_r = False):
82      if not data:
83          return 0
84      entropy = 0
85      for x in range(256):
86          p_x = float(data.count(x))/len(data)
87          if p_x > 0:
88              entropy += - p_x*math.log(p_x, 2)
89      if round_r:
90          return round(entropy, 2)
91      return entropy
92
93
94  def aes_decrypt(data, key, iv):
95      cipher = AES.new(key, AES.MODE_CBC, iv)
96      return cipher.decrypt(data)
97
98
99  def parse_message_layer(data):
100     if not QUIET:
101         print("\n** Message Layer **\n")
102
103     msgs = zlib.decompress(data, zlib.MAX_WBITS|32)
104     h = H(msgs, True)
105     msgs = msgs.hex()
106
107     print("Size of the recovered data: {} bytes".format(len(msgs)))
108     print("Number of packets: {}".format(len(msgs.split("0a"))))
109     print("Entropy: {}".format(h))
110
111
112 def parse_compression_layer(data):
113     if not QUIET:
114         print("\n** Compression Layer **\n")
115
116     type_p = struct.unpack(">B", data[0:1])[0]
117     magic_header = struct.unpack(">H", data[1:3])[0]
118     payload = data[1:]
119
120     if not type_p == 9:
121         print("[ERROR] Not a compression layer packet!")
122         sys.exit(-1)
123
124     if not QUIET:
125         is_mh = " => gzip compressed data, from Unix" if "{}".format(
                GZIP_MAGIC_HEADER) in payload.hex() else ""
126         print("Compression packet: Yes")
127         print("Magic header: 0x{:02x} {}".format(magic_header, is_mh))
128         print("Entropy: {}".format(H(payload, True)))
129
```

```
130     parse_message_layer(payload)
131
132
133 def parse_encryption_layer(data, key):
134     if not QUIET:
135         print("\n** Encryption Layer **\n")
136
137     type_p = struct.unpack(">B", data[0:1])[0]
138     padding = struct.unpack(">B", data[1:2])[0]
139     iv = data[2:18]
140     r_data = data[18:]
141
142     type_txt = ""
143     if type_p == ENC_AES_CBC:
144         type_txt = "AES_CBC"
145     elif type_p == ENC_3DES_CBC:
146         type_txt = "3DES CBC"
147     elif type_p == ENC_DES:
148         type_txt = "DES"
149     else:
150         print("[ERROR] Encryption type not known: {}".format(type_p))
151         sys.exit(-1)
152
153     if not QUIET:
154         div = "Yes" if len(data) % 16 == 0 else "No"
155         print("Length of the packet: {} (hex: {:02x}) (div by 16? {})".
              format(len(data), len(data), div))
156         print("Type of packet: {} (hex: {:02x}) => {}".format(type_p,
              type_p, type_txt))
157         print("Padding: {} (hex: {:02x})".format(padding, padding))
158         print("IV: {}".format(iv.hex()))
159         print("Key: {}\n".format(key.hex()))
160
161     if type_p == 8:
162         blocks = ( len(r_data) // 16 ) * 16
163         data = aes_decrypt(r_data[:blocks], key, iv)
164         parse_compression_layer(data)
165     else:
166         print("[ERROR] {}: encryption type not implemented".format(
              type_txt))
167         sys.exit(-1)
168
169
170 def parse_transport_layer(bin_content, key, get_payload = False):
171     if not QUIET:
172         print("** Transport Layer **\n")
173
174     type_p = struct.unpack(">B", bin_content[0:1])[0]
175     length = struct.unpack(">H", bin_content[1:3])[0]
176     unknown = struct.unpack(">B", bin_content[3:4])[0]
177     p_id = struct.unpack(">H", bin_content[4:6])[0]
178     cm_id = struct.unpack(">I", bin_content[6:10])[0]
```

```
179     checksum = struct.unpack(">H", bin_content[len(bin_content)-2:])[0]
180
181     if not QUIET:
182         print("Type of packets: {} (hex: {:02x})".format(type_p, type_p
                ))
183         print("Length: {} bytes (hex: {:04x})".format(length, length))
184         print("Unknown: {} (hex: {:02x})".format(unknown, unknown))
185         print("Packet ID: {} (hex: {:04x})".format(p_id, p_id))
186         print("CM ID: {} (hex: {:08x})".format(cm_id, cm_id))
187         print("Checksum: {} (hex: {:04x})".format(checksum, checksum))
188
189     payload = bin_content[10:len(bin_content)-2]
190
191     if get_payload:
192         return payload
193     else:
194         parse_encryption_layer(payload, key)
195
196
197 def bruteforce_aes(data, filename_bin):
198     if not QUIET:
199         print("\n[*] Brute forcing the AES key...")
200
201     type_p = struct.unpack(">B", data[0:1])[0]
202     iv = data[2:18]
203     r_data = data[18:]
204     blocks = ( len(r_data) // 16 ) * 16
205     c = 0
206
207     if type_p != ENC_AES_CBC:
208         print("[ERROR] Encryption type not recognized or not
                implemented ({})".format(type_p))
209         sys.exit(-1)
210
211     if not QUIET:
212         with open(filename_bin, "rb") as file_b:
213             print("[*] {} keys to try".format(len(file_b.read())))
214
215     for key in getKeys(filename_bin):
216         if len(key) != 16:
217             return None
218         d = aes_decrypt(r_data[:blocks], key, iv)
219
220         if "{}{}{}".format("09", GZIP_MAGIC_HEADER, "0800") in d.hex():
221             return (key, c - 15)
222
223         if not QUIET:
224             c += 1
225             if c % 10000 == 0:
226                 print(".", end="")
227                 sys.stdout.flush()
228     return None
```

```
229
230
231 def bruteforce_key(data, filename_ram):
232     if not QUIET:
233         print("[*] Brute force mode")
234         print("[*] Binary: {}\n".format(filename_ram))
235
236     payload = parse_transport_layer(data, None, True)
237
238     start = time.time()
239     res = bruteforce_aes(payload, filename_ram)
240     end = time.time()
241     t = int((end - start) * 100) / 100
242
243     if res:
244         (key, addr) = res
245         print("\n[*] Key Found in {}s!\n".format(t))
246         print("Key: {}".format(key.hex()))
247         print("Addr: 0x{:08x}".format(addr))
248     else:
249         print("\nKey not found! Time elapsed: {}s".format(t))
250
251
252 def decrypt_data(data, key):
253     if not QUIET:
254         print("[*] Decrypt mode")
255     parse_transport_layer(data, key)
256
257
258 if __name__ == "__main__":
259     parser = argparse.ArgumentParser(description="Decrypt a file sent
        by the Biotronik HMU or break the key used to encrypt the data.
        ")
260     parser.add_argument("filename", type=str, help="Filename of the
        binary data to be decrypted.")
261     parser.add_argument("-q", "--quiet", action="store_true", help="
        Only prints the result.")
262     parser.add_argument("-n", "--no-sanitize", action="store_true",
        help="Do not sanitize the data before attempting decryption (
        useful for the GSM version.")
263     group = parser.add_mutually_exclusive_group(required=True)
264     group.add_argument("-k", "--key", type=str, help="The key to
        decrypt the data (in hex).")
265     group.add_argument("-b", "--binary", type=str, help="The binary
        file used to brute force the key.")
266     args = parser.parse_args()
267
268     QUIET = args.quiet
269
270     if not QUIET:
271         print_file_info(args.filename)
272
```

```
273     with open(args.filename, "rb") as bin_file:
274         data = bin_file.read()
275 
276         if not args.no_sanitize:
277             data = sanitize(data)
278             if not QUIET:
279                 print("[*] Data sanitized!")
280 
281         if args.key:
282             decrypt_data(data, bytes.fromhex(args.key))
283         else:
284             bruteforce_key(data, args.binary)
```

Listing B.4: APDU Parser

```
1  #!/usr/bin/env python3
2  
3  """
4      File name: apdu-parser.py
5      Version: 1.0
6      Author: Guillaume Bour
7      Last modified: 2018/05/18
8      License: GNU General Public License v3.0
9      Description: A script to parse and extract information from raw
           APDU.
10 """
11 
12 import argparse
13 import os
14 import sys
15 import re
16 import string
17 
18 
19 # Instructions' mapping
20 op_codes = {
21     '0E': "ERASE BINARY",
22     '20': "VERIFY",
23     '70': "MANAGE CHANNEL",
24     '82': "EXTERNAL AUTHENTICATE",
25     '84': "GET CHALLENGE",
26     '88': "INTERNAL AUTHENTICATE",
27     'A4': "SELECT FILE",
28     'B0': "READ BINARY",
29     'B2': "READ RECORD(S)",
30     'C0': "GET RESPONSE",
31     'C2': "ENVELOPE",
32     'CA': "GET DATA",
33     'D0': "WRITE BINARY",
34     'D2': "WRITE RECORD",
35     'D6': "UPDATE BINARY",
36     'DA': "PUT DATA",
```

```
37      'DC': "UPDATE DATA",
38      'E2': "APPEND RECORD",
39  }
40
41  c_phone_numbers = set()
42  c_pin = -1
43
44
45  def rev_number(n):
46      nn = ""
47      for k in range(0, len(n), 2):
48          nn += n[k:k+2][::-1]
49      return "+"+nn
50
51
52  def find_numbers_hex(hex_data):
53      res = re.findall(r"94[0-9]{10}", hex_data)
54      return [rev_number(n) for n in res]
55
56
57  # Adapted from https://stackoverflow.com/questions/17195924/python-
        equivalent-of-unix-strings-utility
58  def strings(hex_data, min=4):
59      result = ""
60      for c in hex_data:
61          c = bytes([c]).decode('utf-8', "ignore")
62          if c in string.printable:
63              result += c
64              continue
65          if len(result) >= min:
66              yield result
67          result = ""
68      if len(result) >= min:  # catch result at EOF
69          yield result
70
71
72  # Parses the Logic Analyzer CSV and returns the HEX data
73  # start_idx is the row's index to start with (1 = skip the first line)
74  def get_hex_data_from_csv(filename, start_idx=1):
75      with open(filename) as csv_file:
76          d = ""
77          for line in csv_file.readlines()[start_idx:]:
78              d += line.split(",")[2].split(' ')[0].replace('\n','').
                    replace('0x', '')
79          return d
80
81
82  # Returns a list of messages based on a delimiter
83  # skip_first: skip the first message if garbage detected
84  def get_apdu(hex_data, delimiter="A0", skip_first = True):
85      splitted_data = hex_data.split(delimiter)
86      apdu = []
```

```
87      prefix = ""
88      garbage = hex_data.startswith("00")
89
90      for line in splitted_data:
91          if prefix:
92              line = "{}{}{}".format(prefix, delimiter, line)
93              prefix = ""
94          if len(line) % 2 == 0:
95              apdu.append("{}{}".format(delimiter, line))
96          else:
97              prefix = line
98
99      if skip_first and garbage:
100         return apdu[1:]
101
102     return apdu
103
104
105 def parse_data(data, op):
106     global c_pin
107
108     if op == '20':
109         data = data.replace("FF", "")
110         pin = bytes.fromhex(data).decode('ascii')
111         c_pin = pin
112         return "PIN: {}".format(pin)
113     if op == "B2":
114         res = re.findall(r"94[0-9]{10}", data)
115         nums = "".join([rev_number(n) for n in res])
116         if nums:
117             c_phone_numbers.add(nums)
118         return "Data = {}\n{}".format(data, nums)
119
120     return "Data: {}".format(data)
121
122
123 def parse_apdu_response(response, le, display):
124     op_code = response[0:2]
125     data = response[2:2+le*2]
126     sw1 = response[2+le*2:4+le*2]
127     sw2 = response[4+le*2:6+le*2]
128
129     op = ""
130     if op_code in op_codes.keys():
131         op = op_codes[op_code]
132     else:
133         op = "UNKNONW"
134
135     d = parse_data(data, op_code)
136
137     if not display:
```

```
138         print("{}|DATA |{}|{} - REPLY TO {}".format(op_code, sw1, sw2,
                op))
139         print(d)
140
141
142 def parse_apdu(pdu, display):
143     cla = pdu[0:2]
144     ins = pdu[2:4]
145     p1  = pdu[4:6]
146     p2  = pdu[6:8]
147     le  = pdu[8:10]
148     le_int = int("0x{}".format(le), 0)
149     op = ""
150
151     if ins in op_codes.keys():
152         op = op_codes[ins]
153     else:
154         op = "UNKNONW"
155
156     if not display:
157         print("{}|{}|{}|{}|{} - {}".format(cla, ins, p1, p2, le, op))
158
159     parse_apdu_response(pdu[10:], le_int, display)
160
161     if not display:
162         print("-" * 30)
163
164
165 if __name__ == "__main__":
166     parser = argparse.ArgumentParser(description="Parses APDU
            eavesdropped with Salae Logic's and outputs the information
            gathered.")
167     parser.add_argument('filename', help="Logic's CSV output")
168     parser.add_argument('--hexonly', action='store_true', help="Parses
            the CSV and outputs the HEX directly on standard output")
169     parser.add_argument('--nodetails', action='store_true', help="Does
            not display any APDU")
170
171     args = parser.parse_args()
172
173     if args.filename and os.path.isfile(args.filename):
174         hex_data = get_hex_data_from_csv(args.filename)
175         if args.hexonly:
176             print(hex_data)
177             sys.exit(0)
178
179         apdus = get_apdu(hex_data)
180         for pdu in apdus:
181             parse_apdu(pdu, args.nodetails)
182
183         print("** ANALYSIS RESULTS **")
184         if c_pin != -1:
```

```
185             print("\nPIN found: {}".format(c_pin))
186
187         c_phone_numbers = list(c_phone_numbers)
188         print("\n{} german's phone numbers found in records:".format(
                len(c_phone_numbers)))
189         for phone in c_phone_numbers:
190             print(phone)
191
192         nums = list(set(find_numbers_hex(hex_data)))
193         print("\n{} german's phone numbers found in total in the hex
                dump:".format(len(nums)))
194         print("/!\ This is raw data and can be misleading, be careful
                when dealing with it.")
195         for phone in nums:
196             print(phone)
197
198         print("\nPlain texts found in the hex dump:")
199         for word in strings(bytes.fromhex(hex_data), 4):
200             print(word.strip().strip())
```

B.3 OpenOCD configuration files

Listing B.5: Basic connection to a target

```
1  # INTERFACE
2  interface bcm2835gpio
3  bcm2835gpio_peripheral_base 0x20000000
4  bcm2835gpio_speed_coeffs 113714 28
5  bcm2835gpio_jtag_nums 11 25 10 9
6  bcm2835gpio_srst_num 24
7  reset_config srst_only srst_push_pull
8  adapter_khz 500
9
10 # TRANSPORT
11 transport select jtag
12
13 # TARGET
14 set WORKAREASIZE 0
15 set CHIPNAME at91rm9200
16 source [find target/at91rm9200.cfg]
17 reset_config srst_only srst_nogate
18 adapter_nsrst_delay 100
19 adapter_nsrst_assert_width 100
20
21 # EXEC
22 init
23 targets
24 halt
```

Listing B.6: Dumping the memory of a target

```
# 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 at91rm9200
source [find target/at91rm9200.cfg]
reset_config srst_only srst_nogate
adapter_nsrst_delay 100
adapter_nsrst_assert_width 100

# EXEC
init
targets
halt

echo "Dumping bootloader..."
dump_image bootloader.img 0x00000000 1048576
echo "Done!"

echo "Dumping SRAM..."
dump_image sram.img 0x00200000 104576
echo "Done!"

echo "Dumping Flash content..."
dump_image flash.img 0x10000000 4194304
echo "Done!"

echo "Dumping RAM..."
dump_image sdram.img 0x20000000 2097152
echo "Done!"
```

Appendix C
Detailed listings

C.1 CardioMessenger II-S T-Line

C.1.1 Modem's configurations

Listing C.1: Default modem's configuration

```
at

OK
at

OK
at#vall


#VVERSION: VERSION 0.11

#ANSWERMODE: 0
#CALLBACKTIMER: 2
#CALLSCREENNUM: "*"
#DIALN1: ""
#DIALN2: ""
#DIALSELECT: 1
#PHYTIMEOUT: 15
#REDIALCOUNT: 0
#REDIALDELAY: 0
#RINGCOUNT: 0

#FTPGETFILENAME: ""
#FTPGETPATH: ""
#FTPMODE: 0
#FTPPORT: 21
#FTPPUTFILENAME: ""
```

```
#FTPPUTPATH: ""
#FTPPW: ""
#FTPSERV: ""
#FTPTYPE: I
#FTPUN: ""

#POP3HEADERMODE: 1
#POP3PORT: 110
#POP3PW: ""
#POP3SERV: ""
#POP3UN: ""

#DOMAIN: ""
#SENDERADDR: ""
#SENDERNAME: ""
#SMTPPORT: 25
#SMTPPW: ""
#SMTPSERV: ""
#SMTPUN: ""
#SMTPAUTH: 1

#BODY1: ""
#CCREC1: ""
#REC1: ""
#SUBJ1: ""

#DLEMODE: 1 , 1
#TCPSERV: 1 , ""
#TCPPORT: 1 , 0
#TCPTXDELAY: 1 , 100

#UDPSERV: 1 , ""
#UDPPORT: 1 , 0
#UDPTXDELAY: 1 , 100

#PINGDELAY: 1
#PINGNUM: 4
#PINGREMOTE: ""

#ISPUN: ""
#ISPPW: ""
#PPPMODE: 3
#PPPMYIP: "0.0.0.0"
#PPPPEERIP: "0.0.0.0"
#AUTHENT: NONE
```

```
#DNSSERV1: ""
#DNSSERV2: ""

+IPR: 115200
+ICF: 2,4
+IFC: 2,2

V: 1
E: 1
&S: 1
&C: 2
#MCOUNTRY: B5

#ATCMD: 0, "-STE=7"
#ATCMD: 1, "+MS=V34"
#ATCMD: 2, ""
#ATCMD: 3, ""

OK
```

Listing C.2: Modem's configuration

```
AT#DIALN1="TW[REDACTED]"
AT#ISPUN="[REDACTED]@cm3-homemonitoring.de"
AT#ISPPW="[REDACTED]"
at#atcmd=0,"-STE=7"
### Connection to the Modem here ###
at

OK
at#vversion

#VVERSION: VERSION 0.11

OK
at#vall


#VVERSION: VERSION 0.11

#ANSWERMODE: 0
#CALLBACKTIMER: 2
#CALLSCREENNUM: "*"
#DIALN1: "TW[REDACTED]"
#DIALN2: ""
#DIALSELECT: 1
```

```
#PHYTIMEOUT: 15
#REDIALCOUNT: 0
#REDIALDELAY: 0
#RINGCOUNT: 0

#FTPGETFILENAME: ""
#FTPGETPATH: ""
#FTPMODE: 0
#FTPPORT: 21
#FTPPUTFILENAME: ""
#FTPPUTPATH: ""
#FTPPW: ""
#FTPSERV: ""
#FTPTYPE: I
#FTPUN: ""

#POP3HEADERMODE: 1
#POP3PORT: 110
#POP3PW: ""
#POP3SERV: ""
#POP3UN: ""

#DOMAIN: ""
#SENDERADDR: ""
#SENDERNAME: ""
#SMTPPORT: 25
#SMTPPW: ""
#SMTPSERV: ""
#SMTPUN: ""
#SMTPAUTH: 1

#BODY1: ""
#CCREC1: ""
#REC1: ""
#SUBJ1: ""

#DLEMODE: 1 , 1
#TCPSERV: 1 , ""
#TCPPORT: 1 , 0
#TCPTXDELAY: 1 , 100

#UDPSERV: 1 , ""
#UDPPORT: 1 , 0
#UDPTXDELAY: 1 , 100

```

```
#PINGDELAY: 1
#PINGNUM: 4
#PINGREMOTE: ""

#ISPUN: "[REDACTED]@cm3-homemonitoring.de"
#ISPPW: "[REDACTED]"
#PPPMODE: 3
#PPPMYIP: "0.0.0.0"
#PPPPEERIP: "0.0.0.0"
#AUTHENT: PAP
#DNSSERV1: ""
#DNSSERV2: ""

+IPR: 115200
+ICF: 2,4
+IFC: 2,2

V: 1
E: 1
&S: 1
&C: 2
#MCOUNTRY: B5

#ATCMD: 0, "-STE=7"
#ATCMD: 1, "+A8E=6,5,0,1,0,0"
#ATCMD: 2, "X3"
#ATCMD: 3, ""

OK
```

C.1.2 JTAGulator

Listing C.3: Pins determination using the JTAGulator

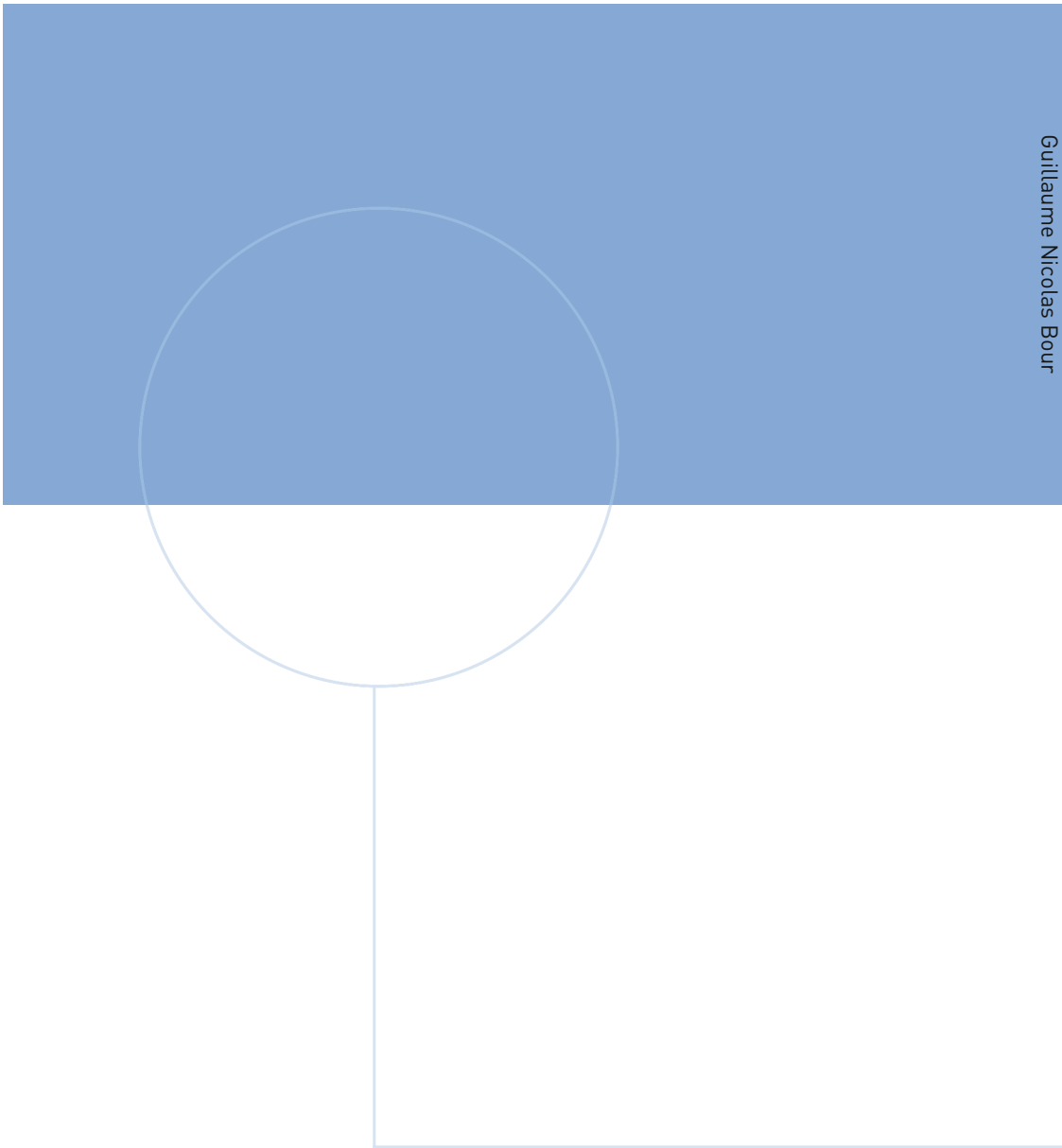
```
Welcome to JTAGulator. Press 'H' for available commands.
Warning: Use of this tool may affect target system behavior!

> h

Target Interfaces:
J   JTAG/IEEE 1149.1
U   UART/Asynchronous Serial
G   GPIO

General Commands:
V   Set target I/O voltage (1.2V to 3.3V)
```

```
13 I   Display version information
14 H   Display available commands
15
16 > v
17
18 Current target I/O voltage: Undefined
19 Enter new target I/O voltage (1.2 - 3.3, 0 for off): 3
20
21 New target I/O voltage set: 3.0
22 Ensure VADJ is NOT connected to target!
23
24 > j
25
26
27 JTAG> h
28
29 JTAG Commands:
30 I   Identify JTAG pinout (IDCODE Scan)
31 B   Identify JTAG pinout (BYPASS Scan)
32 D   Get Device ID(s)
33 T   Test BYPASS (TDI to TDO)
34 Y   Instruction/Data Register (IR/DR) discovery
35 X   Transfer instruction/data
36 C   Set JTAG clock speed
37
38 General Commands:
39 V   Set target I/O voltage (1.2V to 3.3V)
40 H   Display available commands
41 M   Return to main menu
42
43 JTAG> i
44
45 Enter starting channel [0]:
46
47 Enter ending channel [0]: 12
48
49 Possible permutations: 1716
50
51 Bring channels LOW between each permutation? [y/N]: y
52
53 Enter length of time for channels to remain LOW (in ms, 1 -
       1000) [100]:
54
55 Enter length of time after channels return HIGH before
       proceeding (in ms, 1 - 1000) [100]:
```

```
56
57 Press spacebar to begin (any other key to abort)...
58 JTAGulating! Press any key to abort...
59 -----------------------------------------
60 TDI: N/A
61 TDO: 11
62 TCK: 4
63 TMS: 12
64 Device ID #1: 0000 0101101100000010 00000011111 1 (0x05B0203F)
65 TRST#: 3
66 TRST#: 10
67
68 -----------------------------------------
69 IDCODE scan complete.
70
71 JTAG> h
72
73 JTAG Commands:
74 I   Identify JTAG pinout (IDCODE Scan)
75 B   Identify JTAG pinout (BYPASS Scan)
76 D   Get Device ID(s)
77 T   Test BYPASS (TDI to TDO)
78 Y   Instruction/Data Register (IR/DR) discovery
79 X   Transfer instruction/data
80 C   Set JTAG clock speed
81
82 General Commands:
83 V   Set target I/O voltage (1.2V to 3.3V)
84 H   Display available commands
85 M   Return to main menu
86
87 JTAG> b
88
89 Enter starting channel [0]:
90
91 Enter ending channel [12]:
92
93 Are any pins already known? [Y/n]: y
94
95 Enter X for any unknown pin.
96 Enter TDI pin [0]: x
97
98 Enter TDO pin [11]:
99
100 Enter TCK pin [4]:
```

```
101
102 Enter TMS pin [12]:
103
104 Possible permutations: 10
105
106 Bring channels LOW between each permutation? [Y/n]: y
107
108 Enter length of time for channels to remain LOW (in ms, 1 -
        1000) [100]:
109
110 Enter length of time after channels return HIGH before
        proceeding (in ms, 1 - 1000) [100]:
111
112 Press spacebar to begin (any other key to abort)...
113 JTAGulating! Press any key to abort...
114 ----
115 TDI: 5
116 TDO: 11
117 TCK: 4
118 TMS: 12
119 TRST#: 3
120 TRST#: 10
121 Number of devices detected: 1
122 ------
123 BYPASS scan complete.
124
125 JTAG> h
126
127 JTAG Commands:
128 I    Identify JTAG pinout (IDCODE Scan)
129 B    Identify JTAG pinout (BYPASS Scan)
130 D    Get Device ID(s)
131 T    Test BYPASS (TDI to TDO)
132 Y    Instruction/Data Register (IR/DR) discovery
133 X    Transfer instruction/data
134 C    Set JTAG clock speed
135
136 General Commands:
137 V    Set target I/O voltage (1.2V to 3.3V)
138 H    Display available commands
139 M    Return to main menu
140
141 JTAG> d
142
143 TDI not needed to retrieve Device ID.
```

```
144 Enter TDO pin [11]:
145
146 Enter TCK pin [4]:
147
148 Enter TMS pin [12]:
149
150
151 Device ID #1: 0000 0101101100000010 00000011111 1 (0x05B0203F)
152 -> Manufacturer ID: 0x01F
153 -> Part Number: 0x5B02
154 -> Version: 0x0
155 IDCODE listing complete.
156
157 JTAG> t
158
159 Enter TDI pin [5]:
160 Enter TDO pin [11]:
161 Enter TCK pin [4]:
162 Enter TMS pin [12]:
163
164 Number of devices detected: 1
165 Pattern in to TDI:    10001010000011100011111111001001
166 Pattern out from TDO: 10001010000011100011111111001001
167 Match!
168
169
170 JTAG> y
171
172 Enter TDI pin [5]:
173 Enter TDO pin [11]:
174 Enter TCK pin [4]:
175 Enter TMS pin [12]:
176 Ignore single-bit Data Registers? [Y/n]: y
177
178 Instruction Register (IR) length: 4
179 Possible instructions: 16
180 Press spacebar to begin (any other key to abort)...
181 JTAGulating! Press any key to abort...
182 -
183 IR: 0010 (0x2) -> DR: 5
184 -
185 IR: 1110 (0xE) -> DR: 32
186
187 IR/DR discovery complete.
```



Guillaume Nicolas Bour

NTNU
Norwegian University of
Science and Technology