NTNU
Norwegian University of
Science and Technology

Attacking Message Authentication Codes in EFC Using Rainbow Tables

Sverre Turter Sandvold

Master of Science in Communication Technology
Submission date: June 2017
Supervisor: Stig Frode Mjølsnes, IIK
Co-supervisor: Tord Reistad, Statens Vegvesen

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title: Attacking Message Authentication Codes in EFC Using Rainbow Tables

Student: Sverre Turter Sandvold

Problem Description:

Electronic Fee Collection (EFC) is one of many systems in the world of intelligent transportation systems. The main components in EFC are an On-board Unit (OBU) and a Roadside Equipment (RSE), which uses Dedicated Short-Range Communications (DSRC) as a communication protocol. The authentication of the OBU is generated using single Data Encryption Standard (DES), and with a customizable RSE, it should be possible for a chosen plaintext attack on this authentication value. The customizable RSE will be made by using a universal software radio peripheral together with GNU Radio.

In 2003 Philippe Oechslin published a paper describing a cryptanalytic time-memory trade-off method for breaking hashes, called rainbow tables [1]. Generation of rainbow tables is highly parallelizable and therefore graphical processing units will be used for computation to make this attack as fast as possible.

This project will have two main parts. (I) To make a customizable RSE who can communicate with an OBU and (II) to make an attack on the message authentication codes (MAC) by conducting a rainbow table attack.

[1] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings, volume 2729 of Lecture Notes in Computer Science, pages 617–630. Springer, 2003.

Responsible professor: Stig Frode Mjølsnes, IIK

Supervisor: Tord Ingolf Reistad, Statens Vegvesen

Abstract

Electronic Fee Collection (EFC) is a system for collecting fee from cars electronically. EFC contains two main components, an On-board Unit (OBU) and a Roadside Equipment (RSE). Dedicated Short-Range Communications (DSRC) is used as a communication protocol, which is a wireless protocol operating on frequencies around 5.8 GHz. As more and more systems take use of EFC and DSRC, it is important to keep security in mind. Money transactions are involved in EFC, making it a wanted target for attackers.

In this thesis, the possibilities of conducting an attack on Message Authentication Codes (MACs) in EFC are explored. By stealing a valid key used in the calculation of a MAC, it can be possible to get authenticated as another user and hence do not have to pay any fee.

This thesis shows that with a customizable RSE, it is possible to obtain MACs from OBUs. A customizable RSE is built using a Universal Software Radio Peripheral and GNU Radio. Unfortunately, the customizable RSE is not able to communicate with an OBU. Some errors are known, and suggestions on how to solve them are presented. A desktop DSRC transceiver was bought from Q-Free and is used as an RSE in the second part of this thesis. MACs are obtained from a test-OBU using the DSRC transceiver, and it is shown that OBUs can be used as encryption oracles. Before obtaining MACs, access credentials have to be given by the OBU. A test-OBU is used to get access credentials. However, access credentials is not used in every country, making this attack relevant. Possible attacks for obtaining access credentials are also presented.

Additionally, the use of Data Encryption Standard (DES) for calculating MACs is studied. The biggest weakness of DES is the 56-bit key, making it feasible to brute-force the entire keyspace. This thesis shows how a rainbow table can be used to find the key in an OBU. Rainbow tables are a time-memory trade-off method for finding keys in a chosen-plaintext attack. A simple rainbow table is implemented in Python, together with suggestions on how improve the rainbow table generation.

Sammendrag

Elektronisk bompengeinnkreving (EFC) brukes til å elektronisk samle inn bompenger fra biler som kjører på bomveier. EFC består av to hovedkomponenter, en bombrikke (OBU) og en bomstasjon (RSE). Dedikert kortholdslink (DSRC) blir brukt som kommunikasjonsprotokoll, som er en tråløs protokoll for frekvenser rundt 5.8 GHz. Ettersom flere og flere tar i bruk EFC og DSRC, blir det stadig mer viktig å tenke på sikkerheten. Pengetransaksjoner er innvolvert i EFC, og EFC er derfor et naturlig mål for angripere.

Denne oppgaven ser på mulighetene til å gjennomføre angrep på meldingsautentiseringskodene (MAC) i EFC. Ved å stjele en gyldig nøkkel, som blir brukt til å regne ut MAC-verdier, vil det være mulig å kjøre gjennom en bomstasjon uten å måtte betale.

Denne oppgaven viser at med en falsk bomstasjon vil det være mulig å motta meldingsautentiseringskoder fra bombrikker. En falsk bomstasjon er blitt lagd ved hjelp av en Universal Software Radio Peripheral (USRP) og GNU Radio. Den falske bomstasjoen fungerer dessverre ikke. Noen av feilene er kjente, og mulige løsninger til problemene er beskrevet. En DSRC-bordleser ble kjøpt inn fra Q-Free, og denne blir i siste del av oppgaven brukt som bomstasjon. Ved hjelp av bordleseren blir meldingsautentiseringskoder hentet fra en bombrikke, og det blir også vist hvordan bombrikkene kan bli brukt som krypteringsorakler. For å kunne hente ut meldingsautentiseringskoder, må man først skaffe seg aksessrettigheter fra bombrikken. Det ble derfor benyttet en test-bombrikke med kjent aksessnøkkel. Aksess kontroll blir ikke brukt i alle land, noe som gjør angrepet beskrevet i denne oppgaven veldig aktuelt. Mulige angrep mot aksess kontrollen er også presentert.

I tillegg, blir bruken av Data Encryption Standard (DES) diskutert. Den største svakheten til DES er at nøkkelen kun er 56 bit. Med en så kort nøkkel vil det være mulig å gjennomføre et brute-force angrep i løpet av relativt kort tid. Denne oppgaven viser hvordan regnbuetabeller kan brukes til å finne en nøkkel som har blitt brukt til å lage en meldingsautentiseringkode. Regnbuetabeller er en metode der man avveier tid og lagringsplass. Disse tablellene kan bli brukt til å finne nøkler i et kjent klartekst angrep. En enkel regnbuetablell er utviklet i Python sammen med forslag til hvordan en slik tabell kan lages best mulig.

Preface

This Master's thesis is carried out in the 10th semester of my Master of Science degree in Communication Technology at Norwegian University of Science and Technology.

I would like to thank my responsible professor Stig Frode Mjølsnes and my supervisor Tord Ingolf Reistad for their guidance and feedback during this semester. I also want to thank Ulf Bertilsson for inspirational meetings and help with building the customizable RSE.

Trondheim, June 2017

Sverre Turter Sandvold

Contents

List of Figures

List of Tables

List of Acronyms

AcK Access Credentials Key.

AID Attribute Identifier.

AuK Authenticator Key.

BST Beacon Service Table.

CBC Cipher Block Chaining.

CRC Cyclic Redundancy Check.

DES Data Encryption Standard.

DSRC Dedicated Short-Range Communications.

EFC Electronic Fee Collection.

EID Element Identifier.

EPC Electronic Product Code.

FCS Frame Check Sequence.

GHz Gigahertz.

GRC GNU Radio Companion.

GUI Graphical User Interface.

ITS Intelligent Transportation Systems.

LID Link Identifier.

LPDU Link Protocol Data Unit.

MAC Message Authentication Code.

MAcK Master Access Credentials Key.

MAuK Master Authenticator Key.

MHz Megahertz.

NRZI non-returned-to-zero-inverted.

NTNU Norwegian University of Science and Technology.

OBU On-board Unit.

PSK Phase Shift Keying.

RF Radio Frequency.

RFID Radio Frequency Identification.

RSE Roadside Equipment.

RSE624 Q-Free Desktop DSRC Transceiver.

SDR Software Defined Radio.

UHD USRP Hardware Driver.

USRP Universal Software Radio Peripheral.

VST Vehicle Service Table.

XOR Exclusive Or.

Chapter 1
Introduction

1.1 Motivation

As technology is developing rapidly, the need of Intelligent Transportation Systems (ITS) increase. One of them is Electronic Fee Collection (EFC), which is a system for electronically collecting fee on toll roads. EFC uses Dedicated Short-Range Communications (DSRC) as communication protocol, which operates on radio frequencies between 5.725 Gigahertz (GHz) and 8.875 GHz. DSRC is a protocol intended for communication between vehicle and vehicle or vehicle and infrastructure.

The Norwegian toll road system, called AutoPASS is a free-flow system that uses video tolling and reading of On-board Units (OBUs) using DSRC. This thesis will only focus on the OBU part of the system. An EFC system contains two main components, an OBU mounted in the vehicle and a Roadside Equipment (RSE) mounted usually in a gantry over the roads. The OBUs and RSE used in Norway are technically very similar to those used in most of Europe as the use of DSRC OBUs is mandated by the EU (Commission Decision 2009/750/EC and Directive 2004/52/EC).

Security is important in EFC, mainly because money transactions are involved, but also because it can contain sensitive information on its users. In a draft paper [Rei] written by Tord Reistad, he describes that an attack on Message Authentication Codes (MACs) is possible. MACs are used by the RSE to authenticate OBUs. The final goal of the attack is to build a customizable OBU, that will be authenticated by a legitimate RSE, as a valid OBU. By stealing the key used for calculating the MAC from another OBU, and using it in a customizable OBU, another user will have to pay each time the customizable OBU pass any toll station. By stealing hundreds, or thousands, of keys, and changing the key in the OBU frequently, it will be almost impossible to get caught.

1.2 Scope and Objectives

With a customizable RSE it should be possible to conduct an attack on the MACs and hence steal keys. MACs are used by the RSE to authenticate a user, and the MAC is therefore calculated by the OBU. The customizable RSE will be used to obtain MACs, by implementing the DSRC protocol. In Reistad's draft paper [Rei], he shows that EFC is theoretically vulnerable for a chosen-plaintext attack. Data Encryption Standard (DES) is used to calculate MACs and it is calculated over data controlled by the RSE. DES is a block cipher with a key of only 56 bits, making it vulnerable against brute-force attacks. Rainbow tables are precomputed tables and can be used to find a key in a chosen-plaintext attack. The objectives of this thesis will be to:

1. Make a customizable RSE who can communicate with an OBU
2. Obtain several MACs from an OBU
3. Make an attack on MACs by building a rainbow table

The goal of this thesis is to show that an attack on the MACs is feasible, and that it is possible with relatively cheap and available equipment. Please note that building a customizable OBU with shifting keys, is not an objective of this thesis.

1.3 Methodology

A customizable RSE will be made by using a Universal Software Radio Peripheral (USRP). A USRP is a Software Defined Radio (SDR) made by Ettus and can be programmed to fit many different applications. GNU Radio provides signal processing blocks for implementation on SDRs and is used as software on the USRP.

MACs will be obtained from an OBU by using a customizable RSE. A customizable RSE gives possibilities to construct messages down to bit-level, making it possible to communicate with a legitimate OBU.

Initially, a goal was to make a rainbow table for Graphics Processing Units (GPUs). Generating a rainbow table is highly parallelizable, and implementing it on GPUs will make the generation a lot faster than on Central Processing Units (CPUs). Properly doing this is very time consuming and is not an easy task without any experience in GPU programming. Because of lack of time, a rainbow table is not implemented on GPUs but instead written in Python for CPUs.

1.4 Related Work

This thesis will be a continuation of Jonathan Hansen's Master's thesis from 2016. Hansen aimed to set up a test-bed for DSRC applications, and then to analyze the security of the current protocols. The test-bed was intended to be used to simulate an RSE and an OBU. Hansen did not manage to finish the test-bed, but his work is still valuable for this thesis.

1.5 Outline

This thesis is divided into six chapters. The outline is as follows.

Chapter 2 presents a general foundation of EFC and DSRC, including the parts of the standards that are relevant to fully understand the content of this thesis.

Chapter 3 describes the approach for making a customizable RSE with a USRP and GNU Radio.

Chapter 4 explains how the communication between an RSE and an OBU works. It also describes how messages can be constructed to obtain a MAC from an OBU.

Chapter 5 describes how the MACs obtained in Chapter 4 can be broken using rainbow tables. A simple example of how such a table can be built is presented, together with suggestions on how to speed up the table generation.

Chapter 6 includes the conclusion of the work done in this thesis. Also, some suggestions for further work are described and discussed.

Chapter 2
Electronic Fee Collection

This chapter provides background information on EFC and DSRC. Especially, the physical layer of DSRC and the security of EFC is described. Finally, the block cipher used in EFC, DES, is presented.

2.1 Overview

EFC in Norway, and most of Europe, contains two main components, an OBU and an RSE. An OBU is placed inside a car, while an RSE is placed alongside a toll road. Figure 2.1 shows an overview of the main components in EFC. The first part of this thesis will focus on the components and the communication inside the dotted line of Figure 2.1.

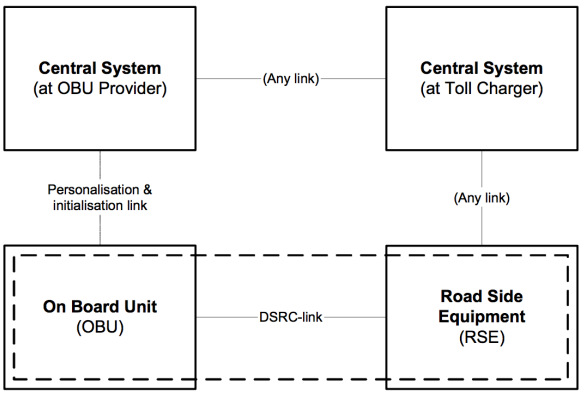

Figure 2.1: An overview of the main components in EFC. Components relevant for this thesis, an OBU, an RSE, and the DSRC-link, are inside the dotted line. Source: [CEN07].

2.2 Dedicated Short-Range Communications

As earlier previously, this project focuses on EFC, which uses DSRC as communication protocol. DSRC follows standards developed by the European Committee for Standardization (CEN) and the International Organization for Standardization (ISO). The relevant standard used are:

- EN 12253:2004 Road transport and traffic telematics – Dedicated short-range communication – Physical layer using microwave at 5.8 GHz [CEN04a]
- EN 12795:2003 Road transport and traffic telematics – Dedicated short-range communication (DSRC) – DSRC data link layer: medium access and logical link control [CEN03a]
- EN 12834:2003 Road transport and traffic telematics – Dedicated short-range communication (DSRC) – DSRC application layer [CEN03b]
- EN 13372:2004 Road transport and traffic telematics (RTTT) – Dedicated short-range communication – Profiles for RTTT applications [CEN04b]

The first three of these standards, EN 12253, EN 12795, and EN 12834, forms a three-layered architecture for DSRC, as seen in Table 2.1. EN 13372 specifies DSRC profiles for Road Transport and Traffic Telematics (RTTT) applications.

Table 2.1: DSRC protocol stack

Layer	DSRC Standard
Layer 7 - Application	EN 12834
Layer 2 - Data link	EN 12795
Layer 1 - Physical	EN 12253

2.2.1 DSRC Physical Layer

To be able to build a customizable RSE, it is essential to know the requirements at the physical layer. These requirements are specified in EN 12253 [CEN04a]. Table 2.2 and Table 2.3 summarize the most relevant parameters of the physical layer of DSRC.

The carrier frequency has two channels, which are at 5.7975 GHz and 5.8025 GHz. The polarization of the signal transmitted is left-hand circular and is modulated with a two level amplitude modulation. Bits are encoded using FM0 and transferred with a bit rate of 500 kbit/s. An OBU is a passive unit, which collects energy from the

signal sent from an RSE to wake up. OBUs also have a small battery, but this is only used for calculations, and not for transmitting signals or reading data.

Data sent from an OBU to an RSE is sent on a sub-carrier frequency in either 1.5 MHz or 2.0 MHz away from the carrier frequency. These signals are modulated using 2-PSK, which are binary phase shift keying. Uplink data is encoded using non-returned-to-zero-inverted (NRZI) and is sent with a bit rate of 250 kbit/s.

Table 2.2: Most important physical layer downlink parameters in DSRC [CEN04a].

Item No.	**Parameter**	**Value**
D1	Carrier Frequencies	Downlink channel 1: 5.7975 GHz Downlink channel 2: 5.8025 GHz
D5	Polarization	Left hand circular
D6	Modulation	Two level amplitude modulation
D7	Data Coding	FM0
D8	Bit Rate	500 kbit/s
D10	Wake-up Trigger for OBU	OBU shall wake on receiving any frame with 11 or more octets
D13	Preamble	16 bits ± 1 bit. Alternating sequence of low and high level.

Table 2.3: Most important physical layer uplink parameters in DSRC [CEN04a].

Item No.	**Parameter**	**Value**
U1	Sub-carrier Frequencies	1.5 MHz or 2.0 MHz
U6	Sub-carrier Modulation	2-PSK
U7	Data Coding	NRZI
U8	Bit Rate	250 kbit/s

2.3 EFC Functions and Applications

In addition to the DSRC standards described above, EN 14906 and EN 15509 contains specifications for EFC applications on top of DSRC:

- EN ISO 14906:2011 Electronic fee collection – Application interface definition for dedicated short-range communication [CEN11]

– EN 15509:2007 Road transport and traffic telematics – Electronic fee collection – Interoperability application profile for DSRC [CEN07]

EN 14906 is an application interface for EFC, and provides a specification on the EFC transaction model, data elements and functions, which can be used to build an EFC transaction. EN 15509 specifies some specific EFC functions, together with examples of transactions. In Table 2.4 the EFC functions are listed together with the corresponding DSRC layer 7 services. A complete EFC-transaction is illustrated in a message sequence diagram in Figure 2.2.

DSRC-L7	**EFC**	**Remark**
INITIALISATION	-	To establish communication between OBU and RSE
ACTION	GET_STAMPED	Get data with authenticator from OBU
GET	-	Get data from OBU
SET	-	Write data to OBU
ACTION	SET_MMI	Invokes a MMI function
ACTION	ECHO	OBU echoes received data
EVENT-REPORT	RELEASE	Terminates communication

Table 2.4: Overview of DSRC layer 7 and EFC functions [CEN07].

Constantly, a Beacon Service Table (BST) is sent out from an RSE. When an OBU enters the communication zone, it evaluates and responds to the BST. The most relevant fields in the BST are BeaconID and Time. An OBU uses these fields to evaluate whether it should respond to the BST or not. The OBU has always stored the most recent BeaconID and Time. When it enters the communication zone and receives a BST, it checks if the BeaconID differs from the one stored. If not the BeaconID is equal, it checks if the time difference is more than 255 seconds. If the result of both these tests is NO, the OBU stores the current Time and do nothing more. If one of the tests are YES, the OBU creates a new Private LID, stores the BeaconID and Time, and respond with a message to the RSE.

After a BST is received and evaluated, the OBU answers with a Private Window Request. Then, the RSE allocates a private window for that OBU, and the OBU answers with a Vehicle Service Table (VST). Next, a GET_STAMPED is sent back and fourth. Here the RSE can read data from the OBU. The data can be details on the contract, account, the vehicle classification, the last transaction, etc. The OBU

can also ask the RSE for access credentials and the RSE ask the OBU to authenticate itself. In the next phases, first, a receipt is sent from the RSE to the OBU, then ECHO is used to track the OBU, and finally, an EVENT_REPORT is sent to close the transaction.

AutoPASS has added an extra security mechanism if a transaction fails or if a car does not have an OBU. In such case, a picture of the car's plate number is captured, and the bill is sent to the owner of the car.

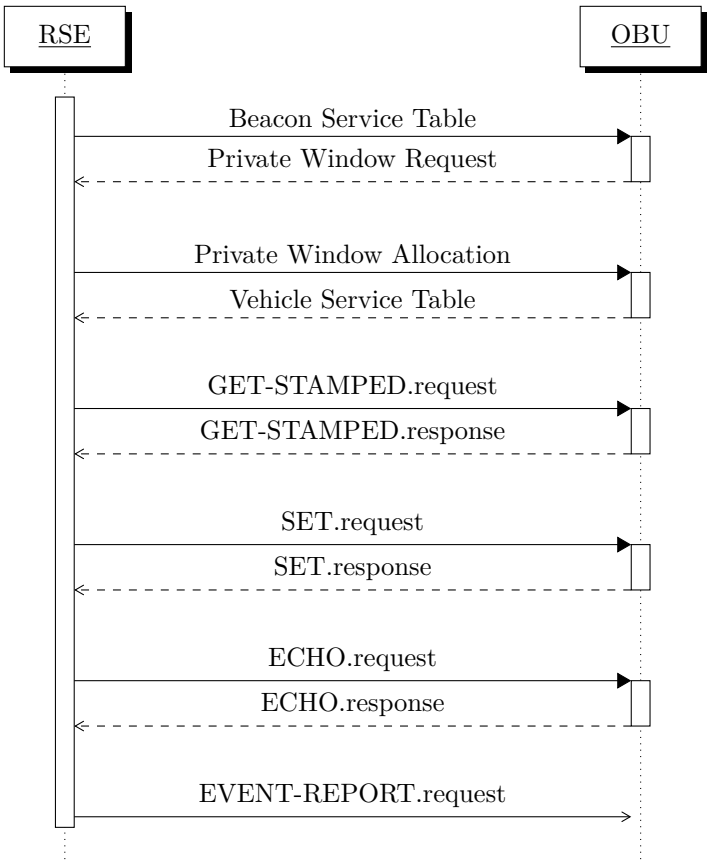

Figure 2.2: Message sequence diagram of a complete EFC transaction.

2.4 EFC Frame Contents

Chapter 5 in EN 12795 [CEN03a] specifies the frame format for DSRC transmissions. All transmissions are in frames where each frame consists of some specific fields. These fields are shown in Table 2.5.

Table 2.5: Specific fields in every DSRC-frame.

Description	**Octets**
Flag	1
Link Address Field	1-4
MAC Control Field	1
LPDU	1-128
Frame Check Sequence	2
Flag	1

All frames start and end with a flag. The flag is specified to be 0111 1110 (7E hex) [CEN03a]. The Link Address Field contains either a private Link Identifier (LID), a multicast LID or a broadcast LID. The MAC[1] Control Field is one octet long and is used to indicate whether the frame contains an LPDU, the direction of the transmission, to allocate and request public and private windows, and to specify the type of the LPDU. All LPDUs starts with one octet LLC Control Field which is used to designate command and response. The final part of the LPDU holds the data sent in the frame and can be up to 128 octets long. All of these fields are transmitted with the least significant bit first in each octet.

2.4.1 Frame Check Sequence

All frames include a 16-bit Frame Check Sequence (FCS). The FCS is calculated over the content of the Link Address Field, the MAC Control Field, and the LPDU. The generator polynomial used is $X^{16} + X^{12} + X^5 + 1$, with the initial value FFFF (hex), which is also called CRC-CCITT [CEN03a]. The 16-bit FCS is the one's complement of the resulting remainder. The FCS is transmitted with the coefficient of the highest term first.

2.5 AutoPASS Elements and Attributes

AutoPASS-OBUs holds 5 different elements, which all are identified with an Element Identifier (EID). Each element contains many attributes, which can be addressed by an Attribute Identifier (AID). The first element is used for EFC and is called *the AutoPass element*. The third element is used for public ITS applications. Element 2, 4 and 5 are reserved for future use and are not used today.

When constructing and sending messages from the RSE to the OBU, you can specify which element and attribute you want to read. All of the attributes listed in

[1]Please note that in this section MAC is an acronym for medium access control and not message authentication code.

Table 2.6 are read only, except receipt data 1 & 2 who can both be read and written. The communication channel between the OBU and the RSE is not encrypted, which means that all messages transmitted can be sniffed and recorded by a radio.

Table 2.6: Attributes in EID 1 stored in an OBU.

AttributeID	**Category**	**Value**	**Length** (in octets)
0	Contract	EFC-ContextMark	6
4		Contract Authenticator	5=1+4
16	Vehicle	Vehicle Licence Plate Number	17
17		Vehicle Class	1
18		Vehicle Dimensions	3
19		Vehicle Axles	2
20		Vehicle Weight Limits	6
22		Vehicle Specific Characteristics	4
23		Vehicle Authenticator	5=1+4
24	Equipment	Equipment OBU-ID	5=1+4
26		Equipment Status	2
32	Payment	Payment Means	14
33	Receipt	Receipt Data 1	28
34		Receipt Data 2	28

2.6 Security in EFC

2.6.1 Overview

EN 15509 defines two different levels of security, 0 and 1. Security level 0 require only authentication of the OBU and have no protection of user related data on the OBU. Security level 1 support security level 0, and has also access control, so the OBU can authenticate the RSE.

2.6.2 Security Related Data

Table 2.7 shows which security related data that is stored in an OBU supporting security level 0. It has eight different authentication keys of 8 octets, a key reference used in the calculation of the authentication value, and a random number of 4 octets received from an RSE. Table 2.8 shows additional data stored in the OBU when security level 1 is used. Opposite to authentication keys, there is only one access key. The access master key is 8 octets and is used for calculation of access credentials (AC_CR).

Table 2.7: OBU security related data in security level 0.

Name	**Length** (in octets)	**Remarks**
AuthenticationKey 1	8	Private
AuthenticationKey 2	8	Private
AuthenticationKey 3	8	Private
AuthenticationKey 4	8	Private
AuthenticationKey 5	8	Private
AuthenticationKey 6	8	Private
AuthenticationKey 7	8	Private
AuthenticationKey 8	8	Private
KeyRef	1	Reference to the AuthenticationKey used in computation of authenticator
RndRSE	4	Random number sent from RSE to OBU for computation of authenticator

Table 2.8: Additional OBU security related data added in security level 1.

Name	**Length** (in octets)	**Remarks**
Master Access Key	8	Private
AC_CR	1+4	Calculated using RndOBU and the AccessKey
AC_CR-KeyRef	2	Reference to the key generation and the diversifier for the computation of AC_CR-key
RndOBU	4	Random number sent from OBU to RSE for computation of access credentials

2.6.3 Access Credentials Calculation

If security level 1 is used, the RSE have to calculate access credentials. In the VST, sent from an OBU to an RSE, a random number (RndOBU) is included together with a key reference. The key reference is used to derive the Access Credentials Key (AcK) from a Master Access Credentials Key (MAcK). How the AcK is derived from the MAcK, is described in Figure 2.3.

a) Let VAL = AC_CR-KeyRef || AC_CR-KeyRef || AC_CR-KeyRef || AC_CR-KeyRef

b) Compute AcK = 3DES[MAcK](VAL)

Figure 2.3: Algorithm of how the Access Key is derived from the Master Access Key.

Triple-DES is used when calculating AcK, with the MAcK as key and VAL as input. After the AcK is calculated, the next step is to calculate access credentials. Access credentials is calculated as described in Figure 2.4. Figure 2.5 illustrates how DES is used to calculate access credentials. Note that DEA refers to the Data Encryption Algorithm, which is the algorithm of DES.

a) Let I = RndOBU || 00 00 00 00

b) Compute O = DES[AcK](I)

c) Let AC_CR = Sub(O,0,4)

Figure 2.4: Algorithm for calculating access credentials based on RndOBU and AcK.

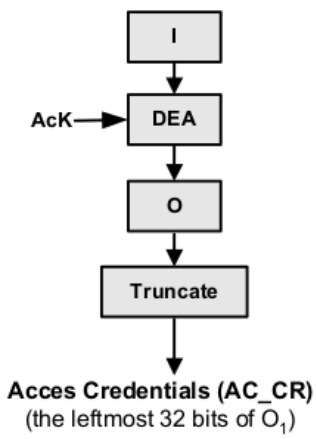

Figure 2.5: Illustration of access credentials calculation. Source: EN 15509 [CEN07].

The access credentials value is the first 4 byte (32 bit) of the output of a DES encryption with AcK as key and I as input. This value is sent from the RSE to the OBU in a GET_STAMPED.request. Then, the OBU evaluates the value received, and if OK, the communication continues. If not OK, no more messages are sent from the OBU to the RSE. A message sequence diagram of a transaction using access credentials is shown in Figure 2.6.

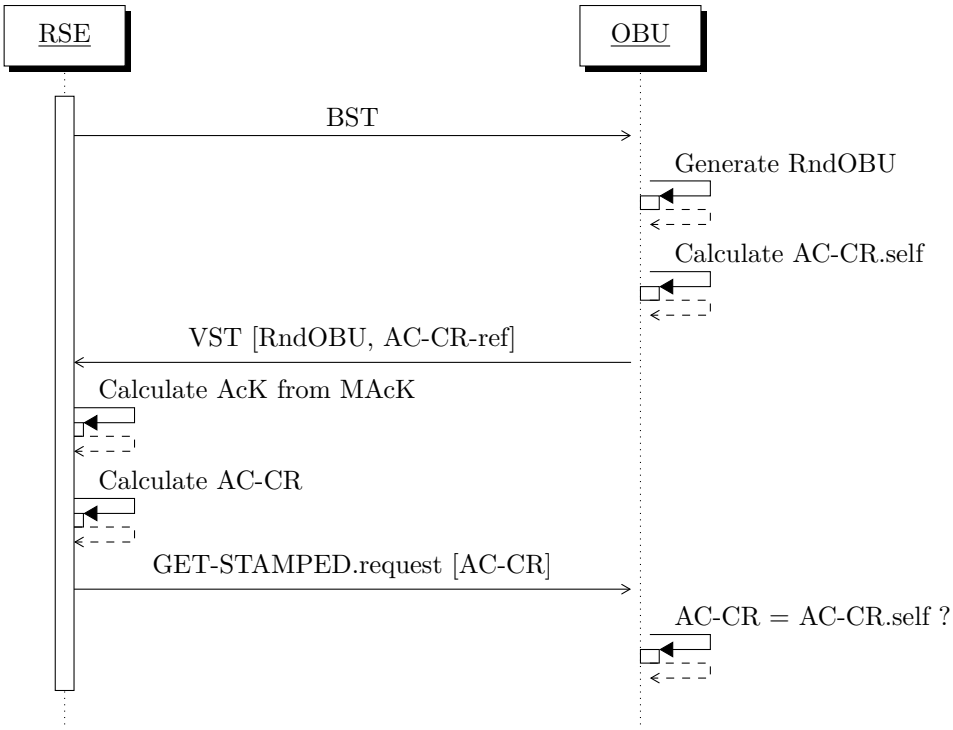

Figure 2.6: Message sequence diagram for access credentials.

2.6.4 Authenticator Calculation

An authenticator is used in both security level 0 and security level 1. The RSE include a random number (RndRSE) in the GET_STAMPED.request and challenge the OBU to calculate a MAC based on this number. First, an authenticator key is derived from one of the eight Master Authenticator Keys (MAuKs) stored in the OBU. Then the Authenticator Key (AuK) is derived from the MAuK according to the algorithm shown in Figure 2.7.

CompactPAN is calculated by taking the first half of Personal Account Number XORed with the second half. Contract Provider is known to both the OBU and the RSE. In Figure 2.10 a message sequence diagram explains the message flow for

obtaining a MAC from the OBU. The algorithm for calculating a MAC based on AttributeIDList, RndRSE, and AuK is shown in Figure 2.8. Figure 2.8 illustrates how DES in CBC-mode is used to calculate an authenticator (MAC). The leftmost 32 bits of the output is sent as the MAC.

a) Let CompactPAN = Sub(PAN,0,4) XOR Sub(PAN,4,4)

b) Let VAL = CompactPAN || Contract Provider || 00

c) Compute AuK = 3DES[MAuK](VAL)

Figure 2.7: Algorithm for deriving Authenticator Key (AuK) from Master Authenticator Key (MAuK).

a) Let M = AttributeIDList(Payment Means) || RndRSE || 00 00

b) Let D1 = I1 = Sub(M,0,8), D2 = Sub(M,8,16), and D3 = Sub(M,16,24)

c) Compute O1 = DES[AuK](I1)

d) Compute O2 = DES[AuK](I2 = O1 XOR D2)

e) Compute O3 = DES[AuK](I3 = O2 XOR D3)

f) Let MAC = Sub(O3,0,4)

Figure 2.8: Algotrihm for calculating a MAC based on AttributeIDList, RndRSE, and AuK.

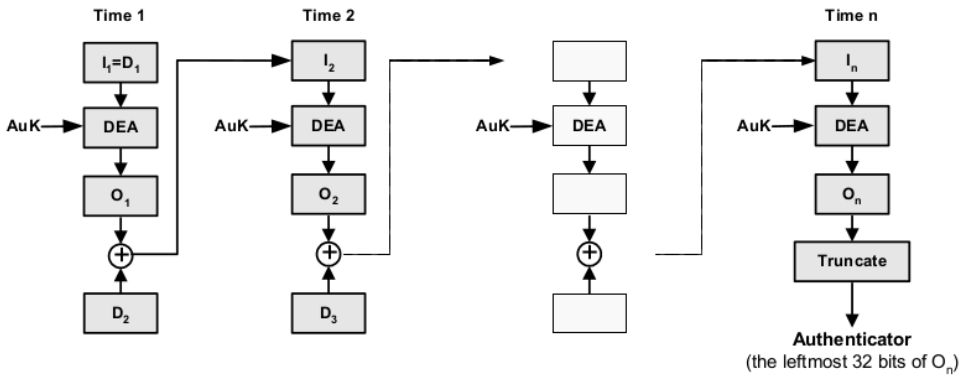

Figure 2.9: Illustration of calculation of MAC using DES in CBC-mode. Soruce: EN 15509 [CEN07].

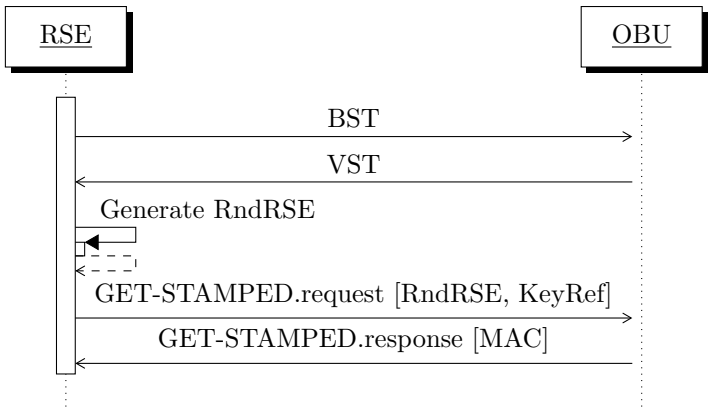

Figure 2.10: Message sequence diagram when an RSE authenticates an OBU.

2.7 Data Encryption Standard

DES is used in calculations of both MACs and access credentials, and is a symmetric-key algorithm used for encryption of data. It was developed in the early 1970s and was first published in 1975 by International Business Machines Corporation (IBM). In 1977 DES was standardized by the National Security Agency (NSA), and was in 1977 published as an official Federal Information Processing Standard in the United States [NIS99].

DES is a block cipher, which means that the key is applied to a block of plaintext rather than one bit or the whole plaintext at the same time. In simple words, DES takes 64 bit of plaintext and returns 64 bit of ciphertext. The key is also 64 bit long,

but only 56 bit is used, as 8 of the 64 bits are used as parity bits. The parity bits are used to ensure that each byte is of odd parity.

First, the plaintext is split into blocks of 64 bits and later split into two smaller blocks of 32 bit. Then 16 rounds, or iterations, are applied to these 32-bit blocks where 16 different keys (subkeys) are used. The subkeys are 48 bits long and are derived from the 56-bit key through a key schedule. Before and after the 16 rounds, an initial and a final permutation is done to the block. Neither the initial nor the final permutation has cryptographic significance.

All block ciphers have a mode of operation. A mode of operation describes how all the 64-bit blocks are linked together and are done to provide confidentiality and authenticity. In FIPS publication 81 [NIS80], DES's modes of operation are described. These are Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB).

In EFC, DES use CBC-mode, with an initialization vector (IV) of zeros. Figure 2.11 illustrates CBC. First, 64-bit plaintext is XORed with an initialization vector, then, the DES algorithm is applied. The next block of plaintext is always XORed with the output of the previous DES encryption.

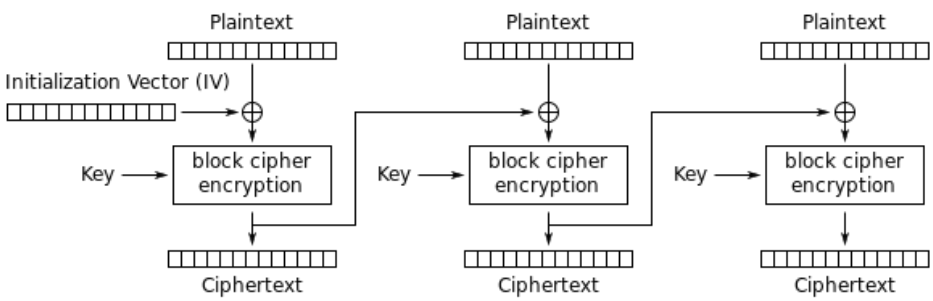

Figure 2.11: Illustration of Cipher Block Chaining mode. Source: [Wik].

Today, DES is considered insecure [NIS], mainly due to the short key. In 1998 Triple DES was published. Triple DES has a key of 168-bit, three times as much as single DES. The first third of the key is used for one DES encryption, the second for a DES decryption, and finally, the third for a new DES encryption. In EFC Triple DES is used to derive both authenticator and access credentials keys from their master key. When calculating MACs and access credentials, single DES is used.

Chapter 3
Making a Customizable RSE

The objective for the first part of this thesis is to make a customizable RSE. A customizable RSE is a radio that can both send and receive messages to and from an OBU. Without this, it would not be possible to communicate with an OBU, and not be able to attack MACs sent from OBUs. This chapter describes how a customizable RSE is built, and the requirements set for such an implementation.

3.1 Methodology

3.1.1 Literature Review

Before building an RSE, it is important to study relevant literature and to find related work. The DSRC standards EN 12253 [CEN04a], EN 12795 [CEN03a], and EN 12834 [CEN03b], defines the requirements for an RSE, and are therefore important. Especially, the physical layer of DSRC is studied, which is presented in Section 2.2.1.

Related Work

In 2016, Jonathan Hansen wrote his Master's thesis at Norwegian University of Science and Technology (NTNU) with the title "*Wireless USRP Test-bed for DSRC Applications*" [Han16]. He tried to build a customizable RSE, but unfortunately, he did not completely succeed. Most of his work was based on Einar Thorsrud's thesis, "*Programvaredefinert Radio - Mulige Hyllevareløsninger for DSRC-anvendelser*," from 2009 [Tho09].

Thorsrud also aimed to build a customizable RSE, by using a USRP and GNU Radio. His implementation worked with test files, but it was not able to communicate with an OBU. Thorsrud used a USRP 1 with motherboard version 4.5 and GNU Radio version 3.2. Because neither the hardware nor the software version was the same from Hansen as Thorsrud, Hansen had to do some modifications.

3.1.2 Hardware and Software

After studying both Hansen's and Thorsrud's program, whom both used a USRP and GNU Radio, it was an easy choice to choose the same hardware and software. Even though neither of them succeeded in making an RSE, the value of their work will save much time.

USRP

Hansen used a USRP N200 with a CBX daughterboard, and the very same USRP will be utilized in this thesis. A USRP is a product in the family of SDRs. SDRs are radios where most of the complex signal handling involved are placed in software. By doing this, a radio can be reprogrammed to fit a lot of different tasks [ARR]. USRPs are produced by Ettus Research and are designed for radio frequency applications up to 6 GHz and support multiple antenna systems [Etta]. Currently, Ettus Research provides four different categories of USRPs, X series, networked series, bus series and embedded series. In addition to the USRPs, a lot of various accessories, RF daughterboards, antennas, and cables are provided.

USRP N200 is part of the USRP Networked Series, and comes with a Gigabit Ethernet interface, allowing data streams up to 50 MS/s. As mentioned, a CBX daughterboard is attached to the motherboard. The daughterboard provides a full-duplex RF front end and covers a frequency band from 1.2 GHz to 6 GHz [Ettb]. Together with the daughterboard, the USRP also need antennas in the wanted frequency band. Two omnidirectional vertical antennas, called VERT2450, were used by Hansen. These cover frequencies from 2.4 to 2.5 GHz and 4.9 to 5.9 GHz.

GNU Radio

GNU Radio is an open-source development platform for SDRs. It provides signal processing blocks and can be used both with or without external RF hardware. GNU Radio has pre-made filters, channel codes, equalizers, demodulators, decoders, and many other blocks used in signal processing systems. It is also possible to create new blocks if needed. GNU Radio's wiki page provides a lot of guides, tutorials, and examples of how to write a new block. Blocks can be written in either Python or C++.

There also exist a graphical user interface, called GNU Radio Companion (GRC). In GRC signal processing applications can be created by drag-and-drop blocks together. GRC makes it very much easier to create applications, and it is also possible to write own blocks for GRC.

GNU Radio also contains the GNU Radio USRP Hardware Driver (UHD) package, which is the interface to the UHD library for sending and receiving data between

USRPs. The two most important blocks in this package are USRP Source and USRP Sink, which act as a receiver/transmitter [Radc].

Both Hansen and Thorsrud used GNU Radio, and Hansen's program was attached with his thesis, so this can be imported and used directly. GNU Radio and UHD are together installed with the "*build-gnuradio*" script made by Marcus Leech. By running the command below, the installer "*build-gnuradio*" is downloaded, made executable, all the dependencies are installed, and the latest version of both UHD and GNU Radio is downloaded from Git [Radb].

```
$ wget http://www.sbrac.org/files/build-gnuradio && chmod a+
    x build-gnuradio && ./build-gnuradio
```

As an option to GNU Radio, National Instruments has developed a program called LabVIEW. LabVIEW is a system engineering software mainly used for visualizing data from any I/O device, such as a USRP. Without any experience in either GNU Radio or LabVIEW, GNU Radio was chosen because both Thorsrud and Hansen had used it.

Computer

An OptiPlex 7040 computer, with 64-bit Linux Ubuntu 16.04 LTS, is attached to the USRP by Gigabit Ethernet. The computer has an Intel Core i7-6700 CPU at 3.40 GHz with four cores and eight threads. The main memory is 32 GB.

3.2 Radio Frequency Identification Reader

To get started before the DSRC implementation, Radio Frequency Identification (RFID) programs was studied. RFID uses electromagnetic fields to identify RFID-tags. Several passive RFID-tags operating in the frequency band around 900 MHz, was available for testing. FasTrak, the EFC system used in California in the United States, uses RFID technology near 915 MHz to read data from an OBU.

After browsing the web for suitable RFID applications, "*Gen2 UHF RFID Reader*" [Kar16] by Nikos Kargas, was chosen. Jonathan Hansen also studied this program in his thesis [Han16]. This program matches the hardware that will be used for building a customizable RSE later on, pretty well. Kargas' hardware was a USRP N200 with an RFX900 daughterboard and two circularly polarized antennas with 7dbi gain. The hardware utilized in this thesis for the RFID-reader was a USRP N200 with an SBX daughterboard and two linear polarized antennas with 3dbi gain. Hopefully, without the same hardware, this program could work to read some RFID-tags.

Kargas' program was downloaded from GitHub [Kar16], and GNU Radio and UHD was installed with the "*build-gnuradio*" script described above. When Hansen studied this program, he early ran into problems reading RFID-tags. The same problem as Hansen early occurred, where the program ran just fine with test files, but with real-time execution, no tags were read.

First, the installation of the program was checked. Ensuring that the program is correctly installed is important and after re-installing the program several times, and even with a clean install of Ubuntu, no tags were read. Nothing solved the problem, so Kargas was contacted by email. He taught the antennas was causing the problem and suggested to try with circularly polarized antennas.

As a result of this, Nils Torbjörn Ekman and Terje Mathiesen at NTNU was contacted. Mathiesen said that it should be possible to read RFID-tags with two linear polarized antennas, by just positioning them correctly. The angle between the two linear antennas has to be 90 degrees for the program to work. Also changing the TX gain in the program from 0 to 20 could solve the problem. With this configuration done, the program was able to read RFID-tags with a range of about 5 cm.

Unfortunately, the read-range is very short, and the tag must be held almost between the antennas. Although the short range, this is not a big problem as the program only will be used for testing. The output of the program, when trying to read some RFID tags, is shown in Figure 3.1. Kargas' program only shows 8 of the 128-bit Electronic Product Code (EPC), so after modifying the program, the program printed out the entire 128-bit EPC. Figure 3.2 shows a MATLAB plot of the signals when the program read the same RFID-tag four times in a row. Kargas added an example file with an explanation, which is shown in Figure 3.3.

```
--------------------------
| Number of queries/queryreps sent : 1000
| Current Inventory round : 1001
--------------------------
| Correctly decoded EPC : 580
| Number of unique tags : 3
| Tag ID : 52  Num of reads : 12
| Tag ID : 53  Num of reads : 515
| Tag ID : 9e  Num of reads : 53
--------------------------
```

Figure 3.1: The output from the RFID reader, when reading three RFID tags. The reader read three unique tags with ID 52, 53, and 9e.

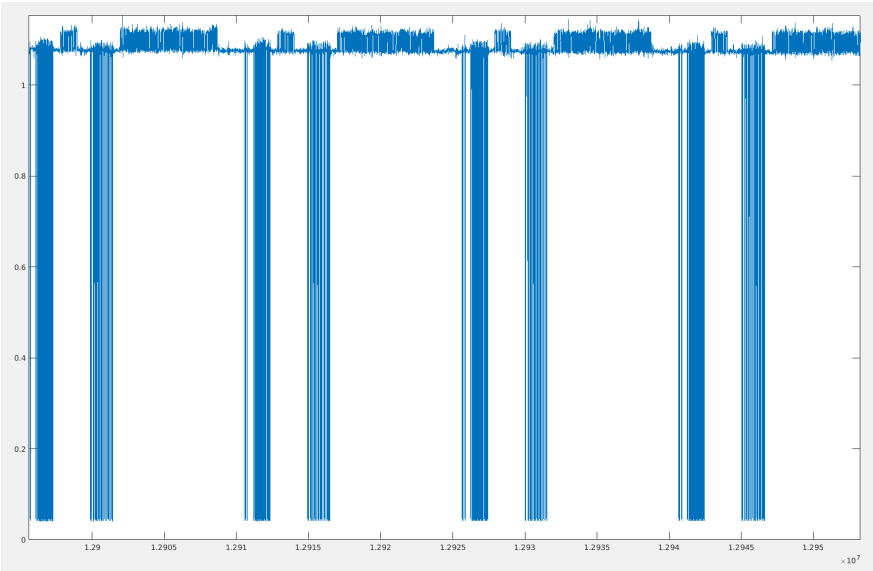

Figure 3.2: MATLAB graph of a RFID-tag read four times.

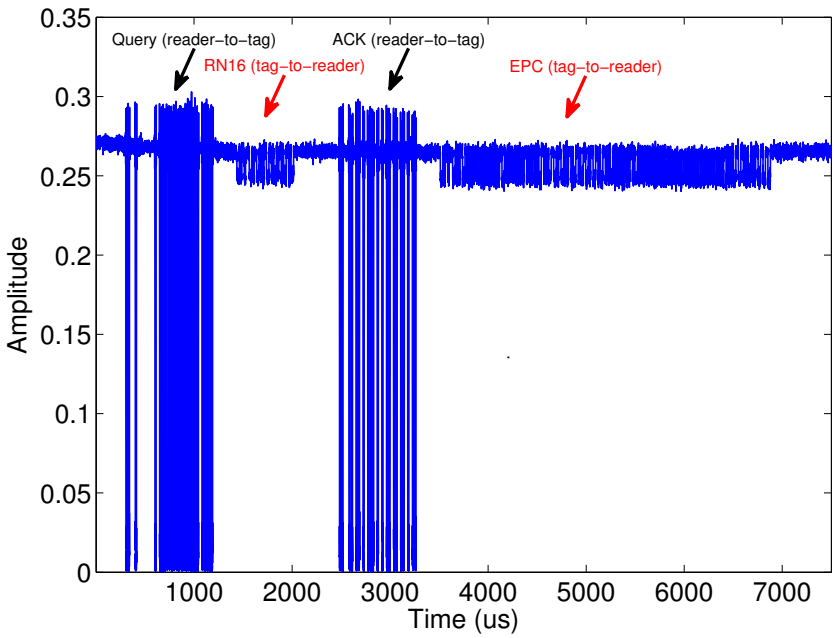

Figure 3.3: Kargas' sample file with description. Source: [Kar16].

This result is promising for the implementation of DSRC. Although the range for RFID-tags was very short, this may be increased by changing the gain values for RX and/or TX. Also changing antennas may solve the problem.

3.3 Building an RSE

3.3.1 Hansen's Program

When building a customizable RSE, I chose to try to improve and to find the mistakes in Hansen's program from 2016. His program was downloaded from BIBSYS Brage [Han], and the blocks were loaded into GRC.

Hansen included three programs together with his master thesis, a DSRC-transmitter, a DSRC-receiver, and a DSRC-transceiver. Initially, he only made a transmitter and a receiver, but because he only had one USRP, and GNU Radio do not allow two programs to run simultaneously, he made a single program with the blocks from both the transmitter and the receiver. In the following subsections, on how the transmitter and receiver are implemented, is described separately.

3.3.2 Transmitter

In section 4.1.4 in Hansen's thesis [Han16], he describes how he built the DSRC program. Hansen recreated Thorsrud's program in GRC, but due to a different version of GNU Radio, some modifications had to be done. After carefully reading both Hansen's and Thorsrud's thesis, the recreation done by Hansen was verified to be correct. Also, Hansen added a Graphical User Interface (GUI) Time Sink, to visualize the transmitted signal.

The final GRC flow chart for the transmitter is shown in Figure 3.4. The transmitter consists of four blocks; a vector source, a pulse shaper, a short-to-complex converter, and the UHD block, USRP Sink. The vector source takes a binary string as input and sends the vector to the pulse shaper block. The pulse shaper block is not a standard GNU Radio block but was made by Hansen. This block does both FM0 encoding, modulation and pulse shaping according to the specifications in EN 12253 [CEN04a]. The next block is just a converter, which converts the signal from data type short to complex. Finally, the data is sent to the USRP Sink. Also, a file source is added, which can replace the vector source, as it may be more practical to read data from a file.

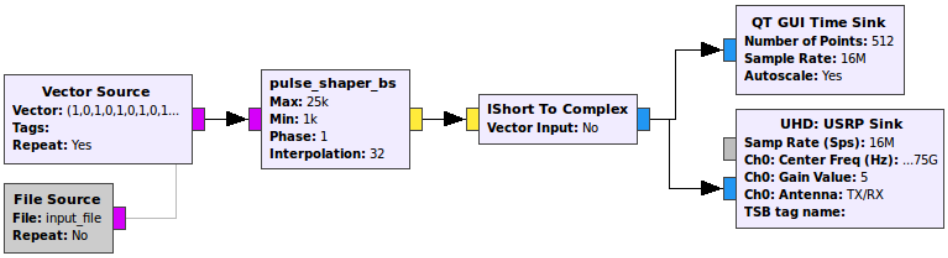

Figure 3.4: Signal processing blocks from the GRC transmitter flow chart.

Some of the blocks mentioned above take variables to define their behavior. Figure 3.5 shows these variables, and most of them are copied from Hansen's program. From Hansen's program, some of these have been changed. Hansen set the frequency to 5.8 GHz, but this is changed to 5.7975 GHz. The reason for this change is explained in the next subsection. The gain variable is increased from 1 to 5, to be sure that the signal sent from the USRP is strong enough for the OBU. The other variables are not changed, and have been verified to fulfill the specification.

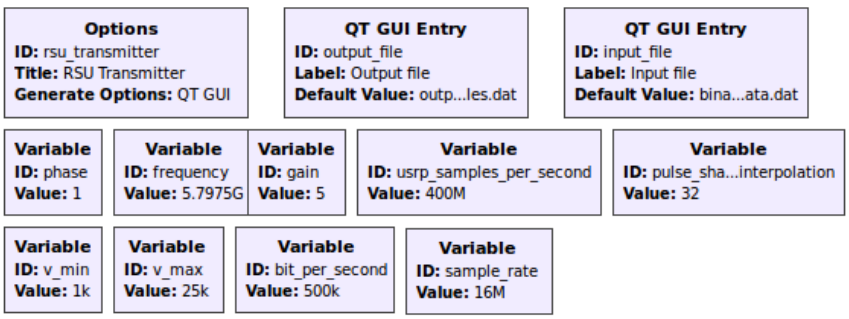

Figure 3.5: Variables from the GRC transmitter flow chart.

As earlier explained, both Hansen and Thorsrud failed to build a working RSE. Their programs are well described, and most of their work seems to be correct. However, some small errors are found in the transmitter. In the following, the findings and corrections are presented.

Frequency

As presented earlier, the frequency of the transmitter is changed from 5.8 GHz to 5.7975 GHz. In Table 2.2, item number D1 specifies two downlink channels, one at

5.7975 GHz and one at 5.8025 GHz. Changing the frequency is a small change but can affect the result of the transmitter. EN 12253 also specifies that the minimum frequency range for an OBU is between 5.975 and 5.815 GHz, so with a transmitter with frequency at 5.8 GHz, the OBU should be able to receive the signal. Even though both frequencies should work, the frequency was changed to 5.7975 GHz.

Antennas

As a result of the antenna-problem when reading RFID-tags, new antennas was bought. The two VERT2450 antennas were replaced by left-hand circular polarized antennas for 5.8 GHz from SpiroNET. These antennas fulfill the specification of item number D5 in Table 2.2.

BST/Vector Source

Hansen does not mention in his thesis how he constructed the BST. In the program attached with his thesis, the input to the vector source was only an alternating sequence of 1's and 0's. Section 7.4.2.1.1 in EN 12795 [CEN03a] describes how a received frame shall be considered. In Figure 3.6 this section is almost directly copied.

7.4.2.1.1 Validity of frame

The MAC sublayer shall inspect all received frames to assess their validity. A received frame shall be considered valid if:

(a) the frame is correctly delimited by start and end flags according to 5.1; and

(b) the frame contains a number of bits corresponding exactly to an integer number of octets; and

(c) the frame contains a valid link address field according to 5.2 containing the private LID of an SAP; and

(d) the frame contains a MAC control field according to 7.3.2; and

(e) the frame does not consist of too many octets; and

(f) the frame contains a valid FCS field according to 5.5.

If the frame received is not valid it shall be discarded. If a non-valid frame is received in a private window, reallocation may be undertaken according to 7.5.2.2.3.

Figure 3.6: Validity of frame, taken from 7.4.2.1.1 in EN 12795 [CEN03a].

As stated in Figure 3.6, there are many requirements to be fulfilled for a frame to be considered valid. A valid BST was received on request from Q-Free, to be sure that this part was correct. The valid BST is shown in Table 3.1. A more detailed description of how BST messages are constructed is described in Section 4.2.2.

Thorsrud, on the other hand, used a preamble consisting of 16 1's, which he repeated 312 times before sending the BST. Then, he had a BST of 68 bits. He has not mentioned how this BST is constructed or what it means, but by doing a quick check against the requirements in Figure 3.6, it is easy to see that this is not a valid frame/BST. It has not a start or a stop flag (0111 1110), and the number of bits does not correspond exactly to an integer number of octets. Without knowing what the bits indicate, it is hard to say if it fails on more of the requirements, but failing at least two is more than enough not to be considered as a valid frame.

```
self.bst_sequence = (
  0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, \
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, \
  0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, \
  0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, \
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 )
```

Figure 3.7: Screenshot of Thorsrud's BST, taken from A.1 in Thorsud's thesis [Tho09].

Table 3.1: Example of a valid BST, used in the transmitter, received from Q-Free.

Octet	**Attribute/Field**	**Bit**	**Hex**
1	Preamble	1010 1010	AA
2		1010 1010	AA
3	Start flag	0111 1110	7E
4	Broadcast LID	1111 1111	FF
5	MAC control field	1010 0000	A0
6	LLC control field	0000 0011	03
7	Fragmentation header	1001 0001	91
8	BST {	1000	
	Option indicator	0	
		000	80
			Continued on next page

Table 3.1 – continued from previous page

Octet	Attribute/Field	Bit	Hex
9	BeaconID.ManufacturerID	0000 0000	00
10		0011 0	
		000	30
11	BeaconID.IndividualID	0000 0110	06
12		0000 0011	03
13		1110 1100	EC
14		0000 1110	0E
15	Time	0111 0101	75
16		1101 0101	D5
17		0010 1000	28
18	Profile	0000 0000	00
19	MandApplications	0000 0001	01
20	Option indicator	0000 0001	01
21	ProfileList }	0000 0000	00
22	Frame check sequence	0100 1100	4C
23		0011 1110	3E
24	Stop flag	0111 1110	7E

GNU Radio Warnings

When running the transmitter in GRC with the USRP attached to the computer, a UHD warning appears in the console window. As seen in Figure 3.8, the requested TX sample rate is not supported by the hardware. The warning disappears if the sample rate is changed from 16 MSps to 16.66667 MSps. If this is a real problem or not, is not clear. The difference between the numbers is so small that it hopefully will not cause any problem.

Figure 3.8 does not only shows the UHD warning, but it also shows a lot of U's. These are printed out to the console constantly when the program is running. U's indicates *underrun* and occurs when the application does not supply samples fast enough to the USRP. Underrun may be a big problem, as the USRP Sink do not receive data fast enough. In GNU Radio's mailing list forum, a suggestion is to lower the sample rate to solve the problem [Radd]. By lower the sample rate to 4 MSps,

the U's disappears, but then the sample rate is very low, causing the signal not to be as sharp as it should.

After some investigation, it became apparent that it is the pulse shaper block which is causing the problem. It is too slow, and by using a process viewer, it is clear that one of the CPU cores is not capable of working fast enough. A screenshot of *htop*[1] is shown in Figure 3.9. It is not an easy fix to remove this problem. The task of the pulse shaper block can not be done in parallel because it handles a data stream. A solution that may solve, or at least decrease, the problem, would be to re-write the pulse shaper block from Python to C++. GNU Radio's wiki page says that when performance is important, GNU Radio blocks should be written in C++ [Rada]. Because of lack of both time and experience in C++, this has not been done in this thesis but is highly recommended for future work on the customizable RSE.

```
UHD Warning:
  The hardware does not support the requested TX sample rate:
  Target sample rate: 16.000000 MSps
  Actual sample rate: 16.666667 MSps
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
```

Figure 3.8: Screenshot of the warnings when running the transmitter in GRC.

```
1  [|||            9.2%] 5  [|||           11.3%]
2  [||||          11.9%] 6  [|||           11.8%]
3  [|||           13.6%] 7  [|||           10.0%]
4  [||||||||||||100.0%] 8  [              0.0%]
Mem[||||    1.92G/31.3G] Tasks: 116, 329 thr, 118
Swp[            0K/31.9G] Load average: 1.61 0.87
```

Figure 3.9: Screenshot of *htop* when running the transmitter. Here core number 4 is running at 100 % and is not able to work any faster. This causes a bottleneck in the transmitter.

3.3.3 Receiver

Similarly to the transmitter, Hansen's receiver is downloaded and studied. The receiver is a bit more complex than the transmitter, and consist of even more blocks. The GRC flow chart is shown in Figure 3.10, and the variables are shown in Figure 3.11.

[1] http://hisham.hm/htop/

The first block is a USRP Source block, which receives data from the USRP RX2 antenna. Then, a frequency filter and an MPSK receiver are added. The Frequency Xlating FIR Filter block performs frequency translation, channel selection, and decimation. The MPSK Receiver block performs carrier frequency and phase locking for PSK modulated signals. Then two blocks are added to convert the data type and recover the binary bits before the nrzi_to_nrz_bb block is applied. This block converts from non-return-to-zero-inverted encoding to non-return-to-zero encoding. Finally, the result is added to a file.

The only parameters that have been changed from Hansen's receiver are the frequency from 5.8 to 5.7975 GHz and the gain from 1 to 5.

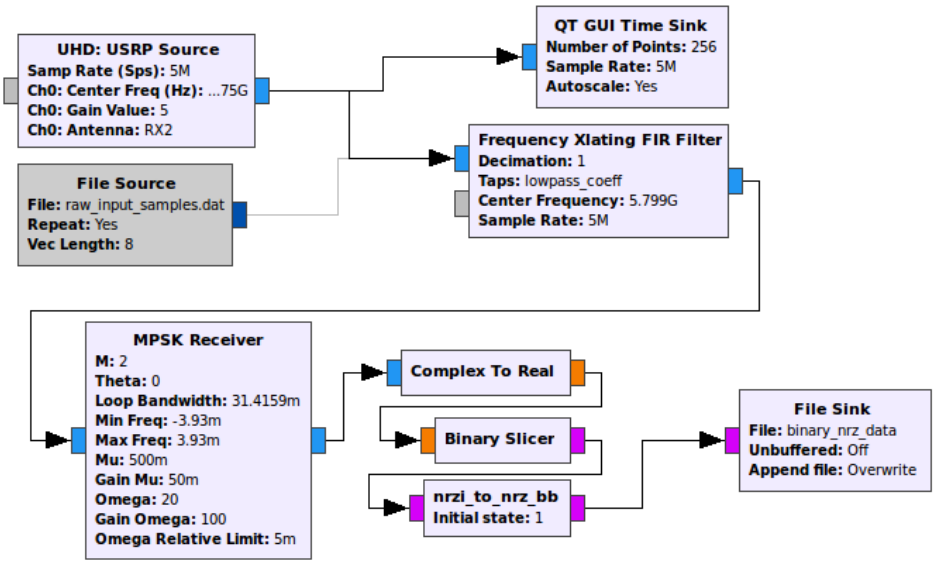

Figure 3.10: Signal processing blocks from the GRC receiver flow chart.

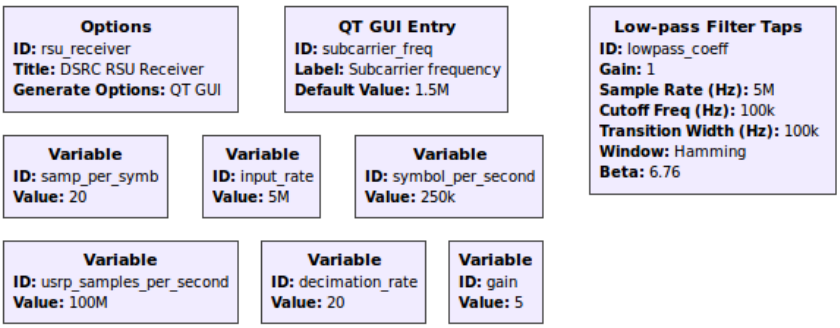

Figure 3.11: Variables from the GRC receiver flow chart.

3.3.4 Results

By combining the blocks from the transmitter and the receiver together, you will get a transceiver. A transceiver can both transmit and receive a signal, and in this case, it can do it simultaneously. By running the transceiver, two windows with the output of the GUI Time Sink for both the transmitter and the receiver is shown. Figure 3.12 shows the output for the transmitter part of the transceiver. A part of the BST from Table 3.1 is shown as correctly FM0 encoded. The output seems correct according to how FM0 encoded signals should be. The bits in Figure 3.12 are: 1010 0000 0000 0011 1001 0001 1000 0000. By looking in Table 3.1, these bits can be found from octet 5 to 8.

The output in the GUI Time Sink for the transmitter seems correct, but it is hard to test if it works. One of Hansen's problems was that he was not able to test or verify the program. A normal OBU gives no visual feedback, and you are therefore dependent that both the transmitter and the receiver is working to make sure the program works. Back in 2009, Thorsrud had an OBU with LED. With such an OBU, it is easy to see if the transmitter is working. After contacting Q-Free, an OBU with LED was obtained.

The LED-OBU has two different diodes, a yellow and a red. The yellow light indicates that the OBU wake up as a result of receiving a modulated signal on the right frequency. The red light indicates that the OBU has received a valid BST and that it responds. Running the transmitter with the LED-OBU in front of the USRP, the yellow light was indicating that the OBU woke up. Unfortunately, the red light remained off, indicating that a valid BST was not received.

Without any answer from an OBU to the USRP/RSE, it is very hard to test the receiver. The output of the GUI Time Sink for the receiver, when both the

transmitter and the receiver is running together, is shown in Figure 3.13. The output looks like noise, and it is not possible to decode the signal. Without any possibilities to test the receiver any further, it was difficult to find the errors in the program.

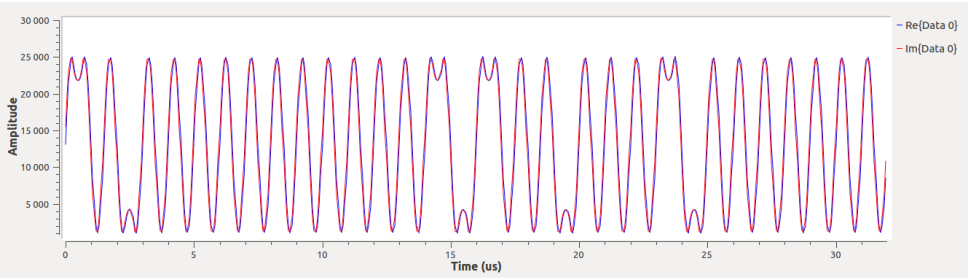

Figure 3.12: The output of the GUI Time Sink for the transmitter. Bits correctly FM0 encoded shown here are "1010 0000 0000 0011 1001 0001 1000 0000."

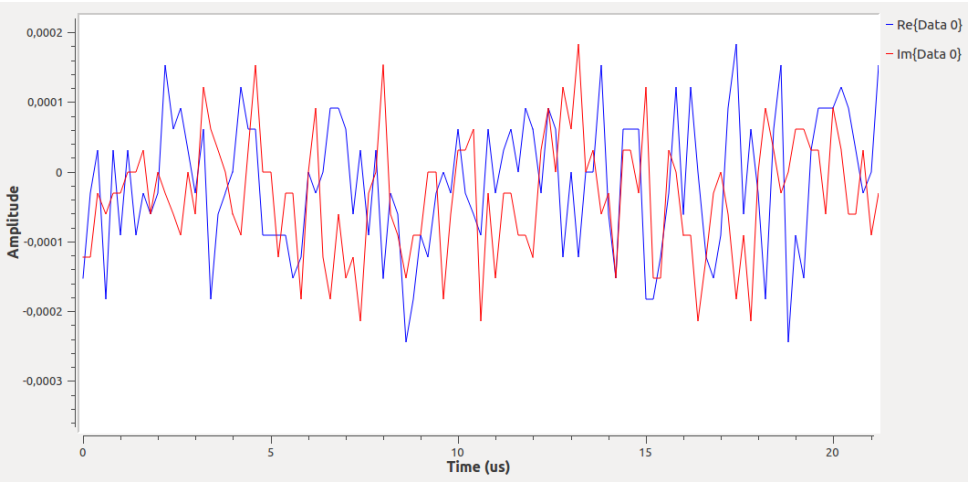

Figure 3.13: The output of the GUI Time Sink for the receiver.

3.4 RSE624 - Desktop DSRC Transceiver

A desktop DSRC transceiver, called RSE624, was bought from Q-Free to be able to communicate with an OBU within this semester. With this reader, DSRC-messages can be constructed down to bit-level, which makes it very practical for the task of this thesis. The reader was connected to a computer using USB. In the user manual [Q-F16], some serial port configuration parameters were specified. These parameters are shown in Table 3.2. With the Python library *serial* [Lie], all these parameters were set, and the reader was successfully connected to the computer.

Table 3.2: RSE624 configuration parameters [Q-F16].

Parameter	Value
Baud rate	57600
Data bits	8
Parity	Even
Stop bits	2

In Figure 3.14 some Python code is included. Here the library *serial* is imported. The serial port is configured to the variable "ser" with the parameters from Table 3.2. Timeout is set to 0.1 seconds, to ensure that the reader waits at least 0.1 seconds when trying to read messages from an OBU. At last, time.sleep(0.1) is added, to avoid that the reader sends out the BST before it is ready to send out messages.

```
import serial
import time

try:
  ser = serial.Serial('/dev/ttyUSB0', //
    baudrate=57600, //
    timeout=0.1, //
    bytesize = serial.EIGHTBITS, //
    parity=serial.PARITY_EVEN, //
    stopbits=serial.STOPBITS_TWO)
  print "Connected to " + ser.name
except:
  sys.exit("Error connecting device")

time.sleep(0.1)
```

Figure 3.14: Python-code for setting up and configuring RSE624.

Together with the user manual, an interface protocol [Q-F01] is included with RSE624. This protocol specifies how DSRC frames are constructed in the reader. Table 3.3 and Table 3.4 are almost directly copied from the interface protocol [Q-F01]. These tables show how the messages are constructed. Every message transmitted from the RSE starts with *F, and every message received starts with *I.

Table 3.3: RSE624 interface protocol for transmitting DSRC-data [Q-F01].

Character No.	**Value**	**Comment**
1-2	*F	DSRC Transmit
3-4	Config	Bit 0: Keep alive (0=on/1=off) Bit 1-2: Keep alive interval (00=32ms, 01=60ms, 10=100ms, 11=11ms) Bit 3-6: NA Bit 7: Protocol (0=DSRC, 1=LDR)
5-6	Length	# of data bytes + CRC(2) + stop flag(1) + 3 bytes
7-(6+2N)	Data	DSRC/LDR frame. N = data bytes
(7+2N)-(10+2N)	FCS/CRC	CRC16 computed on Data
(11+2N)-(12+2N)	Stop flag	7E (hex)

Table 3.4: RSE624 interface protocol for receiving DSRC-data [Q-F01].

Character No.	**Value**	**Comment**
1-2	*I	DSRC Receive
3-4	Length	# of data bytes + CRC(2)
5-(4+2N)	Data	DSRC/LDR frame. N = data bytes
(5+2N)-(8+2N)	FCS/CRC	CRC16 computed on Data

3.5 Recordings With Gqrx

It is not easy to fully understand a physical signal by just reading the specifications. Therefore I was recommended by Ulf Bertilsson to do some recording of a real RSE, to visualize the signal. With a laptop, a USRP and Gqrx, it is possible to visualize the received signal in a waterfall. Gqrx is an open source SDR receiver, build on GNU Radio and Qt [Gqr]. Together with a USRP B200mini and a MacBook Pro, some recordings with Gqrx were done.

In Figure 3.15, a recording of the signal received when sitting in a car who drives pass an RSE is shown. The car's speed is around 40 km/h, so the transaction between the OBU (in the car) and the RSE is done in under a second. Because of much noise and the danger of standing on a road when trying to receive signal, the next recordings are taken from a transaction between the Desktop DSRC Transceiver, RSE624, and an OBU.

In Figure 3.16, three different scenarios are shown. First, in the red frame, a real transaction between an OBU and the RSE is captured. How this transaction is built is explained in Chapter 4. In the red frame, it can be seen that the carrier frequency is 5.7975 GHz and the sub-frequencies are at 5.796 and 5.799 GHz. From Table 2.3, this matches the specification well, with sub-carrier frequency 1.5 MHz away from the carrier frequency. Without having it verified, or finding anything about it in the specification, a hypothesis could be that the RSE sends data approximately 0.5 MHz away from the carrier frequency. So the beam at the carrier frequency (5.7975 GHz) is used to wake up the OBU and give it the energy to work, data from RSE to OBU are sent at 5.797 and 5.798 GHz, and data from OBU to RSE are sent at 5.796 and 5.799 GHz.

In the green frame, a DSRC message is transmitted in repeat, containing 90 % of 0's, but still fulfilling the requirements of a valid frame as explained in Figure 3.6. In the white frame, a message with 90 % if 1's is sent from the RSE to an OBU. From the screenshot, it may seem that the OBU gives some answer to the message, even though the message is not a valid BST, but a valid DSRC message.

Also in Figure 3.17, three different scenarios are shown. Again, in red, a real transaction, where the sub-frequencies are very clear. In green, the RSE sends out the same BST over and over again, but the OBU has blocked that RSE (more about this in Section 4.2.2). Here the signals at 5.797 and 5.798 GHz are clear (data from RSE to OBU), but the signals at 5.796 and 5.799 GHz are very weak. Why they are weak, and not completely gone, is still a question, but maybe the OBU responds with a very short message. In the white frame, the RSE624 is turned OFF, and the transmitter explained in Section 3.3.2 is turned ON. Observe that the carrier frequency, at 5.7975 GHz, seems correct, but the sub-frequencies at 0.5 MHz and 1.5 MHz away from the carrier is not there.

It is hard to find some clear conclusion from this, but it is still valuable. This recording can be used to compare the RSE built in GNU Radio with a working RSE, as RSE624.

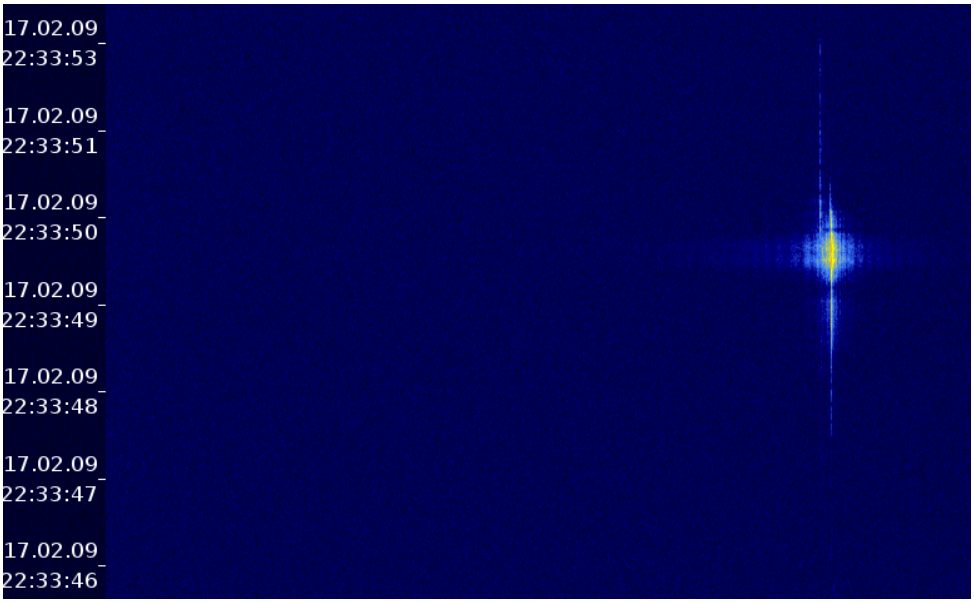

Figure 3.15: The signal received when driving pass an RSE, from 9th February 2017.

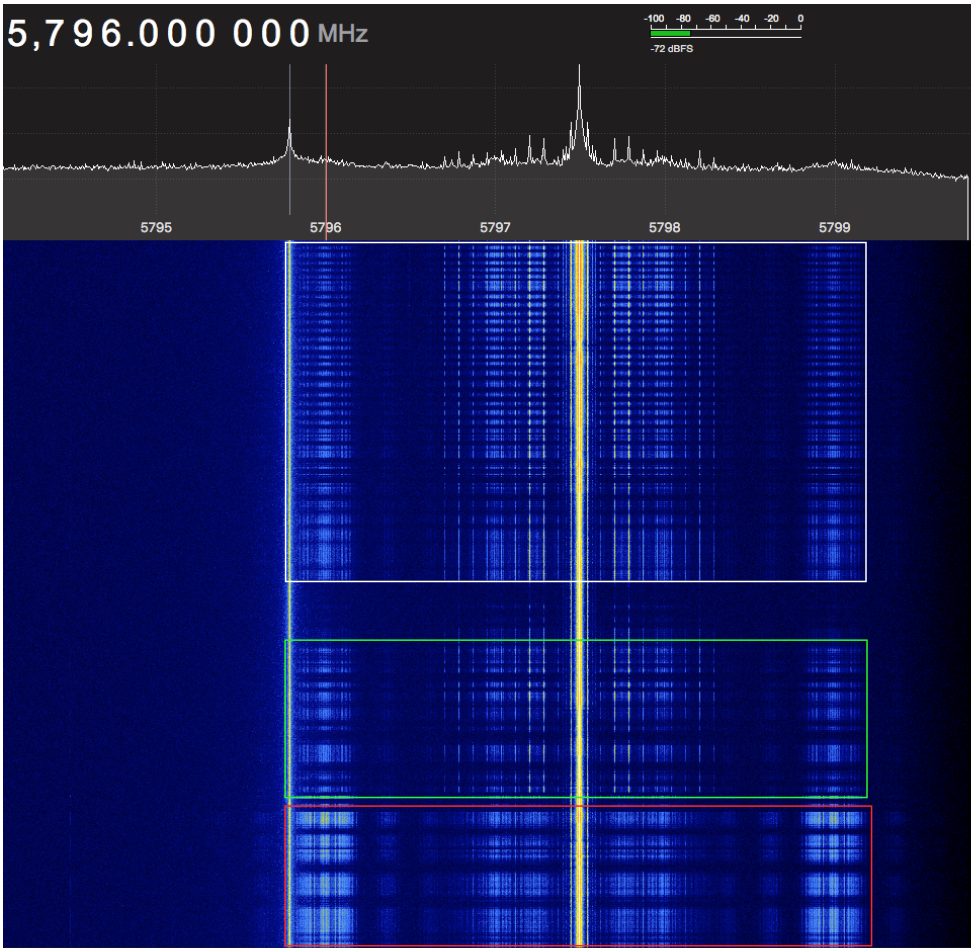

Figure 3.16: Recording of the signal between RSE624 and an OBU. In red, a real transaction, in white a message with many 0's, and in white a message with many 1's.

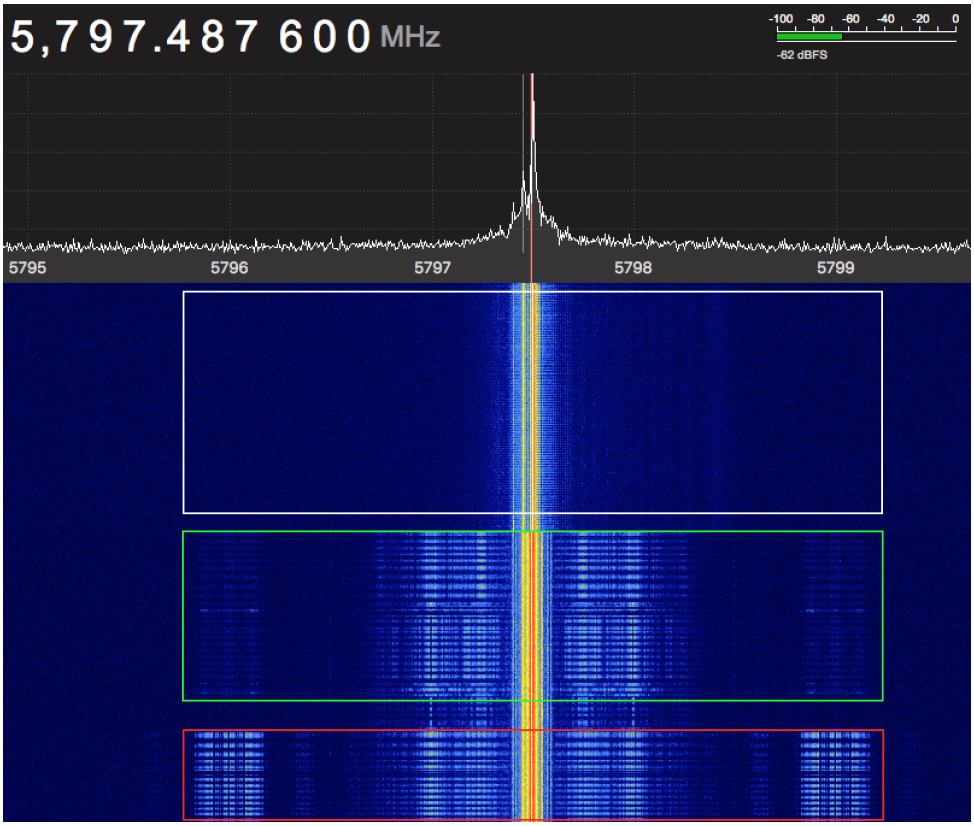

Figure 3.17: In red, recording of a transaction between RSE624 and an OBU. In green, RSE624 sends a message to OBU, and the OBU does not respond. In white, the transmitter build with GNU Radio from Section 3.3.2, is recorded.

3.6 Results and Discussion

This chapter describes how a customizable RSE can be build and the requirements of an RSE. Jonathan Hansen's work on building a customizable RSE from 2016 is studied, and a lot of errors and mistakes are found and corrected. Unfortunately, the DSRC transceiver is still not working correctly, but the program is heading in the right direction. Because of lack of time, a desktop DSRC transceiver, RSE624, was bought from Q-Free. RSE624 is a customizable RSE, with possibilities to construct DSRC messages down to bit level. In the next chapter, this transceiver will be used for communication with an OBU. Some recording of the signals sent between an RSE and an OBU is also captured, and greater understanding of the signal is achieved.

The communication between an RSE and an OBU in DSRC is not encrypted, which may be a security weakness of EFC. With no encryption, an attacker with a

radio receiver can sniff the communication and may read sensitive information sent in the transaction. However, this chapter explains how hard it is to understand the signal and the physical layer in DSRC. No encryption also open possibilities for a replay attack, where real signals are recorded and later replayed. Replay attacks are also a hard task for an attacker because the signal in EFC is very complex.

Chapter 4
Communication between an RSE and an OBU

In the second part of this thesis, the objective is to obtain MACs from an OBU. To be able to do this, many messages have to be constructed correctly based on information received from the OBU. DSRC and EFC are explained in Section 2.2 and Section 2.3. This chapter explains how the messages needed to obtain MACs can be constructed.

4.1 Message Authentication Codes in EFC

With a customizable RSE like RSE624, it is possible to send and receive messages with an OBU. In Norway, security level 1 is used, so to get MACs from an OBU, access credentials is needed. A test-OBU with a known MAcK is used for testing.

MACs are part of the GET_STAMPED.response sent from an OBU to an RSE. As explained in Section 2.6, MACs are generated over AttributeIDList and a random number, RndRSE. Because these values are sent from the RSE, a chosen-plaintext attack is possible. In a chosen-plaintext attack, the attacker can use the OBU as an encryption oracle. In this way, the random number can be changed to many different values, and the OBU responds with the corresponding MAC. Then, many plaintext-ciphertext pairs are obtained.

It is also possible to use an empty AttributeIDList. With an empty AttributeIDList, the MAC is only calculated over 8 bytes, meaning only one DES-encryption is needed. A GET_STAMPED.request can be constructed as shown in Table 4.1. ActionType is 00 (hex), which indicates that it is a GET_STAMPED.request. AttributeIDList is 00 (hex), indicating that no attributes are wanted. Nonce length is 04 (hex), because RndRSE is 4 bytes long, and finally RndRSE contain 4 bytes freely chosen by the attacker.

Table 4.1: Specific parameters set in a GET_STAMPED.request in a chosen-plaintext attack.

Parameter	**Value** (in hex)	**Remark**
ActionType	00	00 specifies a GET_STAMPED.request
AttributeIDList	00	Empty attribute list
Nonce Length	04	Length of RndRSE
RndRSE	XX XX XX XX	4 byte chosen by the attacker

4.2 Frame Contents

To obtain a MAC from an OBU, some messages have to be sent between the RSE and the OBU. A message sequence diagram is shown in Figure 4.1, and a closer description of each message follow below.

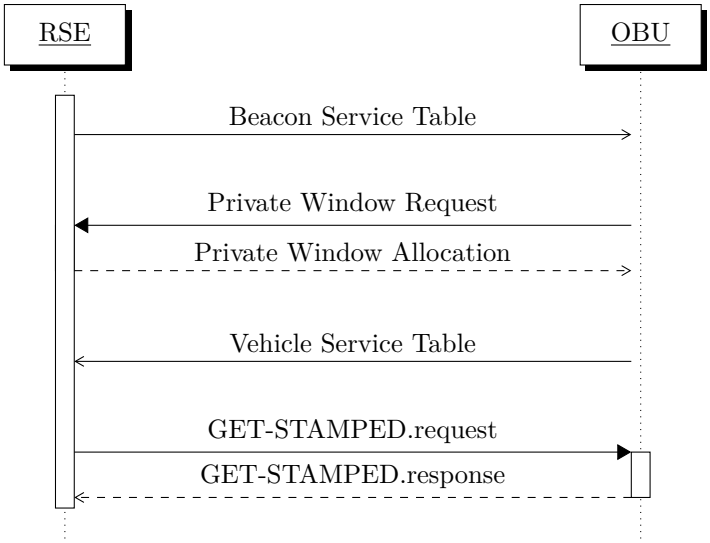

Figure 4.1: Message sequence diagram for obtaining MACs.

4.2.1 Frame Check Sequence

Some trouble were experienced when trying to calculate the correct FCS. Mostly the problem was caused by big/little-endian and converting between these. In the Interface Protocol [Q-F01] to the reader RSE624, an example C-code for FCS calculation was included. Unfortunately, this program did not completely work, but finally, after some time and modifications, a working FCS-program was running.

In Appendix A the C-code used for calculating FCS is included. Please note that the program outputs the FCS in wrong order. ABCD is outputted as CDAB. Using *subprocess* [Fou] in Python, it is possible to run the compiled output file from C. In Figure 4.2, the function "gen_crc" is shown. This function uses subprocess to send data to a.out and receives the FCS back. Then the FCS is reordered to correct order.

```
def gen_crc(data):
  crc3 = subprocess.check_output("./a.out " + data, stderr=
    subprocess.STDOUT, shell=True)
  crchex = hex(int(crc3))
  if len(crchex) == 5:
    # If the CRC is too short
    crchex = crchex[3] + crchex[4] + "0" + crchex[2]
  else:
    crchex = crchex[4] + crchex[5] + crchex[2] + crchex[3]
  return crchex
```

Figure 4.2: Python code for calculation of FCS. Subprocess is used to run the code in Appendix A.

4.2.2 Beacon Service Table

In Table 4.2, an example of a BST is shown. The first three octets are RSE624 specific and follow the protocol shown in Table 3.3 and Table 3.4. Octet 3, 4, and 5 are all link layer specific. Octet 4 is Broadcast LID, the first 7 bits is chosen and set to 1111 111, and the last bit is set to 1, which indicates that the LID only is one octet. The MAC Control Field is set to 1010 0000, which indicates that the frame contains an LPDU and that the direction of the transmission is downlink (from RSE to OBU), and that the RSE allocates a uplink window. The LLC Control Field is set to 0000 0011 indicating that it is unacknowledged connectionless, as specified in 8.3.2.1 in EN 12795 [CEN03a].

Octet 7 to 21 are part of the data unit and starts with a fragmentation header. A fragmentation header contains a fragmentation indicator, a PDU number, a fragment counter and finally an extension indicator. The bits are set according to 6.3.3 in EN 12834 [CEN03b] and are not essential for the tasks of this thesis.

The next part of the frame is the BST, which contains BeaconID and a 32 bit UNIX time-stamp. The option indicator indicates that EID is not present and that AID = 1 (electronic fee collection). Finally, the FCS is calculated, and a stop flag is added.

Table 4.2: Frame content of a Beacon Service Table sent from an RSE to an OBU.

Octet	**Attribute/Field**	**Bit**	**Hex**
1	DSRC Transmit		*F
2	Config	0000 0000	00
3	Length	0001 1000	18
4	Broadcast LID	1111 1111	FF
5	MAC control field	1010 0000	A0
6	LLC control field	0000 0011	03
7	Fragmentation header	1001 0001	91
8	BST {	1000	
	Option indicator	0	
		000	80
9	BeaconID.ManufacturerID	0000 0000	00
10		0011 0	
		000	30
11	BeaconID.IndividualID	0000 0110	06
12		0000 0011	03
13		1110 0001	E1
14		0101 1001	59
15	Time	0000 1001	09
16		1001 1011	9B
17		1110 0110	E6
18	Profile	0000 0000	00
19	MandApplications	0000 0001	01
20	Option indicator	0000 0001	01
21	ProfileList }	0000 0000	00
22	Frame check sequence	0111 0011	73
23		0010 0000	20
24	Stop flag	0111 1110	7E

4.2.3 Private Window Request

When the OBU receives a BST, it evaluates the fields BeaconID and Time. If the tests succeeds, a new Private LID is created. Together with a MAC control field, the Private LID is sent from the OBU to the RSE. In Table 4.3, an example of a Private Window Request is shown. Note that the reader (RSE624) automatically removes the start and stop flag when receiving messages.

Table 4.3: Frame content of a Private Window Request sent from an OBU to an RSE.

Octet	Attribute/Field	Bit	Hex
1	DSRC Receive		*I
2	Length	0000 0111	07
3	Private LID	1110 1010	EA
4		1010 0100	A4
5		0110 0100	64
6		1100 0101	C5
7	MAC control field	0110 0000	60
8	Frame check sequence	1101 0000	D0
9		1100 1001	C9

4.2.4 Private Window Allocation

Similarly to a Private Window Request, a Private Window Allocation only consists of the Private LID and a MAC control field. The MAC control field specifies if it is a request or an allocation. In Table 4.4, an example of a Private Window Allocation is shown.

Table 4.4: Frame content of a Private Window Allocation sent from an RSE to an OBU.

Octet	Attribute/Field	Bit	Hex
1	DSRC Transmit		*F
2	Config	0000 0000	00
3	Length	0000 1011	0B
4	Private LID	1110 1010	EA
5		1010 0100	A4
6		0110 0100	64
7		1100 0101	C5
8	MAC control field	0010 1000	28
9	Frame check sequence	1001 1100	9C
10		0000 0111	07
11	Stop flag	0111 1110	7E

4.2.5 Vehicle Service Table

After the OBU has received a BST and a private window is requested and allocated, the OBU sends a VST to the RSE. EFC-ContextMark is one of the most important elements in a VST. It contains information about CountryCode, IssuerIdentifier, TypeOfContract and ContextVersion. These fields are necessary information for the RSE when handling the transaction. Next, in a VST, a reference to the access credentials key is included together with a random number (RndOBE). These fields are used by the RSE when calculating access credentials. An example of a VST is shown in Table 4.5.

Table 4.5: Frame content of a Vehicle Service Table sent from an OBU to an RSE. The fields needed to calculate access credentials are in bold.

Octet	Attribute/Field	Bit	Hex
1	DSRC Receive		*I
2	Length	0011 1010	3A
3	Private LID	1110 1010	EA
4		1010 0100	A4
5		0110 0100	64
6		1100 0101	C5

Continued on next page

Table 4.5 – continued from previous page

Octet	Attribute/Field	Bit	Hex
7	MAC control field	1100 0000	C0
8	LLC control field	0000 0011	03
9	Fragmentation header	1001 0001	91
10	VST {	1001 0000	90
11	Profile	0000 0000	00
12	Applications	0000 0010	02
13	OptionIndicator	1100 0001	C1
14	EID	0000 0001	01
15	Parameter	0000 0010	02
16		0001 0000	10
17	Country code	0011 0000	30
18	IssuerIdentifier	1100 0000	C0
19		0011 0011	33
20	TypeOfContract	0000 0000	00
21		0000 0001	01
22	Context version	0000 0010	02
23	Container	0000 0010	02
24		0000 0010	02
25	**AC_CR-MasterKeyRef**	**0000 0000**	**00**
26	**AC_CR-Diversifier**	**0010 1010**	**2A**
27	Container	0000 0010	02
28		0000 0100	04
29	**RndOBE**	**0100 1101**	**4D**
30		**1000 1011**	**8B**
31		**1101 1001**	**D9**
32		**0001 1001**	**19**
33	Option indicator	1100 0001	C1
34	EID	0000 0010	02
35	Parameter	0000 0010	02
36		0001 0000	10
37	CountryCode	0011 0000	30

Continued on next page

Table 4.5 – continued from previous page

Octet	Attribute/Field	Bit	Hex
38	IssuerIdentifier	1100 0000	C0
39		0011 0011	33
40	TypeOfContract	0000 0000	00
41		0000 0010	02
42	ContextVersion	0000 0010	02
43	Container	0000 0010	02
44		0000 0010	02
45	AC_CR-MasterKeyRef	0000 0000	00
46	AC_CR-Diversifier	0010 1010	2A
47	Container	0000 0010	02
48		0000 0100	04
49	RndOBE	0100 1101	4D
50		1000 1011	8B
51		1101 1001	D9
52		0001 1001	19
53	EquipmentClass	1111 0011	F3
54		0000 0001	01
55	ManufactureID	0000 0000	00
56		0000 0011	03
57	OBU-status	0000 0011	03
58		0000 0000	00
59	Frame check sequence	0101 0011	53
60		1010 1010	AA

4.2.6 GET_STAMPED.request

Because the test-OBU use security level 1, access credentials have to be calculated and have to be included in GET_STAMPED.request to receive more messages from the OBU. As explained previously, the test-OBU has known Master Access Key. Access credentials were calculated following the algorithms explained in Section 2.6. Python functions for how the access key is derived from the master access key, and how access credentials are calculated, are shown in Figure 4.3. The python library

PyDes [Whi] is used for both DES and Triple DES encryption, in CBC mode and 0000 0000 as initialization value.

```
from pyDes import *


def find_ack(mack, keyref):
  k = triple_des(mack, CBC, "\0\0\0\0\0\0\0\0")
  return k.encrypt(keyref*4).encode('hex')

def calc_accr(ack, rndobe):
  k = des(ack, CBC, "\0\0\0\0\0\0\0\0")
  return k.encrypt(rndobe + //
    b"\x00\x00\x00\x00").encode('hex')
```

Figure 4.3: Python functions for deriving Access Credentials Key from Master Key and how to calculate Access Credentials.

As described to begin with in this chapter, we want the MAC to be calculated over an empty attribute list and the random number. In Table 4.6, ActionType and AttributeIDList is 00, and the random number, RndRSE, is set to 00 00 00 00 (hex). RndRSE is freely chosen bytes, and to obtain a second MAC, it can, for example, be set to FF FF 00 00.

Table 4.6: Frame content of a GET_STAMPED.request sent from an RSE to an OBU.

Octet	**Attribute/Field**	**Bit**	**Hex**
1	DSRC Transmit		*F
2	Config	0000 0000	00
3	Length	0000 1011	25
4	Private LID	1110 1010	EA
5		1010 0100	A4
6		0110 0100	64
7		1100 0101	C5
8	MAC control field	1010 1000	A8
9	LLC control field	0111 0111	77
10	Fragmentation header	1111 0001	F1

Continued on next page

Table 4.6 – continued from previous page

Octet	Attribute/Field	Bit	Hex
11	ACTION.request {	0000	
	AccessCredentials	1	
	ActionParameter	1	
	IID	0	
	Mode	1	0D
12	EID	0000 0001	01
13	ActionType	0000 0000	00
14	Access Credentials	0000 0100	04
15	AC_CR	1111 1010	FA
16		0111 1010	7A
17		1010 1011	AB
18		0111 0010	72
19	ActionParameter	0001 0001	11
21	AttributeIDList	0000 0000	00
22	Nonce length	0000 0100	04
23	RndRSE	0000 0000	00
24		0000 0000	00
25		0000 0000	00
26		0000 0000	00
27	KeyRef_OP }	0110 1111	6F
28	Fragmentation header	1111 0001	F1
29	GET.request {	0110	
	AccessCredentials	1	
	ActionParameter	0	
	IID	0	
	Mode	0	68
30	EID	0000 0001	01
31	AccessCredentials	0000 0100	04
32	AC_CR }	1111 1010	FA
33		0111 1010	7A
34		1010 1011	AB

Continued on next page

Table 4.6 – continued from previous page

Octet	Attribute/Field	Bit	Hex
35		0111 0010	72
36	Frame check sequence	0001 1101	1D
37		0001 0011	13
38	Stop flag	0111 1110	7E

4.2.7 GET_STAMPED.response

In a GET_STAMPED.response, the wanted MAC is included as Operator Authenticator. It is a 4-byte value, which means only the first half of the DES output is sent. For the test-OBU, the MAC is 87DD1D11, when the RndRSE is 00000000.

Table 4.7: Frame content of a GET_STAMPED.response sent from an OBU to an RSE

Octet	Attribute/Field	Bit	Hex
1	DSRC Receive		*I
2	Length	0011 1010	17
3	Private LID	1110 1010	EA
4		1010 0100	A4
5		0110 0100	64
6		1100 0101	C5
7	MAC control field	1100 0000	D0
8	LLC control field	0000 0011	F7
9	LLC status field	0000 0000	00
10	Fragmentation header	1111 0001	F1
11	ACTION.response {	0001	
	IID	0	
	ResponseParameter	1	
	ReturnStatus	0	
	Fill	0	14

Continued on next page

Table 4.7 – continued from previous page

Octet	Attribute/Field	Bit	Hex
12	EID	0000 0001	01
13	ResponseParameter	0001 0010	12
14	AttributeList	0000 0000	00
15	Authenticator	0000 0100	04
16	**OperatorAuthenticator }**	**1000 0111**	**87**
17		**1101 1101**	**DD**
18		**0001 1101**	**1D**
19		**0001 0001**	**11**
20	Fragmentation header	1111 0001	F1
21	GET.response {	0111	
	IID	0	
	AttributeList	0	
	ReturnStatus	1	
	Fill	0	72
22	EID	0000 0001	01
23	ReturnStatus }	0000 0010	02
24	Frame check sequence	1111 1111	FF
25		0111 0000	70

4.3 Results and Discussion

This chapter shows that it is possible to send a GET_STAMPED.request to an OBU with an empty AttributeIDList. A direct consequence of this is that the MAC is calculated over a known plaintext, and that OBUs can be used as encryption oracles, producing a lot of plaintext-ciphertext pairs. Attacking MACs can be done with a chosen-plaintext attack, which will be described in Chapter 5.

In the GET_STAMPED.response frame content in Section 4.2.7, the MAC "87 DD 1D 11" is received from the test-OBU. The MAC is only 32 bit, meaning that the second half of the DES-output is not included.

In Appendix B the complete Python code for communicating with an OBU is included. The program ran two times with different RndRSE to obtain two different

MACs. With only half of the output, the total number of possible keys are 2^{24}. If we obtain another MAC, the number of possible keys are reduced down to only one, or at least a few. With a third MAC, this would not be a problem, as the third value can be used to verify that the key found is correct. Changing RndRSE to "FF FF 00 00", the MAC received is "E5 CF FC 13".

The program calculates Access Credentials and receives two MACs from the test-OBU. As earlier explained, OBUs blocks the communications if the BeaconID is the same or the time difference is less than 255 seconds. In the program, BeaconID is changed from the first to the second BST to avoid BST-blocking from the OBU.

4.3.1 Access Credentials

It is worth mentioning that in Norway, security level 1 is used. In security level 1, the OBU asks for access credentials from the RSE. Without correct calculation of this, it is not possible to obtain MACs from OBUs. Chapter 5, explains an attack for finding the key used in the computation of the MAC. With possibilities to conduct such an attack, the only real security mechanism left in EFC is the access control.

Attacking the access control is not as easy as attacking MACs. When calculating access credentials, the OBU chooses a random number which the access credentials shall be calculated over, as explained in Section 2.6.3. The random number, RndOBU, is 32 bit, resulting in at most 2^{32} different access credential values for each key, AcK. The KeyRef used to derive AcK from a master key, is only 16 bit, resulting in only 2^{16} different master keys.

Building a customizable OBU, and let it communicate with a legitimate RSE could be a solution to this problem. By placing it in the communication zone of an RSE, and then iterate through every possible KeyRef, all 65 536 access keys will be obtained. Because of the truncation, where only the leftmost 32 bits is sent, you will need at least 131 072 transactions to be able to obtain every access key. Estimating that it is possible to obtain 10 access keys per second, the total time would be around 3.6 hours. Such an attack is easily detectable for the RSE because it will receive much non-valid transaction in a very short amount of time.

Another possibility is to study how the random number is generated in OBUs. Is it random? Can it generate 2^{32} different numbers, or does it only use a few of them? This would be very interesting to investigate. There are only two values that change between each transaction, the private LID and the random number, RndOBU. My guess is that the random number is depended on a known value and that it should be possible to anticipate the random number, or even better, be able to choose what the random number should be.

Chapter 5
Building a Rainbow Table

This chapter will present some of the most basic functions for creating a rainbow table, and also some suggestions on how to improve the generation of a rainbow table.

5.1 Rainbow Tables

Rainbow tables are precomputed tables used for reversing cryptographic hash functions. This method is invented by Philippe Oechslin and was first published in his paper *"Making a Faster Cryptanalytic Time-Memory Trade-Off"* in 2003 [Oec03]. In 1980, Martin Hellman described a method for reducing the time of cryptanalysis as a cryptanalytic time-memory trade-off by precalculating data. Two years later, in 1982, Ronald Rivest improved Hellman's technique by introducing distinguished points.

Figure 5.1 illustrates Hellman's traditional tables. It consist of several tables, where each table contains a lot of chains. A chain is build by applying a hash function and a reduction function to a start value ($m_{1,0}$) over and over again. A reduction function is not an inverse of the hash function, but it is a function used to map hashes back to plaintexts. It is only the start ($m_{1,0}$) and the end value ($m_{1,t}$) that are stored in the table. The reduction function is equal in each table, but is different for each table.

Oechslin proposed a new way of pre-calculating the data, namely rainbow tables. Rainbow tables has only one table, where each column has it own reduction function. By doing it this way, the probability of merges decrease. For a merge to arise, a collision have to occur at the exact same column. Figure 5.2 illustrates Oechslin's rainbow tables. If a color is assigned to each reduction function, it will form a the colors of a rainbow. Rainbow tables can also be used for attacking encryption functions, like DES, when both the plaintext and the ciphertext is known. In that case, a DES-key is used as start value and a ciphertext as end value.

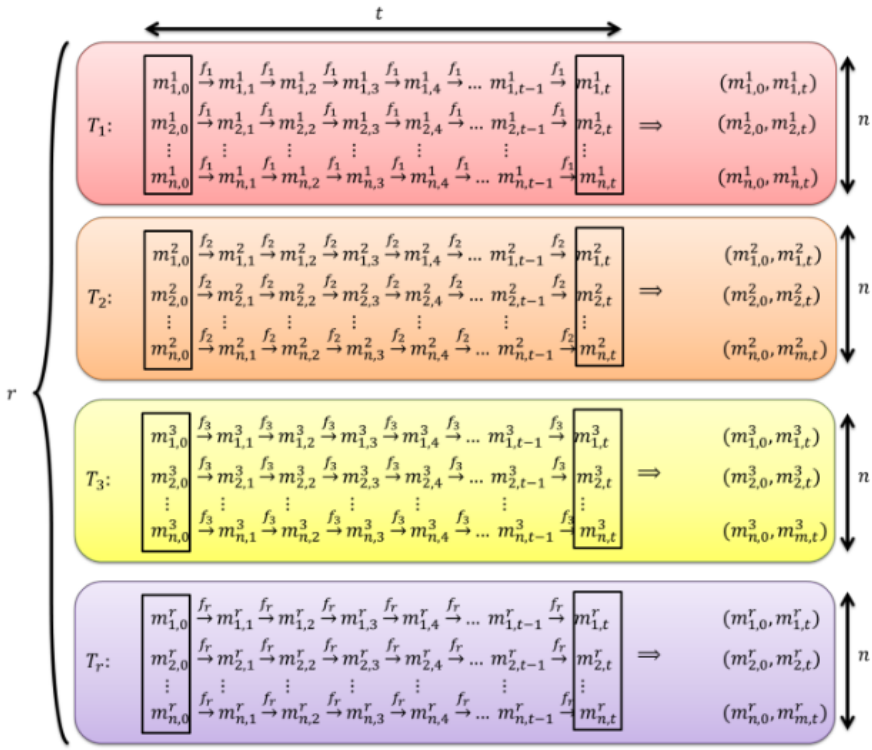

Figure 5.1: Illustration of Hellman's traditional tables. r is the number of tables, t the length of a chain, and n the number of chains in a table. Source: [Mey11].

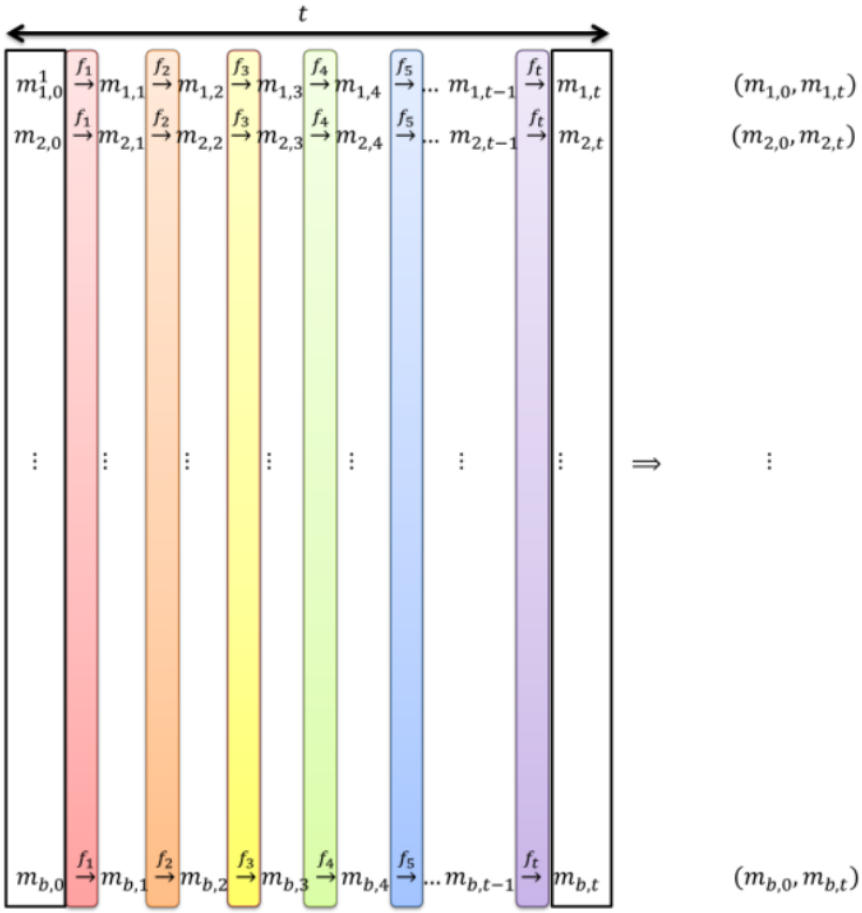

Figure 5.2: Illustration of Oechslin's rainbow tables. t is the length of a chain and b is the number of chains. Source: [Mey11].

After the rainbow table for a specific hash function is generated, the next step will be to search for a hash in the rainbow table. The hash is known, but the corresponding plaintext is not. The algorithm is quite simple and is shown in Figure 5.3. The first part of the algorithm, is to find the chain that has the wanted plaintext and hash. If the hash is not found in any of the end values stored in the table, the reduction function from the last column is applied. Then the new value is hashed, before a new search among the end values is done. If still not found, the two last reduction functions is applied together with the hash function. This is done over and over again until the hash is found. When the hash is found in a chain, that chain is recreated from the start value until the hash and the corresponding plaintext is found.

Rainbow tables are an excellent trade-off between time and memory. It does not store entire rainbow chains, like complete look-up tables, but uses a bit more time for searching and finding the secret plaintext.

1. Look for the hash in the rainbow table.
   a) If found: Go to step 2
   b) If not: Reduce the hash with the N last reduction functions. Go back to step 1. N += 1.
2. Take the plaintext value of that hash and hash it.
   a) If the two hash values match: The plaintext just hashed is the secret plaintext.
   b) If not: Apply the reduction function and go to step 2.

Figure 5.3: Algorithm for finding secret plaintext to a given hash using rainbow tables.

5.2 Generating the Rainbow Table

5.2.1 Overview

Initially, one of the goals for this thesis was to implement a rainbow table on a computer with GPUs. Such implementation would give some numbers of how long time it would take to generate a table, but also to crack the MACs obtained. Because each chain can be created separately, generating a rainbow table is highly parallelizable. It is not an easy task to implement and generate a rainbow table at full size. There are many things to have in mind, and some aspects as memory handling and threading are crucial to getting the wanted speed and output. Because of lack of both time and knowledge of GPU programming, a simple rainbow table for CPUs is created in Python.

Recall that the MAC only is 32-bit and not the entire 64-bit output of a DES encryption. With two plaintext-ciphertext pairs, the possible key space is reduced to only a few. Several possible keys may occur because two keys can produce the very same first halves of a ciphertext. With a third plaintext-ciphertext pair, this should not be a problem, as the third pair can be used for verifying the keys found.

5.2.2 Precalculations

There are many choices to be made before generating a rainbow table. Ideally, we want a perfect rainbow table. A perfect rainbow table is a table, where every end value in the table is unique [Oec03]. With a perfect rainbow table and 2^{56} keys generated, the probability of finding a key is close to 100 %.

As previously explained, rainbow tables are a time-memory trade-off, meaning a decision on how you want to trade time for memory in your rainbow table need to be done. In Table 5.1 an example of some critical parameters to be decided before generating a rainbow table is shown. The total amount of different keys in DES is 2^{56}. Storing one chain in a file takes 35 Bytes, and with a hard drive of 3 TB, there is room for storing up to 9.42 x 10^{10} chains. If you want to generate a table that contains 2^{56} DES-encryptions, each chain has to have a length of 764 587. However, loops, merges and reuse of the same key more than once will occur. A rainbow table with 2^{56} DES-encryptions will not cover the entire key space. Generating a table with 100 % probability of containing the key will be almost infinitely big, but when attacking DES-keys in EFC, this is not a requirement. The goal is not to attack one single OBU, but rather a bunch of them. Then it is not critical if the rainbow table is not able to find the key for one of the OBUs.

Table 5.1: An example of initial parameters to be decided before generating a rainbow table.

Total # of keys	2^{56}
Storage	3 TB
Byte per chain	11 Bytes
Total amount of chains	3 TB / 35 Bytes = 9.42 x 10^{10}
# of keys generated	2^{56} = chains * chainlength
Chainlength	2^{56} / 9.42 x 10^{10} = 764 587

The previous paragraph explains how some rainbow table parameters can be calculated based on available storage and how many keys the table should contain. However, time is also an important factor. The rainbow table can be generated a long time before the actual attack, but with applying the encryption function 2^{56} times, this will take much time.

At NTNU, Department of Information Security and Communication Technology's hacker group, Itemize, has a computer with 4 Nvidia GEFORCE GTX 1080 GPUs installed. To get some numbers of how long it would take to generate a 3 TB big rainbow table on this machine, Hashcat was installed. Hashcat "*is the world's fastest and most advanced password recovery utility, supporting five unique modes of*

attack for over 200 highly-optimized hashing algorithms"[Has]. Using the benchmark function in Hashcat, Itemize' computer can do about 74 000 mega hashes per second (4 x 18500 MH/s). The exact result of the benchmark done in Hashcat is shown in Table 5.2.

Table 5.2: Exact result of benchmarking DES with Hashcat on 4 Nvidia GEFORCE GTX 1080 GPUs.

GPU 1	18560.9 MH/s
GPU 2	18551.4 MH/s
GPU 3	18505.6 MH/s
GPU 4	18422.1 MH/s
Total	**74040.1 MH/s**

Generating a table with the parameters described in Table 5.1, the encryption function will be running 2^{56} times. As will be outlined in the next section, each encryption function contains 2 DES-encryptions, resulting in a total of 2^{57} DES-encryptions.

$$\frac{2^{57}}{74000 \times 10^6 \ H/s} = 1.95 \times 10^6 \ seconds \tag{5.1}$$

$$\frac{1.95 \times 10^6 \ s}{60 \times 60 \times 24} = 22.5 \ days \tag{5.2}$$

By the calculation above, it will take over 22 days only to do the encryption function. On top of that time for the reduction function and chain and key generation have to be added. These are not so time-consuming tasks, but still, have to be taken into account. Also, the start and end points, have to be written to one or several files, and the tables have to be sorted. Without any numbers of how long these tasks will take, it is hard to estimate the total time of generating a rainbow table with the parameters set in Table 5.1. However, because the rainbow table generation can be done before the attack, and the fact that the generation time is in order of days and not months, a rainbow table of 3 TB, may be a reasonable size.

It is also important to know something about how long it would take to search in a rainbow table before generating it. Longer chains lead to longer searching time, but with a computer like the one mention above, this will not be a problem. At most,

the total number of encryptions/reductions are the sum of an arithmetic progression from 1 to 764 587.

$$S_n = \frac{n\ (a_1 + a_n)}{2} \tag{5.3}$$

$$S_n = \frac{764587\ (1 + 764587)}{2} = 2.92 \times 10^{11} \tag{5.4}$$

$$\frac{2 \times 2.92 \times 10^{11}}{74000 \times 10^6\ H/s} = 7.90\ seconds \tag{5.5}$$

By the equations above, we can see that the highest count of total encryptions/reductions is 2.92×10^{11}. The computer mentioned above will use under 8 seconds for only the DES encryptions. The most time-consuming task would be to compare the current value with all the end keys in the rainbow table. Next, one chain has to be reconstructed until the key is found. Without an actual table or implementation, it is hard to estimate the time for this.

5.2.3 Encryption Function

As said, to reduce the number of possible keys to only a few, two MACs are combined in the rainbow table. In Figure 5.4 Python code for encrypting two plaintexts and combine them to only one output is shown. First plaintext1, 00 04 00 00 00 00 00 00 (hex), is encrypted with DES using the current key and in CBC-mode with an initial vector. Then, plaintext2, 00 04 FF FF 00 00 00 00 (hex), is encrypted in the same way as plaintext1. Finally, the last halves of the output are removed, and the output is combined into one output. *PyDES* [Whi] is used for encryption. Table 5.3 gives an example of how two plaintexts are encrypted, truncated and finally merged.

```
from pyDes import *

def crypt (key):
  plaintext1 = "\x00\x04\x00\x00\x00\x00\x00\x00"
  plaintext2 = "\x00\x04\xff\xff\x00\x00\x00\x00"
  k = des(key, CBC, "\0\0\0\0\0\0\0\0")
  o1 = k.encrypt(plaintext1).encode('hex')[:8]
  o2 = k.encrypt(plaintext2).encode('hex')[:8]
  output = (o1+o2).decode('hex')
  return output
```

Figure 5.4: Python code for encrypting, truncating and merging two plaintexts.

Table 5.3: An example of how two plaintexts are encrypted, truncated and finally merged. The DES key used is 00 00 00 00 00 00 00 00 (hex).

	Plaintext1 (in hex)	**Plaintext2** (in hex)
Plaintext	00 04 00 00 00 00 00 00	00 04 FF FF 00 00 00 00
DES encrypted	E6 D5 F8 27 52 AD 63 D1	1E B5 B7 EB 3F 76 9D 86
Truncated	E6 D5 F8 27	1E B5 B7 EB
Merged	E6 D5 F8 27 1E B5 B7 EB	

5.2.4 Reduction Function

A reduction function can be chosen freely by the attacker and has only a few requirements. An essential prerequisite to a reduction function is that it reduce a ciphertext back to a key, in a way it can be reproduced and that it are pretty random [Oec03]. Ideally, it should produce a unique plaintext for each ciphertext, in a way that two different ciphertexts do not produce the same key. Avoiding merges are almost impossible to achieve. With a different reduction function on each column, the possibility for merges between chains is very low, because the same ciphertext has to occur in the very same column for a merge to be created.

One of the great features of DES is that the input, output, and key is all of the same sizes, meaning that the output can almost be used as the next key directly. Figure 5.5 shows how the reduction function for this rainbow table works. It takes a MAC and a column as input. The column is just an integer telling for which column in the rainbow table the reduction function is being applied. Both the MAC and the column is converted to hexadecimal. Then, the last N characters of the MAC is removed, where N equals the length of the column in hex. Finally, the MAC and column are merged to return a new key based on the MAC and the column. Table 5.4 gives an example of how a MAC and a column number together creates a new key.

```
def reduction(mac,column):
  mac = mac.encode('hex')
  macstring = str(mac)
  columnhex = hex(column)
  newkey = macstring[:16-len(columnhex[2:])]+columnhex[2:]
  return newkey[-16:].decode('hex')
```

Figure 5.5: Python code for converting a MAC to new key based on the column in the rainbow table.

Table 5.4: An example of how the reduction function is applied to a MAC, at column number 12. A new key is created based on these two values.

	MAC	**Column**
Input	E6 D5 F8 27 1E B5 B7 EB	12
Input (hex)	E6 D5 F8 27 1E B5 B7 EB	C
Strip key	E6 D5 F8 27 1E B5 B7 E	C
Key = MAC || Column	E6 D5 F8 27 1E B5 B7 EC	

5.2.5 Initial Key Generation

Generating initial keys for each chain is not as easy that it might seem. Without the parity bits in DES-keys, a simple for loop, iterating up from 00 00 00 00 00 00 00 00 (hex), would have done the job just fine. With every 8th bit as a parity bit, a simple for loop would for each key generated 256 (2^8) other keys producing the same ciphertext.

Figure 5.6 shows how the initial keys are generated in this rainbow table. The current key number is taken as input and converted to a 56-bit number. For every 7th bit in the raw key (key without parity bits), a "1" is added as the parity bit. Finally, the key is converted back to hexadecimal and padded with zeros that may disappear at the converting.

```
def key_generation(keynumber):
  x = bin(keynumber)
  rawkey = x[2:].zfill(56)
  tempkey = ""
  for j in range(0,56,7):
    tempkey += rawkey[j:j+7] + "1"
  key = hex(int(tempkey,2))[2:].zfill(16)
  return key.decode('hex')
```

Figure 5.6: Python code for generating initial keys for each chain. Every parity bit is set to 1.

5.2.6 Generating Chains

With a working encryption function, reduction function, and the initial key generation, the next step is to generate the chains. First, a decision of how many chains and how long each chain should have to be made. Figure 5.7 shows a Python function for generating the chains. The chain number and the length of each chain are taken as input, and an initial start key is generated for that chain using the key generation

function described in Section 5.2.5. It is important to store the start key and the current end hash at all times. It is these two values that are stored in the final rainbow table. Then, the current end hash is encrypted and reduced as many times as the chain is long. Finally, the start key and the final end hash are stored in a table. The if/else-sentence is just for ensuring that the final value of the chain is not reduced after it is encrypted.

```
global table
table = []

def chain_maker(chainNumber, chainlength):
  startkey = key_generation(chainNumber)
  endhash = startkey
  for y in range(chainlength):
    if y < chainlength-1:
      endhash = reduction(crypt(endhash),y)
    else:
      endhash = crypt(endhash)
  chain = startkey, endhash
  table.append(chain)
  return None
```

Figure 5.7: Python code for generating the chains in the rainbow table.

5.3 Searching in the Rainbow Table

Rainbow tables are often pre-computed before the actual attack. When the table is generated, and at least two MACs are obtained from the OBU, the next step is searching and finding the key used in that OBU. As explained in Section 5.1, that first, we look at the MAC in the table (end hash), if it is not there the reduction function for the last column and encryption is applied to the MAC. The new value is again checked against the end values in the table, and if no match, the reduction function for the column before that last is applied. Finally, when the modified MAC match an end hash in the table, that chain is recreated from the start key until the original MAC, and the key is found.

In Figure 5.8 and Figure 5.9 two Python functions are included, find_hash and hash_found. Find_hash finds in which chain how holds the MAC, and hash_found recreates a chain and prints out where the key was found and what the key is. After a key is found in the rainbow table, it should be verified with a third plaintext-ciphertext pair.

```
def find_hash(mac, table, chainlength):
  print "\nLooking for key for MAC:"
  print mac.encode("hex")
  omac = mac
  for y in range(chainlength):
    for x in range(len(table)):
      if omac == table[x][1]:
        return hash_found(x, table, chainlength, mac, omac)
    omac = mac
    for i in range(chainlength-2-y,chainlength-1, 1):
      if i < 0:
        break
      omac = reduction(omac,i)
      omac = crypt(omac)
  return "\nHash not in table\n"
```

Figure 5.8: Python code for finding a MAC in a chain.

```
def hash_found(x, table, chainlength, mac, omac):
  startkey = table[x][0]
  x += 1
  fmac = crypt(startkey)
  print('\nEndkey found: %r' %omac.encode("hex"))
  print('Start key: %r' % startkey.encode("hex"))
  print('Chain: %r' %x)
  lastkey = startkey

  for z in range(chainlength):

    if mac == fmac:
      print ('\nKey found: %r' % lastkey.encode("hex"))
      print ('Chain nr %r' %x)
      print ('Place nr %r' %(z+1))
      return ""
    else:
      if z < chainlength-1:
        lastkey = reduction(fmac,z)
        fmac = crypt(reduction(fmac,z))
      else:
        fmac = crypt(fmac)
  return "Something went wrong"
```

Figure 5.9: Python code for reconstructing a chain until the wanted key is found.

5.4 Improving the Rainbow Table

In this section, some suggestions for improving the rainbow table are presented. There is a lot of small improvements that can be done for saving both time and space.

With the current method, both the start value and the end value stored is 64-bit. As earlier explained, 8 of these bits are used for parity check and does not effect the encryption. Removing the parity bits in the start value, which is a key, will save in total 1 Byte per chain. The end value in a chain is not a key but a 64-bit MAC value, so no bits can be stripped. By saving 1 Byte per chain, even more chains can be generated in the same amount of space. However, adding parity bits to the stripped key, will add more time for searching in the table.

Recall that Hellman's originally time-memory trade-off tables consisted of many tables where each table had its reduction function. In rainbow tables, there is one big table where each column has its reduction function. This gives rainbow tables much lower risk of merges than the traditional Hellman's table. In Steven Meyer's Master's thesis from 2011, he stated that creating more rainbow tables with different reduction functions will lower the risk of merges. "*(...) the best performance on a single rainbow table is 86 % due to the quantities of merges. Therefore it becomes necessary to create more tables*" [Mey11]. Meyer attacked MD4 passwords and not DES, so there might be a difference between the probability of merges in MD4 and DES.

Meyer also presents a method to avoid spending time on generating chains that already has merged. By pausing the chain generation, sort the table, remove the duplicates, and then continue to generate the chain, he saved about 3 % of the total generation time.

When searching in a rainbow table, the search will be a lot faster if the table is already sorted by the end values. By doing this, it is also very easy to see and remove chains that end with the same value. Removing merges will save both time and memory.

5.5 Results and Discussion

In this chapter, methods of how to create a rainbow table for attacking MACs obtain from an OBU is presented. Unfortunately, a complete rainbow table was not generated, so no real MACs were attacked. However, implementing a rainbow table on GPUs, require a lot of time and experience in both GPU programming and rainbow table generation. Hopefully, this chapter will be helpful for future work on

this table. A lot of serious calculations and trade-offs have to be made to achieve the wanted result.

The complete Python code is included in Appendix C. It includes all the functions described above. It also sorts the table and writes the table to a CSV-file. Threading is also used to speed up the generation of chains. Finally, it searches for the MAC "87 DD 1D 11 E5 CF FC 13". The MAC is a combination of the MACs received from the test-OBU as explained in Chapter 4.

In Figure 5.10 the output of the Python code in Appendix C is shown. The code produces ten chains with a chain length of 50. First, the generated rainbow table is printed out, sorted by the end value. Then some statistics of the current rainbow table generation is shown, and finally, it searches for the MAC "87 DD 1D 11 E5 CF FC 13."

The program ran on a computer with an Intel Core i7-6700 CPU with 8 3.40 GHz cores and Python 2.7.12 installed.

```
0101010101010101   438cb8f0216f72bb
010101010101010d   495433743c719fff
0101010101010109   4ab7687f32afe254
0101010101010105   67c6aaa2aa628139
010101010101010f   90fc68034b089604
0101010101010103   a1b2c8c5912c3a6f
010101010101010b   a4b91f4f1fe48417
0101010101010107   b751adadcaf8b6a8
0101010101010111   ca249d32d96bad73
0101010101010113   d0609114067e7caf

Generated 10 chains of length 50 in 0.4260389804840088s
Total 6.938893903907228e-15 of all keys
Genarating all would take 1946939.6182172378 years

Looking for key for MAC:
87dd1d11e5cffc13

Hash not in table

Gen time:   0.4260389804840088
Search time:   0.9797370433807373
Total time:   1.4058301448822021
```

Figure 5.10: Output of the Python code in Appendix C.

Chapter 6
Conclusion

This Master's thesis has looked at the possibilities of conducting a chosen-plaintext attack on MACs in EFC. MACs are calculated over data controlled by the RSE, and with a customizable RSE, a chosen-plaintext attack may be possible. Rainbow tables are a perfect time-memory trade-off method for chosen-plaintext attacks like this. Due to the use of single DES, and the fact the key is only 56-bit, a rainbow table can be generated in reasonable time.

The first objective was to *"make a customizable RSE who can communicate with an OBU"*. Chapter 3 describes how a customizable RSE is made by using a USRP and GNU Radio. Jonathan Hansen's program from 2016 is studied, and a lot of his work is verified to be correct. However, some critical errors are found and corrected. The antennas have been replaced with left-hand circular antennas, the gain is increased, the frequency is corrected, and a valid BST is created. The signal generated by the transmitter seems to be correct. Unfortunately, neither the transmitter nor the receiver is working correctly, so the first objective is not achieved. However, the program is heading in the right direction. The most crucial mistake at this moment is the bottleneck caused by the pulse shaper block. A solution may be to rewrite the block from Python to C++, but it is not sure that this will solve the problem.

The second objective was to *"obtain several MACs from an OBU"*. How MACs are obtained is described in Chapter 4. A desktop DSRC transceiver bought from Q-Free is used to communicate with an OBU, and two MACs are obtained. A program for obtaining MACs is written in Python, and this can be almost directly used when the customizable RSE from Chapter 3 is working. It is worth mentioning that the MACs are obtain from a test-OBU. In Norway, security level 1 is used, meaning that access credentials are needed in order to communicate with an OBU. The test-OBU have a known master access key, so access credentials can be correctly calculated for that OBU.

The third and last objective was to *"make an attack on MACs by building a rainbow table"*. Rainbow tables in general and how a rainbow table can be constructed, are described in Chapter 5. With several plaintext-ciphertext pairs obtained and the use of DES with the short 56-bit key, rainbow tables are perfect. Initially, one of the goals of this thesis was to implement a rainbow table on GPUs. Because of lack of both time and experience in GPU programming, this was not done. A simple rainbow table is created in Python, showing the most basic features of a rainbow table. Also, some suggestions on how to save both time and space are presented. Even though a complete rainbow table is not built and MACs are not attacked, this chapter is very valuable for further work on this project.

This thesis shows that an attack on MACs in EFC is feasible. However, it is only possible when security level 0 is used. The only real security mechanism in EFC is the access credentials and not the authentication of OBUs. In Section 4.3.1 a possible attack on access credentials is described, but this attack is easily detectable, and it is not sure if it is possible to conduct.

6.1 Further Work

For further work, it will be natural to continue to work on the customizable RSE. Rewriting the pulse shaper block to C++, may solve the problem, and should be one of the first tasks to be done. By cooperating with companies working with this, like Q-Free or Norbit, building a customizable RSE should be possible. These companies have much experience in this field and may contribute with much valuable guidance.

Implementing a rainbow table in full scale on GPUs would be a fun and interesting task. A lot of calculations and trade-offs have to be done. This thesis gives an explanation of the most basic features of building a rainbow table for attacking EFC, which is valuable for future work on this table.

Finally, it would be interesting to spend even more time studying EFC. Especially, finding a good way to obtain access credentials as mention in Section 4.3.1. This could be either to obtain every access key or be able to anticipate the random number sent from the OBU. Other security weaknesses in EFC may also be found.

References

[ARR] ARRL. Software defined radio. http://www.arrl.org/software-defined-radio. Accessed: 2017-05-18.

[CEN03a] CEN. EN 12795:2003 Road transport and traffic telematics - Dedicated short-range communication (DSRC) - DSRC data link layer: medium access and logical link control. Technical report, European Committee for Standardization, May 2003.

[CEN03b] CEN. EN 12834:2003 Road transport and traffic telematics - Dedicated short-range communication (DSRC) – DSRC application layer. Technical report, European Committee for Standardization, November 2003.

[CEN04a] CEN. EN 12253:2004 Road transport and traffic telematics - Dedicated short-range communication - Physical layer using microwave at 5.8 GHz. Technical report, European Committee for Standardization, April 2004.

[CEN04b] CEN. EN 13372:2004 Road transport and traffic telematics (RTTT) - Dedicated short-range communication - Profiles for RTTT applications. Technical report, European Committee for Standardization, April 2004.

[CEN07] CEN. EN 15509:2007 Road transport and traffic telematics - Electronic fee collection - Interoperability application profile for DSRC. Technical report, European Committee for Standardization, March 2007.

[CEN11] CEN. EN ISO 14906:2011 Electronic fee collection - Application interface definition for dedicated short-range communication. Technical report, European Committee for Standardization, August 2011.

[Etta] Ettus. About Ettus Research. https://www.ettus.com/about. Accessed: 2017-05-18.

[Ettb] Ettus. CBX. https://www.ettus.com/product/details/CBX. Accessed: 2017-05-25.

[Fou] Python Software Foundation. Subprocess management. https://docs.python.org/2/library/subprocess.html#.

[Gqr] Gqrx. Welcome to gqrx. http://gqrx.dk/. Accessed: 2017-05-30.

[Han] Jonathan Hansen. Wireless USRP Test-bed for DSRC Applications - attachment. http://hdl.handle.net/11250/2405115. Accessed: 2017-05-26.

[Han16] Jonathan Hansen. Wireless USRP Test-bed for DSRC Applications. Master's thesis, Norwegian University of Science and Technology, 2016.

[Has] Hashcat. Hashcat. https://github.com/hashcat/hashcat. Accessed: 2017-05-15.

[Kar16] Nikos Kargas. Gen2 UHF RFID Reader. https://github.com/nikosl21/Gen2-UHF-RFID-Reader, 2016.

[Lie] Chris Liechti. pyserial 2.7. https://pypi.python.org/pypi/pyserial/2.7.

[Mey11] Steven Meyer. Breaking 53 bits passwords with Rainbow tables using GPUs. Master's thesis, Ecole polytechnique federale de Lausanne, September 2011.

[NIS] NIST. NIST Withdraws Outdated Data Encryption Standard. https://www.nist.gov/news-events/news/2005/06/nist-withdraws-outdated-data-encryption-standard. Accessed: 2017-06-05.

[NIS80] NIST. FIPS PUB 81 - DES Modes of Operation, December 1980.

[NIS99] NIST. FIPS PUB 46-3 - Data Encryption Standard, October 1999.

[Oec03] Philippe Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2003.

[Q-F01] Q-Free. *QF-RPC Interface Specification*, 201.

[Q-F16] Q-Free. *RSE624 User Manual*, June 2016.

[Rada] GNU Radio. Guided Tutorial GNU Radio in C++. https://wiki.gnuradio.org/index.php/Guided_Tutorial_GNU_Radio_in_C%2B%2B#4.1_C.2B.2B_or_Python.3F. Accessed: 2017-05-29.

[Radb] GNU Radio. InstallingGRFromSource. https://wiki.gnuradio.org/index.php/InstallingGRFromSource#Using_the_build-gnuradio_script. Accessed: 2017-05-25.

[Radc] GNU Radio. UHD Interface. https://gnuradio.org/doc/doxygen/page_uhd.html. Accessed: 2017-05-25.

[Radd] GNU Radio. USRP underflow issue. http://gnuradio.4.n7.nabble.com/USRP-underflow-issue-td54034.html. Accessed: 2017-05-29.

[Rei] Tord Ingolf Reistad. Securing EN 15509 against brute-force attacks. Unpublished.

[Tho09] Einar Thorsrud. Programvaredefinert radio - mulige hyllevareløsninger for DSRC-anvendelser. Master's thesis, Norwegian University of Science and Technology, 2009.

[Whi] Todd Whiteman. pyDes 2.0.1. https://pypi.python.org/pypi/pyDes/.

[Wik] Wikipedia. Block cipher mode of operation. https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation. Accessed: 2017-06-02.

Appendix A

C Code for Frame Check Sequence Calculation

```
#include <stdio.h>
#include <stdint.h>
#include "crc16.h"
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

int hctoi(const char h){
    if(isdigit(h))
        return h - '0';
    else
        return toupper(h) - 'A' + 10;
}

int main(int argc, const char * argv[]) {

    uint16_t val;
  char cdata[200];
  strcpy(cdata, argv[1]);

    unsigned char udata[(sizeof(cdata)-1)/2];
    const char *p;
    unsigned char *up;

    for(p=cdata,up=udata;*p;p+=2,++up){
        *up = hctoi(p[0])*16 + hctoi(p[1]);
    }
    val = crc16_calc(udata,strlen(argv[1])/2);
    printf("%u\n", val);
```

```
    return 0;
}
```

```
#include <stddef.h>

#ifndef crc16_h
#define crc16_h

uint16_t crc16_calc(const uint8_t * data, size_t size);

char printcrc(uint16_t value);


#endif /* crc16_h */
```

```
//#include <inttypes.h>
#include <stddef.h>
#include <stdio.h>
#include <stdint.h>



const uint16_t fcstab[256] =
{
0x0000, 0x1189, 0x2312, 0x329b, 0x4624, 0x57ad, 0x6536, 0
    x74bf,
0x8c48, 0x9dc1, 0xaf5a, 0xbed3, 0xca6c, 0xdbe5, 0xe97e, 0
    xf8f7,
0x1081, 0x0108, 0x3393, 0x221a, 0x56a5, 0x472c, 0x75b7, 0
    x643e,
0x9cc9, 0x8d40, 0xbfdb, 0xae52, 0xdaed, 0xcb64, 0xf9ff, 0
    xe876,
0x2102, 0x308b, 0x0210, 0x1399, 0x6726, 0x76af, 0x4434, 0
    x55bd,
0xad4a, 0xbcc3, 0x8e58, 0x9fd1, 0xeb6e, 0xfae7, 0xc87c, 0
    xd9f5,
0x3183, 0x200a, 0x1291, 0x0318, 0x77a7, 0x662e, 0x54b5, 0
    x453c,
0xbdcb, 0xac42, 0x9ed9, 0x8f50, 0xfbef, 0xea66, 0xd8fd, 0
    xc974,
0x4204, 0x538d, 0x6116, 0x709f, 0x0420, 0x15a9, 0x2732, 0
    x36bb,
```

```
0xce4c, 0xdfc5, 0xed5e, 0xfcd7, 0x8868, 0x99e1, 0xab7a, 0
    xbaf3,
0x5285, 0x430c, 0x7197, 0x601e, 0x14a1, 0x0528, 0x37b3, 0
    x263a,
0xdecd, 0xcf44, 0xfddf, 0xec56, 0x98e9, 0x8960, 0xbbfb, 0
    xaa72,
0x6306, 0x728f, 0x4014, 0x519d, 0x2522, 0x34ab, 0x0630, 0
    x17b9,
0xef4e, 0xfec7, 0xcc5c, 0xddd5, 0xa96a, 0xb8e3, 0x8a78, 0
    x9bf1,
0x7387, 0x620e, 0x5095, 0x411c, 0x35a3, 0x242a, 0x16b1, 0
    x0738,
0xffcf, 0xee46, 0xdcdd, 0xcd54, 0xb9eb, 0xa862, 0x9af9, 0
    x8b70,
0x8408, 0x9581, 0xa71a, 0xb693, 0xc22c, 0xd3a5, 0xe13e, 0
    xf0b7,
0x0840, 0x19c9, 0x2b52, 0x3adb, 0x4e64, 0x5fed, 0x6d76, 0
    x7cff,
0x9489, 0x8500, 0xb79b, 0xa612, 0xd2ad, 0xc324, 0xf1bf, 0
    xe036,
0x18c1, 0x0948, 0x3bd3, 0x2a5a, 0x5ee5, 0x4f6c, 0x7df7, 0
    x6c7e,
0xa50a, 0xb483, 0x8618, 0x9791, 0xe32e, 0xf2a7, 0xc03c, 0
    xd1b5,
0x2942, 0x38cb, 0x0a50, 0x1bd9, 0x6f66, 0x7eef, 0x4c74, 0
    x5dfd,
0xb58b, 0xa402, 0x9699, 0x8710, 0xf3af, 0xe226, 0xd0bd, 0
    xc134,
0x39c3, 0x284a, 0x1ad1, 0x0b58, 0x7fe7, 0x6e6e, 0x5cf5, 0
    x4d7c,
0xc60c, 0xd785, 0xe51e, 0xf497, 0x8028, 0x91a1, 0xa33a, 0
    xb2b3,
0x4a44, 0x5bcd, 0x6956, 0x78df, 0x0c60, 0x1de9, 0x2f72, 0
    x3efb,
0xd68d, 0xc704, 0xf59f, 0xe416, 0x90a9, 0x8120, 0xb3bb, 0
    xa232,
0x5ac5, 0x4b4c, 0x79d7, 0x685e, 0x1ce1, 0x0d68, 0x3ff3, 0
    x2e7a,
0xe70e, 0xf687, 0xc41c, 0xd595, 0xa12a, 0xb0a3, 0x8238, 0
    x93b1,
```

```
0x6b46, 0x7acf, 0x4854, 0x59dd, 0x2d62, 0x3ceb, 0x0e70, 0
    x1ff9,
0xf78f, 0xe606, 0xd49d, 0xc514, 0xb1ab, 0xa022, 0x92b9, 0
    x8330,
0x7bc7, 0x6a4e, 0x58d5, 0x495c, 0x3de3, 0x2c6a, 0x1ef1, 0
    x0f78
};

uint16_t crc16_calc(const uint8_t * data, size_t size) {

    uint16_t fcsval = 0xffff;
    size_t i;
    for (i = 0; i < size; i++) {
        fcsval = (fcsval >> 8) ^ fcstab[(fcsval ^ data[i]) &
    0xff];
    }
    return ~fcsval;
}

unsigned char* printcrc(uint16_t value) {
    char* hexcode[5];
    sprintf(*hexcode,"%0x", value);
    puts(*hexcode);
    return *hexcode;
}
```

Appendix B
Python Code for the DSRC Program

```
import serial, time
from pyDes import *
from binascii import unhexlify
import os
import sys
import subprocess
import datetime
import binascii
import struct
from datetime import datetime


try:
  ser = serial.Serial('/dev/ttyUSB0', baudrate=57600,
    timeout=0.1, bytesize = serial.EIGHTBITS, parity=serial.
    PARITY_EVEN, stopbits=serial.STOPBITS_TWO)
  print "Connected to " + ser.name
except:
  sys.exit("Error connecting device")

time.sleep(0.1)

def gen_bst(beaconID):
  dt = str(datetime.now())
  print dt[0:19]
  time2 = hex(int(time.mktime(time.strptime(dt[0:19], '%Y-%m
    -%d %H:%M:%S'))) - time.timezone)
  data = "FFA00391800030" + "0603E" + str(beaconID) + time2
    [2:] + "00010100"
```

```
  return "*F00" + hex(int(str(len(data)/2+6)))[-2:] + data +
    gen_crc(data) + "7e\r"


def gen_crc(data):
  crc3 = subprocess.check_output("./a.out " + data, stderr=
   subprocess.STDOUT, shell=True)
  crchex = hex(int(crc3))
  # print "CRC: \t" + crchex
  if len(crchex) == 5:
    # print "For kort CRC!!!"
    crchex = crchex[3] + crchex[4] + "0" + crchex[2]
  else:
    crchex = crchex[4] + crchex[5] + crchex[2] + crchex[3]
  return crchex

def gen_private_window_allocation(lid):
  data = lid + "28"
  return "*F000B" + data + gen_crc(data) + "7E\r"


def find_ack(mack, keyref):
  k = triple_des(mack, CBC, "\0\0\0\0\0\0\0\0")
  return k.encrypt(keyref*4).encode('hex')

def calc_accr(ack, rndobe):
  k = des(ack, CBC, "\0\0\0\0\0\0\0\0")
  return k.encrypt(rndobe + b"\x00\x00\x00\x00").encode('hex
   ')

def make_get_stamped(lid, accr, num):
  eid = "01"
  frag = "F1"
  if num == 1:
    rnd = "0000"
  else:
    rnd = "FFFF"
  data = lid + "A877" + frag + "0D" + eid + "0004" + accr +
   "1100" + "04" + rnd + "0000" + "6F" + frag + "68" + eid +
    "04" + accr
```

```
  return "*F00" + hex(int(str(len(data)/2+6)))[-2:] + data +
    gen_crc(data) + "7e\r"



def main():
  bst = gen_bst(1)
  ser.write(bst)
  response =  ser.readline()
  r = response.split("*")
  r2 = r[-1]
  data = r2[3:-5]
  lid = data[0:8]
  pwa = gen_private_window_allocation(lid)
  ser.write(pwa)
  vst =  ser.readline().split('*')[-1]
  keyref = unhexlify(vst[47:51])
  ack = find_ack(mack1, keyref)
  accr = calc_accr(unhexlify(ack), unhexlify(vst[55:63]))
  accr = accr[0:8]
  get_stamped = make_get_stamped(lid, accr, 1)
  ser.write(get_stamped)
  gsa =  ser.readline().split('*')[-1]
  if int(gsa[1:3]) == 17:
    print "Auth 1: \t\t" + gsa[29:37]
  else:
    print "No auth received"

  time.sleep(0.1)

  bst = gen_bst(2)
  ser.write(bst)
  response =  ser.readline()
  r = response.split("*")
  r2 = r[-1]
  data = r2[3:-5]
  lid = data[0:8]
  pwa = gen_private_window_allocation(lid)
  ser.write(pwa)
  vst =  ser.readline().split('*')[-1]
  keyref = unhexlify(vst[47:51])
```

```
  ack = find_ack(mack1, keyref)
  accr = calc_accr(unhexlify(ack), unhexlify(vst[55:63]))
  accr = accr[0:8]
  get_stamped = make_get_stamped(lid, accr, 2)
  ser.write(get_stamped)
  gsa =  ser.readline().split('*')[-1]
  if int(gsa[1:3]) == 17:
    print "Auth 2: \t\t" + gsa[29:37]
  else:
    print "No auth received"

mack1 = b'\x53\x56\x56\x2d\x74\x65\x73\x74\x62\x72\x69\x6b\
    x31\x31\x31\x30'

main()
```

Appendix C

Python Code for the Rainbow Table

```
from pyDes import *
import os
import time
import struct
from random import randint
import csv
import threading



def crypt (key):
  plaintext1 = "\x00\x04\x00\x00\x00\x00\x00\x00"
  plaintext2 = "\x00\x04\xff\xff\x00\x00\x00\x00"
  k = des(key, CBC, "\0\0\0\0\0\0\0\0")
  o1 = k.encrypt(plaintext1).encode('hex')[:8]
  o2 = k.encrypt(plaintext2).encode('hex')[:8]
  output = (o1+o2).decode('hex')
  return output

def reduction(mac,column):
  mac = mac.encode('hex')
  macstring = str(mac)
  columnhex = hex(column)
  output = macstring[:16-len(columnhex[2:])]+columnhex[2:]
  return output[-16:].decode('hex')

def make_chains (chains,chainlength):
  threads = []
  for x in range(chains):
```

```
        t = threading.Thread(target=chain_maker, args=(x,
    chainlength))
        threads.append(t)
        t.start()
        for thread in threads:
          thread.join()
  return

def key_generation(keynumber):
  x = bin(keynumber)
  rawkey = x[2:].zfill(56)
  tempkey = ""
  for j in range(0,56,7):
    tempkey += rawkey[j:j+7] + "1"
  key = hex(int(tempkey,2))[2:].zfill(16)
  return key.decode('hex')

def chain_maker(chains, chainlength):
  startkey = key_generation(chains)
  endhash = startkey
  for y in range(chainlength):
    if y < chainlength-1:
      endhash = reduction(crypt(endhash),y)
    else:
      endhash = crypt(endhash)
  chain = startkey, endhash
  table.append(chain)
  return None

def find_hash(mac, table, chainlength):
  print "\nLooking for key for MAC:"
  print mac.encode("hex")
  omac = mac
  for y in range(chainlength):
    for x in range(len(table)):
      if omac == table[x][1]:
        return hash_found(x, table, chainlength, mac, omac)
    omac = mac
    for i in range(chainlength-2-y,chainlength-1, 1):
      if i < 0:
        break
```

```
        omac = reduction(omac,i)
        omac = crypt(omac)
    return "\nHash not in table\n"

def hash_found(x, table, chainlength, mac, omac):
    startkey = table[x][0]
    x += 1
    fmac = crypt(startkey)
    print('\nEndkey found: %r' %omac.encode("hex"))
    print('Start key: %r' % startkey.encode("hex"))
    print('Chain: %r' %x)
    lastkey = startkey

    for z in range(chainlength):

      if mac == fmac:
        print ('\nKey found: %r' % lastkey.encode("hex"))
        print ('Chain nr %r' %x)
        print ('Place nr %r' %(z+1))
        return ""
      else:
        if z < chainlength-1:
          lastkey = reduction(fmac,z)
          fmac = crypt(reduction(fmac,z))
        else:
          fmac = crypt(fmac)
    return "Something went wrong"


def main(chains, chainlength, mac):
    start = time.time()
    make_chains(chains, chainlength)
    table1 = table
    table1 = sorted(table1, key=lambda pair: pair[1])
    for i in table1:
      print i[0].encode("hex") + "\t" + i[1].encode("hex")
    end1 = (time.time() - start)
    total = (chains*chainlength)/2.0**56
    alle = ((end1*(2.0**56)) / (chains*chainlength))
     /(60*60*24*365)
```

```
    print ("\nGenerated %r chains of length %r in %rs" %(
      chains, chainlength,end1))
    print ("Total %r of all keys" %total)
    print ("Genarating all would take %r years" %alle)
    start2 = time.time()
    print find_hash(mac, table, chainlength)
    print "Gen time:\t %r" %(end1)
    print "Search time:\t %r" %(time.time()-start2)
    print "\nTotal time:\t %r" %(time.time()-start)
    with open('test_file.csv', 'w') as csvfile:
      writer = csv.writer(csvfile)
      [writer.writerow(r) for r in table]

global table
table = []

mac = '87DD1D11E5CFFC13'.decode('hex')

main(10,50,mac)
```