NTNU

Norwegian University of
Science and Technology

Wireless USRP Test-bed for DSRC Applications

Jonathan Hansen

Master of Science in Communication Technology
Submission date: June 2016
Supervisor: Stig Frode Mjølsnes, ITEM
Co-supervisor: Tord I. Reistad, Statens Vegvesen

Norwegian University of Science and Technology
Department of Telematics

Title: Wireless USRP Test-bed for DSRC Applications

Student: Jonathan Hansen

Problem description:

Dedicated Short-Range Communications (DSRC) is a wireless communications capability used in Intelligent Transport Systems (ITS). It can be used for vehicle-to-vehicle and vehicle-to-infrastructure communications, and is the standard for electronic fee collection in many toll-road systems. The authentication protocol used in this standard is based on a challenge/response type, and the centralized key distribution scheme is by tamper-resistant on-board devices [1].

It is important to understand the threats and secure communications adequately, because of the increasing deployment of such vehicular wireless communications systems. This project aims to use Universal Software Radio Peripheral (USRP) devices to set up a test-bed for DSRC applications, and then to analyze the security of the current protocols and threat models. The USRP devices will be used to simulate a Road Side Unit (RSU) and an On Board Unit (OBU) from a toll-road system. The goal of the project is to explore possible attacks on the cryptographic protocol by using the wireless test-bed.

[1] CEN/TC 278 Intelligent Transport Systems. Electronic Fee Collection - Assessment of security measures for applications using dedicated short-range communication. 2015-11. 42 pages.

Responsible professor: Stig Frode Mjølsnes, ITEM

Supervisor: Tord I. Reistad, Statens Vegvesen

Abstract

Dedicated Short-Range Communications (DSRC) is a wireless communication technology used in Intelligent Transport Systems for road vehicles. Applications utilizing DSRC are becoming more and more widespread, and include both vehicle-to-vehicle and vehicle-to-infrastructure systems. As this technology expands, it is important to keep security in mind. New technologies can open up new opportunities for exploitation not previously thought of. Several threats against the DSRC system for Electronic Fee Collection (EFC) have recently been presented [CEN15].

In this thesis, the possibilities for communicating with DSRC applications using a Universal Software Radio Peripheral (USRP) are explored. The open software toolkit GNU Radio is used to implement a DSRC transmitter and receiver, aiming to imitate the communication between a roadside unit (RSU) and an on-board unit (OBU). The functionality of these programs were not completely verified, but the work gives a good indication that DSRC communication should be possible with a USRP.

Additionally, this thesis studies the security of the EFC system. Calculations of access credentials and authentication values are described, and weaknesses identified. The use of the Data Encryption Standard (DES) for computing these values is especially in focus. Attacks against the DES keys in EFC are presented, where customized DSRC devices are used for obtaining the values needed. To crack the encryption and retrieve the access credential and authentication keys, a modified brute-force attack is necessary. This attack is explained in detail, and an example implementation is provided. Time-memory trade-off algorithms to speed up the brute-forcing are also presented and compared.

Sammendrag

Dedikert kortholdslink (DSRC) er en type trådløs kommunikasjonsteknologi brukt i intelligente transportsystemer (ITS) for landkjøretøy. Applikasjoner som anvender DSRC blir stadig mer utbredt, og inkluderer både kjøretøy-til-kjøretøy- og kjøretøy-til-infrastruktur-systemer. Etter hvert som denne teknologien utvikler seg og tas mer i bruk, er det viktig å tenke på sikkerhet. Nye teknologier kan åpne for nye typer misbruk, som tidligere ikke har vært nødvendig å ta stilling til. I en nylig trusselanalyse ble en mengde svakheter ved DSRC-systemet for elektronisk bompengeinnkreving (EFC) påpekt [CEN15].

Denne masteroppgaven undersøker mulighetene for kommunikasjon med DSRC-applikasjoner ved hjelp av den programvaredefinerte radioen Universal Software Radio Peripheral (USRP). Det frie rammeverket GNU Radio blir brukt til å implementere en DSRC-sender og -mottaker, i et forsøk på å imitere kommunikasjonen mellom en veikantenhet og en bilbrikke. Funksjonaliteten til disse programmene ble ikke fullstendig verifisert, men arbeidet gir en god indikasjon på at DSRC-kommunikasjon burde være mulig gjennom en USRP.

I tillegg blir sikkerheten i EFC-systemet studert i denne oppgaven. Utregninger av verdier for tilgangskontroll og autentisering beskrives, og svakheter blir identifisert. Bruken av Data Encryption Standard (DES) for beregning av disse verdiene er spesielt i fokus. Angrep mot DES-nøkler i EFC blir presentert, hvor tilpassede DSRC-innretninger brukes for å få tak i de nødvendige verdiene. For å knekke krypteringen og hente ut nøklene, er det nødvendig med et modifisert brute-force-angrep. Dette angrepet er forklart i detalj, og en eksempelimplementasjon er gitt. Såkalte «time-memory trade-off»-algoritmer for å effektivisere brute-force-søket blir også presentert og sammenlignet.

Preface

This Master's thesis is carried out at the Norwegian University of Science and Technology in Trondheim, in the 10th semester of my Master of Science degree in Communication Technology. The thesis is written under the supervision of Professor Stig Frode Mjølsnes from the Department of Telematics (ITEM) and Tord Ingolf Reistad from the Norwegian Road Administration (Statens Vegvesen).

I would like to thank Professor Stig Frode Mjølsnes and Tord I. Reistad for their valuable guidance and feedback during the work with this thesis.

Trondheim, June 2016

Jonathan Hansen

Contents

List of Figures

List of Tables

List of Acronyms

ADC analog-to-digital converter.

AES Advanced Encryption Standard.

BST Beacon Service Table.

CBC Cipher Block Chaining.

CEN European Committee for Standardization.

CFB Cipher Feedback.

DAC digital-to-analog converter.

DES Data Encryption Standard.

DSRC Dedicated Short-Range Communications.

ECB Electronic Codebook.

EFC Electronic Fee Collection.

EPC Electronic Product Code.

FIPS Federal Information Processing Standard.

FPGA field-programmable gate array.

GPS Global Positioning System.

GPSDO GPS Disciplined Oscillator.

GPU graphics processing unit.

GRC GNU Radio Companion.

GUI Graphical User Interface.

IBM International Business Machines Corporation.

ISO International Organization for Standardization.

ITS Intelligent Transport Systems.

MAC Message Authentication Code.

NBS National Bureau of Standards.

NI National Instruments.

NIST National Institute of Standards and Technology.

NRZ non-return-to-zero.

NRZI non-return-to-zero-inverted.

NTNU Norwegian University of Science and Technology.

OBU on-board unit.

OFB Output Feedback.

OSI Open Systems Interconnection.

QA quality assurance.

RF Radio Frequency.

RFID Radio-frequency identification.

RSU roadside unit.

SDR Software Defined Radio.

UHD USRP Hardware Driver.

USRP Universal Software Radio Peripheral.

V2I vehicle-to-infrastructure.

V2V vehicle-to-vehicle.

VST Vehicle Service Table.

XML Extensible Markup Language.

Chapter 1
Introduction

The use of Intelligent Transport Systems (ITS) is increasing in line with the technological advances in our society. Especially within vehicular transport this is noticeable, where new services emerge constantly. Dedicated Short-Range Communications (DSRC) is a wireless communication technology used in such systems. This type of technology is expected to only keep developing in the years to come. Electronic Fee Collection (EFC) is one of the oldest and most widespread types of ITS using DSRC. In systems like this, which has been in use for several years, it is especially important to keep the security in mind. New technologies can emerge and create threats not previously thought of, and increasing computational power can make previously unfeasible attacks more feasible.

Software Defined Radio (SDR) is one of the technologies that has emerged and created countless new possibilities within the radio communication field. It makes it possible to perform complex real-time signal processing on a general processing platform, creating a flexible radio aiming to replace any kind of other radio. The cost of such radios is decreasing, as the technology keeps evolving. The Universal Software Radio Peripheral (USRP) family by Ettus is an example of SDR products available at a reasonable price today.

1.1 Objectives

In this thesis, I will explore the possibilities of communicating with DSRC applications using a USRP. The EFC system is especially in focus, and I aim to create software programs able to imitate communication between a roadside unit (RSU) and an on-board unit (OBU). Showing that direct communication from an SDR to actual EFC entities is possible, would illustrate one of the security concerns with today's EFC standards.

This would also make other threats to the EFC system more evident. The use of the Data Encryption Standard (DES) is an apparent weakness, which also will be

discussed in this thesis. Possible brute-force attacks against DES keys are explored, with a special focus on time-memory trade-off algorithms. I will present concrete examples of how such attacks against EFC can be carried out, with the help of customized DSRC devices like the ones attempted to create in this thesis.

During the work with the thesis, I found part one to be more time consuming than anticipated, because of problems with the software implementation. This created a shortage of time for the rest of the work, which resulted in part two becoming more theoretical than originally planned.

1.2 Structure of the Report

This thesis is divided into 6 chapters, including this introduction chapter:

Chapter 2 provides background information about the topics discussed in the thesis. This includes ITS, DSRC, EFC, USRP, GNU Radio, and DES.

Chapter 3 presents related work to this thesis. Other projects using USRP with DSRC are introduced, before different time-memory trade-off algorithms used for brute-force attacks on DES are explained.

Chapter 4 describes the approach for creating a wireless USRP test-bed for DSRC applications using GNU Radio, and presents and discusses the results.

Chapter 5 looks at the security in current EFC standards and presents possible attacks on these. A brute-force attack against the DES encryption using trade-off algorithms is described.

Chapter 6 includes the conclusion of the work done in this thesis and proposals for possible further work.

Chapter 2
Background

2.1 Intelligent Transport Systems

The term Intelligent Transport Systems (ITS) is used to describe information and communication systems in road, rail, water and air transport. These systems are intended to increase efficiency and safety, and include e.g. navigation systems, emergency warning systems, traffic signal control systems, collision avoidance and electronic fee collection.

2.2 Dedicated Short-Range Communications

DSRC is a wireless communications capability used in ITS. The technology is used both for vehicle-to-vehicle (V2V) communication and vehicle-to-infrastructure (V2I) communication. The DSRC systems used around the world vary a little in e.g. frequency spectrum and protocols used, but in general the communication channels lie around 5.8-5.9 GHz. In this thesis I will work with the European standards developed by the European Committee for Standardization (CEN) and the International Organization for Standardization (ISO):

- EN 12253:2004 Road transport and traffic telematics - Dedicated short-range communication – Physical layer using microwave at 5.8 GHz [CEN04a]
- EN 12795:2002 Road transport and traffic telematics - Dedicated short-range communication (DSRC) – DSRC data link layer: medium access and logical link control [CEN03a]
- EN 12834:2002 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 DSRC system is divided into layers, corresponding to some of, but not all, the layers in the Open Systems Interconnection (OSI) model [ISO94]. Traditionally, the model consists of seven layers: Physical, Data link, Network, Transport, Session, Presentation, and Application. DSRC uses a simplified version of this model, including only the two bottom layers and the top layer: Physical, Data link, and Application, described in the first three standards listed above.

2.3 Electronic Fee Collection

One of the most common types of ITS using DSRC is Electronic Fee Collection (EFC). How these systems should work in Europe is described in the following standards:

- 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]

The standards describe EFC services and functions, on top of the underlying DSRC standards. EFC functions and the related DSRC Layer 7 services from EN 12834 are listed in Table 2.1:

DSRC-L7 service	**EFC function**
INITIALISATION	N/A
ACTION	GET_STAMPED
GET	N/A
SET	N/A
ACTION	SET_MMI
ACTION	ECHO
EVENT-REPORT	RELEASE

Table 2.1: DSRC Layer 7 services and EFC functions

INITIALISATION is used to establish communication between an RSU and OBU, and select application and contract.

ACTION - GET_STAMPED is used to retrieve data from the OBU to the RSU, with an authenticator. Can be used both with and without access credentials.

GET is used to retrieve data from the OBU to the RSU, without an authenticator. Can be used both with and without access credentials.

SET is used to write data to the OBU, with or without access credentials.

ACTION - SET_MMI is used to invoke an MMI function.

ACTION - ECHO makes the OBU echo received data.

EVENT-REPORT - RELEASE is used to terminate the communication.

2.3.1 Security in EFC

Two different levels of security are described in the standards, 0 and 1, where level 0 is mandatory and level 1 is optional. Level 0 includes message authentication, while level 1 adds access control on top. Security level 0 requires the following data elements in the OBU, listed in Table 2.2:

Name	Length in octets
AuthenticationKey1	8
AuthenticationKey2	8
AuthenticationKey3	8
AuthenticationKey4	8
AuthenticationKey5	8
AuthenticationKey6	8
AuthenticationKey7	8
AuthenticationKey8	8
KeyRef	1
RndRSE	5 (1+4)

Table 2.2: OBU data elements for security level 0

KeyRef is a reference to the authentication key used for computation of the Authenticators. The OBU must be able to calculate an Authenticator in order to validate data integrity and origin of the application data. RndRSE is a random number containing SessionTime, also used in the Authenticator computation. For level 1, some more security data elements are required for the OBU, listed in Table 2.3:

Name	Length in octets
AccessKey	8
AC_CR	5 (1+4)
AC_CR-KeyReference	2
RndOBU	5 (1+4)

Table 2.3: OBU data elements for security level 1

At security level 1, the OBU must be able to calculate the Access Credentials AC_CR using the random number RndOBU and the AccessKey. AC_CR-KeyReference is a reference to the key generation and the diversifier for the key computation. The RSU must also be able to calculate Authenticators at level 0 and Access Credentials at level 1.

Authenticator Calculation

The procedure for calculating an Authenticator is explained in annex B.2 in [CEN07] and presented in [Rei]. This algorithm includes the use of DES encryption in Cipher Block Chaining (CBC) mode. The complete calculation is described below, copied almost directly from the standard, and illustrated in Figure 2.1. Authenticators like this are computed when a GET_STAMPED.request is received.

a) Let AuK be the OBU's Authentication Key of a given generation k, referenced by the KeyRef in the GET_STAMPED.request.

b) Let M be the Attributelist in the GET_STAMPED.response concatenated by the octet string containing the RndRSE sent in the GET_STAMPED.request. The RndRSE shall contain the Session Time.

c) Split M into 8-octet blocks D1 (octets 1 to 8), D2 (octets 9 to 16), ... , D_{n-1} (octets 8(n-1) to 8n).

d) According to ISO/IEC 9797-1:1999 MAC Algorithm 1, the remaining bits shall be left justified. To the right of these shall be appended zero value bits, so that a final 8-octet block results D_n.

e) First Step: the first block $I_1 = D_1$ shall be encrypted with AuK:
$O_1 = \text{e[AuK]}(I_1)$

f) The output O_1 shall be XORed with D_2 and this result shall be the input I_2 of the next step:
$I_2 = [O_1]$ XOR $[D_2]$

g) Second Step: I_2 shall be shall be encrypted with AuK:
$O_2 = \text{e[AuK]}(I_2)$

h) The output O_2 shall be XORed with D_3 and this result shall be the input I_3 of the next step:
$I_3 = [O_2]$ XOR $[D_3]$

i) This process shall continue with further 8-octet blocks D_x until the ultimate step D_n.

j) Finally the input I_n shall be encrypted with e[AuK].
O_n = e[AuK](I_n)

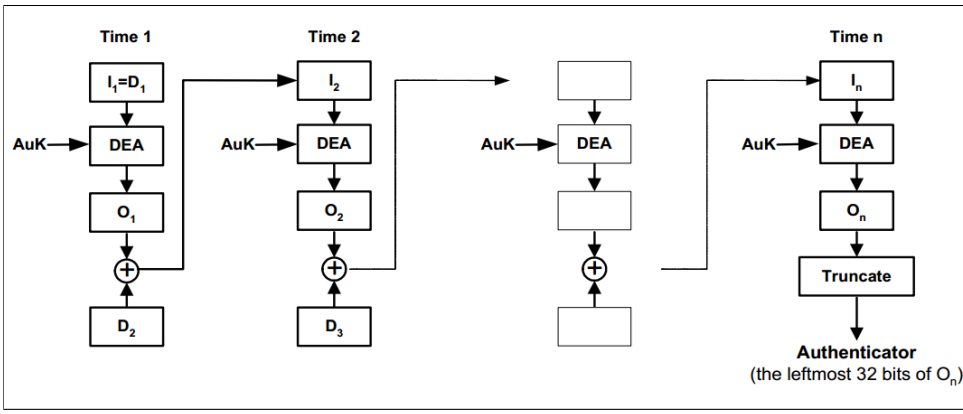


Figure 2.1: Illustration of the authenticator calculation.

Access Credential Calculation

Access Credentials are used in EFC transactions to keep sensitive user data protected, and make sure no non-authorized operators are able to use the OBU. When communication is initialized, the OBU responds to a Beacon Service Table (BST) from the RSU with a Vehicle Service Table (VST), containing an access credential key reference and a random number. The access credentials AC_CR are then calculated at both sides. Next, the RSU sends a request for reading data from the OBU, with the AC_CR included. If the access credentials match, the RSU is accepted and allowed to read data. This procedure is illustrated in Figure 2.2. The calculation of

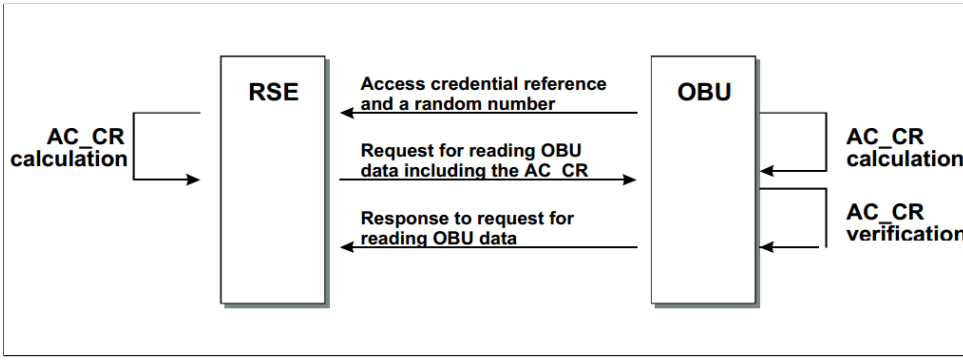


Figure 2.2: Illustration of the access credentials transactions.

access credentials is described in annex B.3 in EN 15509 [CEN07], and replicated below. Figure 2.3 depicts the computation.

a) The 4-octet RndOBU is sent from the OBU to the RSU in the VST. These 4-octet RndOBU shall be left aligned.

b) Let AcK be the derived Access Key.

c) To the right 4 zero octets shall be appended. The result shall be an 8-octet string:
I = 'RndOBU' || '00 00 00 00'Hex.

d) The resulting 8-octet string I shall be encrypted using as follows:
O = e[AcK](RndOBU || '00 00 00 00'Hex).

e) The access credentials AC_CR shall be obtained by truncating the 8-octet string output O and keeping the four left-most octets.

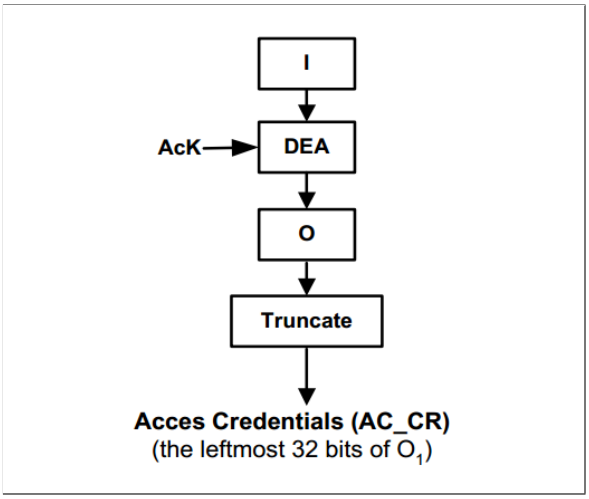


Figure 2.3: Illustration of the calculation of access credentials. The input `I` is a concatenation of the 4-octet RndOBU and 4 zero octets.

2.4 Universal Software Radio Peripheral

Universal Software Radio Peripheral (USRP) is a family of Software Defined Radio (SDR) products created by Ettus Research, which is owned by National Instruments (NI). The products are designed for use with Radio Frequency (RF) applications from 0 to 6 GHz and support both single channel and multiple antenna systems [Ett].

Each USRP contains a motherboard providing a field-programmable gate array (FPGA), digital-to-analog converters (DACs), analog-to-digital converters (ADCs), host processor interface, clock generation and synchronization, and power regulation. Additionally, a daughterboard needs to be installed on the motherboard, in order to provide the RF front end. Different daughterboards and antennas exist for different frequency bands. A more detailed description of the USRP device used for this thesis is given in 4.1.2.

2.4.1 USRP Hardware Driver

To connect the USRP to computer software, Ettus provides the USRP Hardware Driver (UHD). A manual for this driver can be found online at [UHD]. The manual includes information about installation, setting up the USRP and different ways to use it.

2.5 GNU Radio

GNU Radio is a free open-source software development kit for Software Defined Radios. The project was started by Eric Blossom, and the first package was published in 2001. In 2004, he published [Blo04], describing the GNU Radio software and its applications. Information about new releases, documentation, tutorials and the GNU Radio community can be found at the project's wiki page [GRw]. The toolkit provides a variety of signal processing blocks for software radios, as well as the possibility to create your own. These blocks are for the most part created in C++, but can also be written in Python. GNU Radio programs are built as flow charts, which can either be coded directly in Python or designed in a graphical interface called GNU Radio Companion (GRC). When using GRC, the program compiles the Python code automatically.

2.6 Data Encryption Standard

The Data Encryption Standard (DES) is a symmetric-key encryption algorithm, originally developed by the International Business Machines Corporation (IBM) in the 1970s. It was created after the National Bureau of Standards (NBS) published requests for a standard encryption algorithm in 1973 and 1974. DES was approved as a federal standard in 1976, and published as a Federal Information Processing Standard (FIPS) in 1977. It has later been reaffirmed four times, latest in 1999 [NIS99], where the preferred use of Triple DES was specified. DES had by then been proved insecure, and was no longer permitted in new systems. Some years later the Advanced Encryption Standard (AES) was published, and in 2005 the National Institute of Standards and Technology (NIST) (former NBS) withdrew their DES publication.

The DES algorithm design is described in [NIS99]. It is a typical block cipher, which takes 64 plaintext bits and returns a 64-bit ciphertext. A DES key is also 64 bits, but eight of these are used for parity checks, so the actual key length used in the algorithm is 56 bits. Every eighth bit is a parity bit, making sure that each byte is of odd parity. The complete encryption computation is shown in Figure 2.4. The 64-bit input block is first subjected to an initial permutation, before it is split into two 32-bit blocks. Then these blocks go through 16 iterations of a cipher function, where 16 different keys K_1 to K_{16} are used. These keys consist of 48 bits derived from the original 56-bit key through a key schedule. Finally, the output of the 16 iterations are subjected to a new permutation, which is the inverse of the initial permutation. A detailed description of these permutations, as well as the cipher function and the key schedule, can be found in [NIS99].

This block cipher has to be used in a mode of operation, which is an algorithm describing how to apply the block operation repeatedly to ensure confidentiality and authenticity for a complete message consisting of several blocks. DES modes of operation are described in FIPS publication 81 [NIS80], and the modes included are Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB).

2.6.1 Security Issues

Already around the time of publication, the DES algorithm received criticism regarding its security. The small key size of 56 bits was the main issue, making the key vulnerable to brute-force attacks. Whitfield Diffie and Martin Hellman tried to convince the NBS that the security of the standard was too weak, with their paper "*Exhaustive Cryptanalysis of the NBS Data Encryption Standard*" in 1977 [DH77]. In this paper they proposed a DES-cracking machine with an estimated cost of 20 million USD that could find a DES key within a day. Several such brute-force machines were proposed later, and in 1997 the first one was implemented, by the DESCHALL Project [DES]. Currently, the machine holding the DES brute-force record is the COPACOBANA RIVYERA developed by SciEngines [RIV].

Besides brute-force, there also exist some other, faster attacks on DES. One of these is differential cryptanalysis, described in [BS93] by Biham and Shamir. Linear cryptanalysis [Mat93] and Improved Davies' attack [BB97] are two others. However, these attacks require a large amount of known plaintexts, $2^{49}, 2^{43}$ and 2^{50}, respectively. Therefore, brute-force attacks are a lot more effective in practice. The brute-force attacks can be made even more efficient, by doing precomputations. This is called a time-memory trade-off, and will be described more in detail under 3.2 in Related Work.

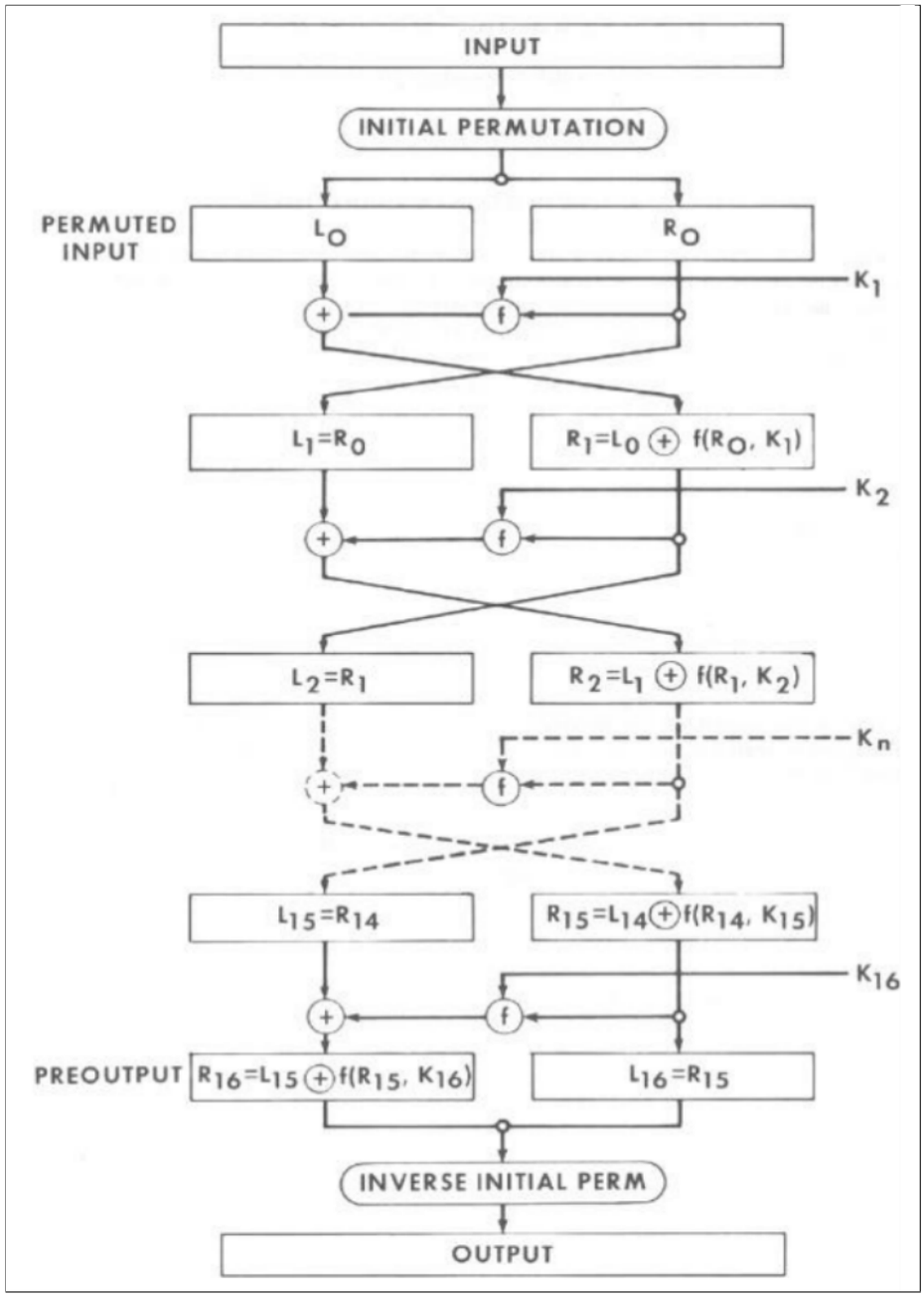


Figure 2.4: Illustration of the DES block cipher computation, taken from [NIS99].

Chapter 3
Related Work

3.1 USRP and DSRC

Even though programmable radios like the USRP is a relatively new technology, there exist a variety of projects within this field. However, very few of these look at the area of DSRC. One that does so is a master's thesis from Norwegian University of Science and Technology (NTNU) in 2009, by electronics student Einar Thorsrud [Tho09]. The title of his thesis translated from Norwegian is "*Software Defined Radio - Possible off-the-shelf solutions for DSRC applications*." In this project Thorsrud looks at different open-source frameworks for SDRs, and discusses how these can be used in relation to DSRC and electronic fee collection. He attempts to implement parts of the physical layer for an EFC RSU using GNU Radio and a USRP. Testing with an OBU test chip shows that this implementation is only partially successful. The OBU wakes up, indicating it receives a modulated carrier wave, but it does not recognize the signals and give any response.

Thorsrud created two GNU Radio flow charts for his thesis, called `dsrc_rsu_transmitter.py` and `dsrc_rsu_receiver.py`. In these programs he used some self-made signal processing blocks, in addition to blocks from the standard GNU Radio library. The blocks he created are collected in a module called `dsrc`.

This thesis has been helpful for my own thesis, as it has given me a starting point for my work with GNU Radio. The signal processing blocks developed by Thorsrud are not compatible with newer versions of GNU Radio and USRP, but it is possible to attempt to recreate similar blocks. The hardware is also different from mine, as he used a USRP 1 with USB connection, a different daughterboard and a specially designed radio attachment instead of a usual antenna.

Another project working with USRP and DSRC is [KN13] from 2013, by A. Kumar and S. Noghanian at the University of North Dakota. Their paper describes how they used a USRP N210 to create a cost efficient test-bed for ITS. Specifically,

they test the received signal power with different antenna positions for a V2V system. The USRP is used in combination with NI LabVIEW to transmit a sine wave at 5.9 GHz through a highly directional antenna. This means that, unlike this thesis, they did not have to consider any encoding or modulation of the signal.

3.2 Breaking DES With Trade-Off Algorithms

To break the DES encryption used in EFC, I will look at time-memory trade-off algorithms to make the brute-force attacks more efficient. This section will present previous work with such algorithms, and describe the different variants of the method.

3.2.1 Hellman's Original Method

As described in 2.6.1, DES has been the target of cryptanalysis and brute-force attacks since it was published. In 1980, Martin Hellman introduced the time-memory trade-off to cryptanalysis [Hel80], making these attacks a lot more efficient. Hellman uses DES as a specific example in his paper, and presents the following statements in the introduction chapter:

> "Breaking the 56-bit Data Encryption Standard (DES) with this method is less complex than doing an exhaustive search on a 38-bit key system. (...) the cost per solution of breaking the DES drops from approximately $5000 for exhaustive search to approximately $10 using the time-memory trade-off."

Hellman's method works by doing precomputations, to trade search time with memory. In these precomputations he creates tables containing ciphertexts (hashes) and their corresponding keys. This is done by encrypting the same plaintext over and over again with different keys. However, the tables are not built as usual look-up tables where all the computed values are stored. He generates hash chains, and stores only the starting point and end point of each chain. To produce these chains, something called a reduction function is used. A reduction function is a function that takes a hash output and reduces this to a usable key. In the case of DES, a 64-bit hash is reduced to a 56-bit key. The function can be very simple, e.g. just dropping the last 8 bits of the hash. Each chain begins with a starting point randomly chosen from the key space, which is used as key in the first encryption. The output of this encryption is then reduced to a new key, which is used in the next encryption, and so on. This procedure is illustrated in Figure 3.1. Finally, after t rounds of encryption and reduction, the end point is stored alongside the starting point. A complete table will then contain m such entries, representing m hash chains, and the table is sorted on the end points. The encryption and reduction steps can be looked at as one

step, denoted as a function f. Figure 3.2 illustrates the table generation using this denotation.

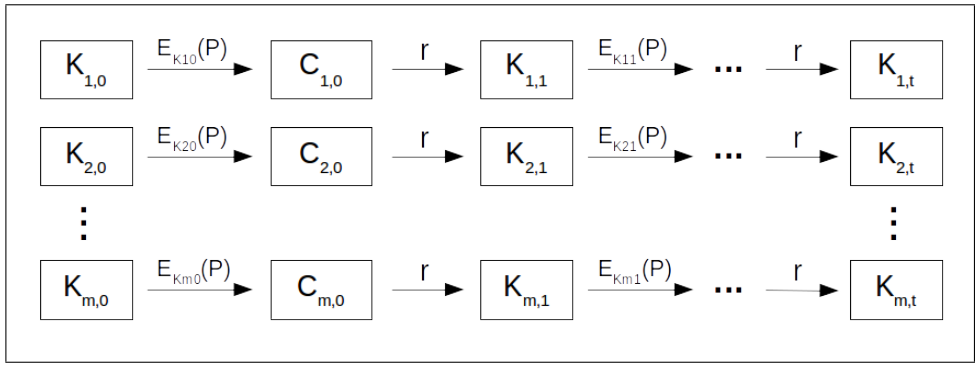


Figure 3.1: Illustration of the hash chain generation, where $K_{i,j}$ is key number j in chain number i, $E(P)$ is the encryption of plaintext P, C is the ciphertext, and r is the reduction function.

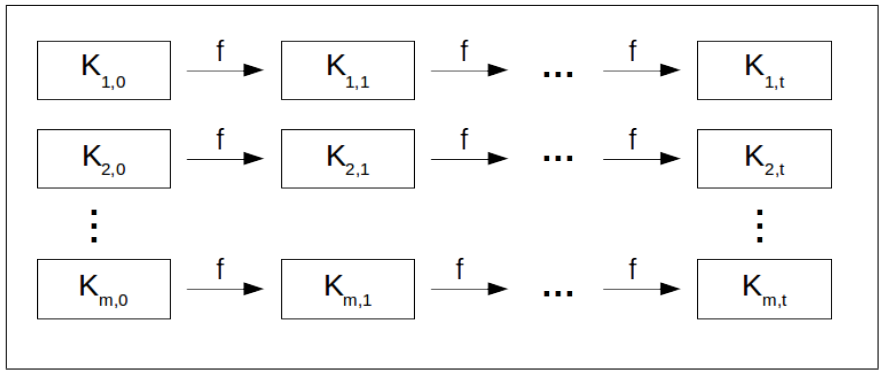


Figure 3.2: Illustration of the hash chain generation, where $K_{i,j}$ is key number j in chain number i, and f is the combined function of encryption and reduction.

When the table is generated, it can be used to find a key with a chosen plaintext attack. We assume the attacker has obtained a ciphertext C corresponding to the plaintext P used to create the hash chains. The first step is then to apply the reduction function to C, and check if the output Y_1 matches any end points in the table. If it does, the key is either the second to last point in that chain, $K_{i,t-1}$, or it is a false alarm. To find $K_{i,t-1}$, you start at the corresponding starting point $K_{i,0}$ and apply f t-1 times. A false alarm can occur if the end point has more than one inverse image. Therefore, a check needs to be performed to make sure the correct key is found. This could for instance be to see if the key deciphers C correctly into P.

If Y_1 does not match any of the end points, f is applied to get Y_2. Then the same check is done for Y_2; first see if it matches an end point, continue to Y_3 if not. Suppose that Y_3 matches an end point, then the key would be $K_{i,t-3}$ in that chain. This check is done all the way to Y_t, and if no matching end points are found by then, the precomputed table does not contain the key. The values t and m can be chosen in different ways, depending on time and memory preferences. In [Hel80], Hellman uses $t = 10^6$ and $m = 10^5$ for the DES-breaking machine he proposes.

3.2.2 Rivest's Distinguished Points

Several improvements to Hellman's time-memory trade-off has been proposed since the method was published. In [Den82], a suggestion from Rivest about distinguished points was mentioned. He had observed a way to reduce the search time, by forcing the end points to satisfy some easily tested property. Typical properties could be that the n first bits are all 0's or all 1's. This means that the chains would no longer have a fixed length t, instead the iterations would simply run until a distinguished point is reached. The search time is then drastically reduced, because it is no longer necessary to do look-ups for points that are not distinguished. To avoid too long chains, the property chosen should be expected to hold after t encipherments on average. Then the expected number of entries in a precomputed table with m chains would still be mt.

In addition to the reduced search time, using distinguished points also has some other advantages. The method makes it possible to detect loops in the hash chains, if no distinguished point is found after a improbable number of iterations. Then the chain can be suspected to contain a loop and discarded. Next, it is also easy to detect merging chains. This is because two merging chains will always reach the same end point, the first distinguished point after the merge. The merging chain can then also be discarded, which means we end up with a table without both loops and merges.

3.2.3 Rainbow Tables

In 2003, Philippe Oechslin proposed a new way to do the precalculations, in order to reduce the number of calculations needed [Oec03]. His method does not use distinguished points, but uses a new type of chains that can handle collisions without merging. This is done by using t different reduction functions in each chain, one for each step. Then a collision will have to occur at exactly the same point in the chain for a merge to happen. If this does happen, the chains will get the same end point, making it easy to discard the merge. So just like with distinguished points, it is easy to generate merge free tables. Furthermore, these tables are also free of chain loops. Since each reduction function is only used once per chain, loops will never occur.

This is clearly an advantage over the other tables, where loops have to be detected and removed. The fact that the chains have a fixed length is also an advantage, as Oechslin shows in [Oec03] that this reduces the number of false alarms. He called this new type of chains "rainbow chains", and a table with these chains is called a "rainbow table."

Chapter 4
Wireless USRP Test-bed

The first part of the project consisted of implementing a USRP test-bed for DSRC applications, specifically EFC. I started by making myself familiar with the system that was going to be simulated, all the way from the physical DSRC layer to the application specific rules for EFC. Next, I focused at the hardware and software tools available, and looked for similar projects that could be helpful. Finally, I implemented the software and tested with the USRP.

4.1 Methodology

4.1.1 Literature Review

To begin the literature review for the first part of my thesis, I started by looking at the DSRC standards described in 2.2 and 2.3. This was arguably the most important literature for my thesis, as it described the complete system I aimed to implement.

Subsequently, I studied the master's thesis by Einar Thorsrud from 2009 [Tho09], introduced in 3.1. When going through his thesis, it was important to pick out which parts that were useful for my own thesis. Since he was an electronics student, his text contains a lot of theory about the signal processing related to the USRP, which is not the main focus of my thesis. Thorsrud's descriptions of the USRP and GNU Radio were not very relevant for my work either, because of the big difference between his versions in 2009 and my versions in 2016. The most interesting part for me was where he described the GNU Radio software implementation, as this was something I intended to do as well.

Other papers about USRP and DSRC were also studied, including [KN13], but the main focus were given to those mentioned above.

4.1.2 Hardware and Software

After the literature review, the next step was to get familiar with the available hardware and software tools.

USRP

The USRP device used in this project is the N200, which is part of the USRP Networked Series. It comes with a Gigabit Ethernet interface, allowing streaming up to 50 MS/s. Additionally, the motherboard contains a Xilinx Spartan 3A-DSP 1800 FPGA, a 100 MS/s dual ADC, and a 400 MS/s dual DAC. A complete overview of the N200 architecture is shown in Figure 4.1, and more details can be found at the product home page [N20].

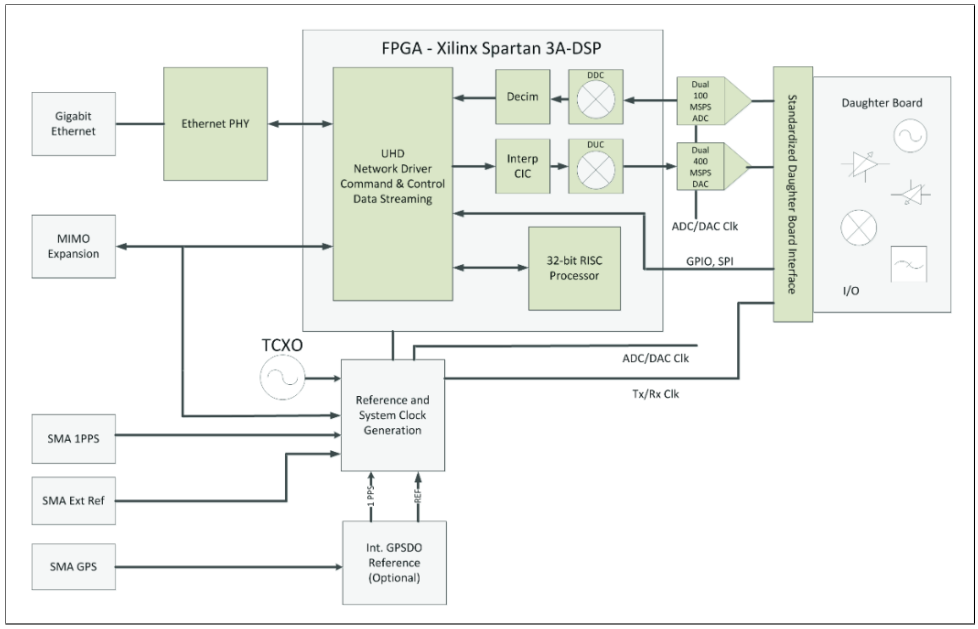


Figure 4.1: Complete overview of the USRP N200 architecture. Image taken from the product datasheet found at [N20].

To provide the RF front end, a CBX daughterboard was attached to the motherboard. This daughterboard was chosen because it covers a frequency band from 1.2 GHz to 6 GHz, meaning it supports DSRC communication at 5.8 GHz. The USRP also needs antennas working at this frequency, so two VERT2450 antennas were used. These antennas cover 2.4 to 2.5 GHz and 4.9 to 5.9 GHz. Furthermore, a GPS Disciplined Oscillator (GPSDO) kit was installed to the motherboard and connected to a Global Positioning System (GPS) antenna. The GPS clock is used to provide increased accuracy for the internal USRP clock. In figure 4.2, you can see a

picture of the N200 used in this thesis, with the VERT2450 antennas and the GPS antenna attached.

Figure 4.2: The N200 with GPS, VERT2450 antennas, Ethernet cable and power cable attached.

Computer

Together with the USRP an HP Compaq 8100 Elite CMT PC was used, with 64-bit Linux Ubuntu 14.04 installed. The processor was an Intel Core i7-860 at 2.8 Ghz with 4 cores and 8 threads, and the installed memory was 8 GB of RAM.

Software Frameworks

The considered software frameworks for this project were GNU Radio, introduced in 2.5, and NI LabVIEW Communications for USRP [Lab]. LabVIEW is a system design suite developed by National Instruments, initially released in 1986. It provides a graphical programming interface similar to GRC and many libraries with functions for e.g. signal processing, data acquisition, mathematics and analysis.

Before deciding whether to use GNU Radio or LabVIEW, I did some research for available open projects that could help me with my DSRC program. The first observation worth mentioning is that there seems to be very few USRP projects focusing on DSRC. Those mentioned in 3.1 were the only ones that looked really relevant. I decided to use GNU Radio, because of the help I could get from Thorsrud's thesis [Tho09], and the fact that there seems to be a lot more support for GNU Radio on the web in general. A likely reason for this is the fact that GNU Radio is free, open-source and supports multiple platforms, while LabVIEW requires a licence and the current version of the System Design Suite is only supported on Windows 7.

Installing UHD and GNU Radio

The different ways to install UHD are described in the online manual from Ettus [UHD]. For most Linux distributions UHD comes as a part of the package management, and for Debian and Ubuntu it can be installed with the command

```
$ sudo apt-get install libuhd-dev libuhd003 uhd-host
```

GNU Radio can also be installed in several different ways, as explained in the installation guide on the wiki page [GRw]. The guide recommends installation via standard repositories, simply by executing

```
$ sudo apt-get install gnuradio
```

Since it seemed like the easiest way, I installed both UHD and GNU Radio with these commands. However, when trying to run GNU Radio Companion after the installation, I encountered some problems. The program would not start, and an error message about a "segmentation fault" was shown. When running with the "verbose" option, the output showed that the fault happened while trying to import UHD packages. Some research told me that several others also had encountered this problem[1], but no solutions were given. Deleting the UHD packages causing the fault made it possible to run GRC, but then the UHD modules would not be accessible inside the program.

[1] One example at http://gnuradio.org/redmine/issues/796

The only working solution to this problem seemed to be to reinstall the programs. After uninstalling GNU Radio and UHD with the package manager, I first tried building and installing the software from source instead. Unfortunately, the same error kept occurring, and sometimes even new errors showed up when trying to build. The problem appeared to be with compatibility between the GNU Radio and UHD versions. Finally, I found a tip about using a script created to install both UHD and GNU Radio, called `build-gnuradio`[2]. For this to work, it was important that all other builds and installations of the programs were removed. Therefore, I decided to reinstall a clean version of Ubuntu before running the script. This ultimately solved the problem, and I was able to move on to developing in GNU Radio.

4.1.3 RFID Reader

As a start for my DSRC implementation, I was recommended to look at Radio-frequency identification (RFID) programs, because of the similarities between the technologies. Since RFID is a more common technology than DSRC, it would also be easier to find existing implementations. One such implementation that matched well with both my hardware and software, was a "*Gen2 UHF RFID Reader*" by Nikos Kargas [Kar15]. This reader was created late in 2015, and used a USRP N200 and GNU Radio v3.7.4. After following the instructions for installation and configuration, I was able to run the program successfully. At this point I had no RFID tags to test with, but the program ran as it was supposed to and worked with a test file. Since RFID uses a lower frequency than DSRC, 910 MHz in this case, the reader was tested with different USRP attachments than those described in 4.1.2. A SBX daughterboard and VERT900 antennas were used, before the CBX daughterboard and VERT2450 antennas were installed later for the DSRC program.

Kargas' RFID reader was created with GNU Radio, but he had not used the graphical interface GRC. To get familiar with GRC and hopefully create something that could help me with the DSRC program, I wanted to make a graphical representation of the RFID reader. Since a textual version of the flow chart already existed, this should be a pretty straightforward task.

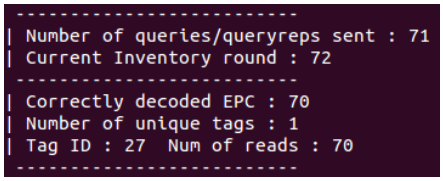
I first started by making the blocks created by Kargas usable in GRC. Each block is graphically represented through an Extensible Markup Language (XML) file, which describes the name, key, parameters, input and output for the block. These files are automatically created when you make a new GNU Radio block, so all I had to do was to fill in the correct information. A guide for this and all other information about creating your own blocks is available under the GNU Radio wiki [GRw]. An example of how these XML files look is shown below, illustrated with `rfid_gate.xml`:

[2] http://www.sbrac.org/files/build-gnuradio

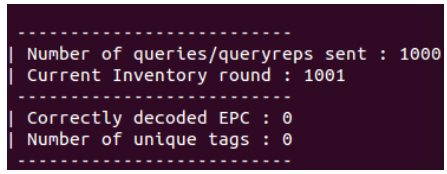
```
<?xml version="1.0"?>
<block>
  <name>gate</name>
  <key>rfid_gate</key>
  <category>rfid</category>
  <import>import rfid</import>
  <make>rfid.gate()</make>

  <param>
    <name>sample_rate</name>
    <key>sample_rate</key>
    <type>int</type>
  </param>

  <sink>
    <name>in</name>
    <type>complex</type>
  </sink>

  <source>
    <name>out</name>
    <type>complex</type>
  </source>
</block>
```

When the XML files were edited, I ran "make install" from the project's build directory, and the blocks became available in GRC. I could then add all the blocks used in Kargas' reader.py file, and recreate the flow chart graphically. After including variables and connecting the blocks, I was ready to compile a new python file and run it just like reader.py. Thanks to GS1 Norway[3], I was able to get hold of some RFID tags to test the reader. However, the program was not able to communicate with any of the tags. Neither Kargas' original program or my GRC program could get any answer when trying to read the tags. Since everything seemed to be working as it should, and the reader was able to read offline test files correctly, it was difficult to find the reason for this. Figure 4.3 shows the output from the RFID reader program run both with the test file and with the USRP. As seen in (a), the reader was able to decode 70 Electronic Product Codes (EPCs) correctly offline, but when testing with the USRP in (b), no tags were found. After trying to get the RFID reader to work with the tags for a while without success, I decided to move on to the DSRC module.

[3] http://www.gs1.no/



(a) Output from the RFID reader run offline with test file



(b) Output from the RFID reader run with the USRP

Figure 4.3: Screenshots of the output from the RFID reader

4.1.4 Creating the DSRC Program

For the DSRC receiver and transmitter, I chose to try to recreate Thorsrud's GNU Radio programs from [Tho09]. His DSRC module and flow charts were written for GNU Radio version 3.2, which is not compatible with version 3.7, so I could not use any of it directly. Nevertheless, it would still be a great help to look at the source code for his blocks, and customize this to my hardware when creating my own blocks. Especially for the signal processing this was very helpful, as it would require a lot of extra work for me to obtain all the necessary knowledge in that field. Thorsrud had not used the graphical interface of GRC to create his flow charts either, which made it a little more challenging for me to create these. All the necessary information about blocks, parameters, connections and variables needed to be extracted from his Python files `dsrc_rsu_transmitter.py` and `dsrc_rsu_receiver.py`.

GNU Radio Module and Blocks

The GNU Radio wiki [GRw] offers some great tutorials for creating your own out-of-tree modules. Out-of-tree modules means modules that are not in the original GNU Radio source tree, such as Thorsrud's module and the module I was going to implement. To begin, I followed the guide and used gr_modtool to create a new module called `dsrc_mod` with the command

```
$ gr_modtool newmod dsrc_mod
```

Next step was to create the blocks. GNU Radio blocks can be written in either C++ or Python. However, writing signal processing blocks in Python comes with a performance penalty, according to the wiki. Therefore I decided to try to use C++, even though this was a new programming language to me. This turned out to be more trouble than expected, as I ran into problems already with the first block. Debugging errors for a new program in a new language took a lot of time and effort, so to avoid wasting unnecessary time I decided to write the blocks in Python after all. Then, if I got the program working with Python blocks, more time could be spent

improving the performance with C++ later if necessary. For this thesis, getting the program to work was the most important priority.

To create a new block, the gr_modtool command "add" was used. I began with the NRZI-to-NRZ block, and created this with the command

```
$ gr_modtool add -t sync -l python
```

The parameter "-t" decided the block type synchronized, and "-l" set the language to Python. I then chose the name, identifier, arguments and whether to add quality assurance (QA) code:

```
GNU Radio module name identified: dsrc_mod
Language: Python
Enter name of block/code (without module name prefix):
nrzi_to_nrz_bb
Block/code identifier: nrzi_to_nrz_bb
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n] y
Adding file 'Python/nrzi_to_nrz_bb.py'...
Adding file 'Python/qa_nrzi_to_nrz_bb.py'...
Editing Python/CMakeLists.txt...
Adding file 'grc/dsrc_mod_nrzi_to_nrz_bb.xml'...
Editing grc/CMakeLists.txt...
```

The block was called `nrzi_to_nrz_bb` because the input and output types are both byte, and as you can see from the empty argument list, it took no arguments. Python QA code is a very smart and effective way to test new blocks, so I chose to add this. For this block, I simply used the same tests in the QA file as Thorsrud. The task performed by this block was to change from non-return-to-zero-inverted (NRZI) to non-return-to-zero (NRZ) encoding, and the C++ code from Thorsrud's block was translated to Python without any big difficulties. The complete code for `nrzi_to_nrz_bb.py` is attached in A.1. To make the block available in GRC, the associated XML file was edited as explained for the RFID blocks above. In exactly the same way, Thorsrud's `nrz_to_nrzi_bb` block was recreated in Python for my `dsrc_mod` module. This code can also be found in the Appendices, under A.2.

The next block was a more advanced block, called `pulse_shaper_bs`. This is a combined block that does FM0 encoding, modulation and pulse shaping. It takes four parameters: `v_min, v_max, phase`, and `pulse_shaper_interpolation`.

- `v_min` and `v_max` represents the minimum and maximum signal values, U_{min} and U_{max}.
- `phase` defines the last signal value given before the input data, which is needed for the FM0 encoding.
- `pulse_shaper_interpolation` decides the samples-per-symbol rate.

Additionally, `pulse_shaper_bs` takes input of the type byte and gives output of the type short, as indicated by the two last letters in the name. Inside this block I have copied two tables with signal sample values used in Thorsrud's block, `half_sin` and `half_sin2`. These tables contain 256 samples each and are used to create two different complete signal periods of 512 samples. The source code for my Python version of `pulse_shaper_bs` is attached in A.3.

Transmitter Flow Chart

To begin the implementation of the transmitter flow chart, I first needed to find all the GNU Radio blocks used in Thorsrud's `dsrc_rsu_transmitter.py` and find the equivalent blocks in my version of GNU Radio. Going through his code, the following blocks were found:

- `dsrc.pulse_shaper_bs`: This out-of-tree block I had already created my own version of, which could be found in the `dsrc_mod` module.
- `gr.vector_source_b`: This block is the same as `blocks.vector_source_b`.
- `gr.file_source`: This block is equivalent to `blocks.file_source`.
- `grc_usrp.simple_sink_s`: The corresponding block to this had to be `uhd.usrp_sink`. However, the old block and the new USRP sink were not identical, so some adjustments had to be made. The most obvious one was that the old sink was able to take input of the type short, while the new could only take the types "Complex float32," "Complex int16," or "VITA word32." I therefore had to add another block, to convert from short to complex. This could be done by a block called `blocks.interleaved_short_to_complex`.
- `gr.file_sink`: This is equivalent to `blocks.file_sink`.

In addition to these blocks, I added a Graphical User Interface (GUI) Time Sink, to visualize the transmitted signal. The next step was to look at the connections in the flow chart. This was pretty straightforward, as there were only two connections needed: the source to the pulse shaper and the pulse shaper to the sink. In Thorsrud's `dsrc_rsu_transmitter.py`, the source and sink were decided through conditional

statements reading textual input. Since I used GRC, I could simply disable and enable the different sources and sinks in the flow chart depending on which one I wanted to use. Therefore, connections were created from both the vector source and the file source to the pulse shaper, and from the pulse shaper to both the USRP sink and the file sink. The connected signal processing blocks are shown in Figure 4.4. In this screenshot the file source and file sink are disabled, as the vector source and USRP sink are used.

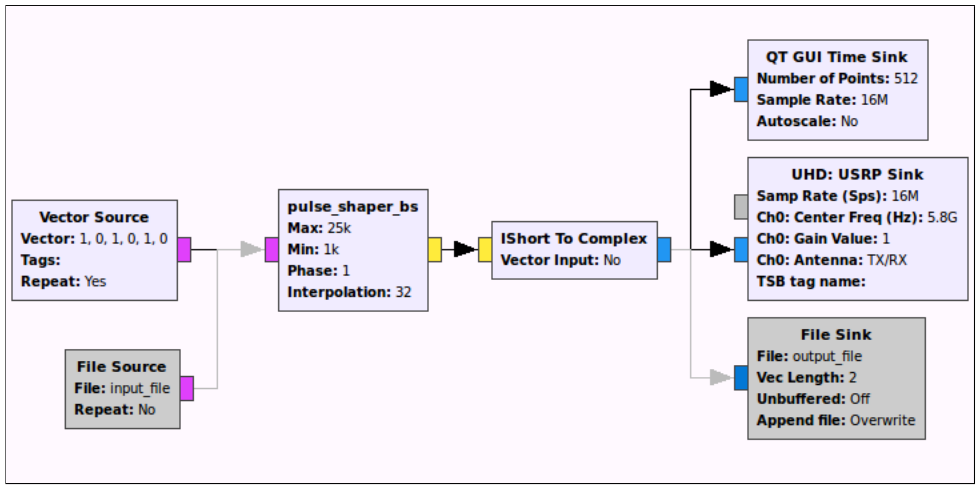


Figure 4.4: The signal processing blocks from the GRC transmitter flow chart.

Most of the variables in the program also needed to be represented in my flow chart. In GRC, variables are added as simple blocks containing only an ID and a value. These were the variables used in Thorsrud's program:

- `frequency`: The center frequency for the USRP sink block. Thorsrud used 0 in his program, because of his specialized radio attachment that changed the frequency later. I set this value to 5.8 GHz.
- `gain`: The USRP transmitter gain was set to 1.0.
- `bit_per_second`: The bit rate was 500 kbit/s, as decided by the DSRC downlink parameters.
- `usrp_samples_per_second`: This variable is limited by the USRP DAC, which handles 400 MS/s in my case. In Thorsrud's program this was set to 128 MS/s.
- `v_min`: Minimum signal value for the pulse shaper. Set to 1000, which Thorsrud used.

- `v_max`: Maximum signal value for the pulse shaper. Also set to the same value as in Thorsrud's program, 25000.
- `phase`: For the pulse shaper, must be either 1 or -1. I set it to 1.
- `pulse_shaper_interpolation`: This variable decides the number of samples per symbol for the pulse shaper. The USRP N200 with Gigabit Ethernet can handle up to 25 MS/s, so with 16-bit samples and a bitrate of 500 kbit/s, this gives the maximum pulse shaper interpolation

$$ps\_interpolation = \frac{25MS/sec}{500kbit/sec} = 50samples/bit \tag{4.1}$$

  I set this value to 32, as it needed to be able to divide 512. To compare, Thorsrud used 16 for this value.
- `interpolation_rate`: In Thorsrud's transmitter, the interpolation rate is calculated with the following equation:

$$interpolation = \frac{2 * usrp\_samples\_per\_second}{bit\_per\_second * pulse\_shaper\_interpolation} \tag{4.2}$$

  However, in my version of GNU Radio, the USRP sink block did not take interpolation as a parameter. Instead, it needed the sample rate directly. I chose to calculate this as the `bit_per_second` times the `pulse_shaper_interpolation`, which gave 16 MS/s.

Additionally, Thorsrud's program took some input from the user as parameters in the command line. All of these were not necessary for me, but the following parameters were included as GUI entries in my GRC program:

- `input_file`: The file name of the input data file.
- `output_file`: The file name of the output data file.

Figure 4.5 shows all the variables and GUI entries used in the transmitter flow chart. The complete GRC flow chart is attached in B.1.

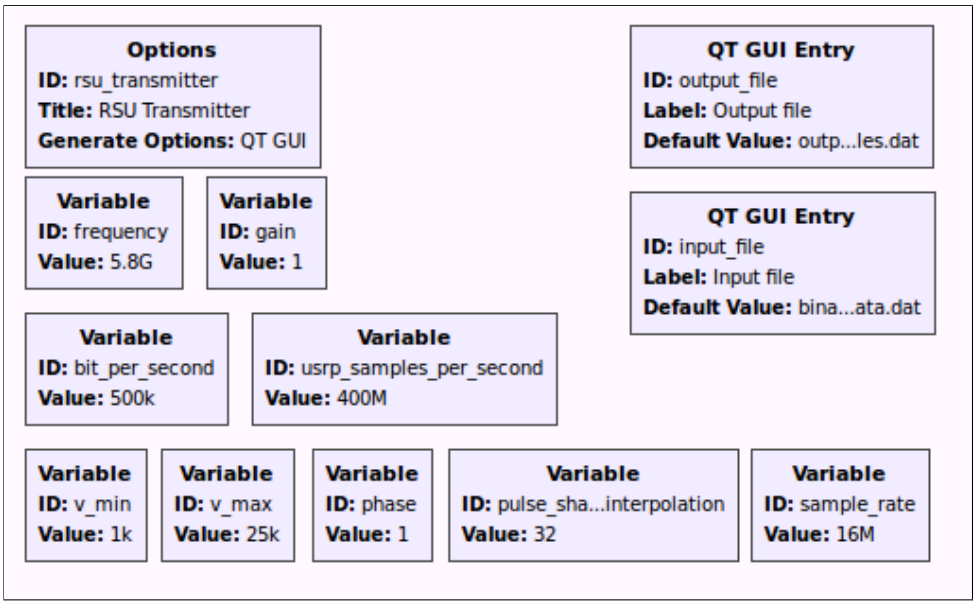


Figure 4.5: The variables and GUI entries from the GRC transmitter flow chart.

Receiver Flow Chart

For implementing the receiver flow chart, I used the same approach as with the transmitter. First, all the blocks used in `dsrc_rsu_receiver.py` was identified:

- `gr.mpsk_receiver_cc`: The equivalent to this block in my version of GNU Radio was `digital.mpsk_receiver_cc`. This block took a lot of parameters, which were filled in directly instead of creating variable blocks. The following parameters taken from Thorsrud's receiver were entered:
  - `M` = 2
  - `Theta` = 0
  - `Min Freq` = -0.00393
  - `Max Freq` = 0.00393
  - `Mu` = 0.5
  - `Gain Mu` = 0.05
  - `Omega` = samp_per_symb (a variable entered later)
  - `Gain Omega` = (samp_per_symb * samp_per_symb) / 4
  - `Omega Relative Limit` = 0.005

- `gr.complex_to_real`: A type converter block that takes the complex signal and removes the imaginary part, equivalent to `blocks.complex_to_real`.
- `gr.binary_slicer_fb`: A block used as a decision device, equivalent to `digital.binary_slicer_fb`.
- `dsrc.nrzi_to_nrz_bb`: The encoder block I have implemented my own version of, found in the `dsrc_mod` module.
- `gr.freq_xlating_fir_filter_ccf`: A channel filter that is used for down-conversion to baseband and lowpass filtering. The block is identical in my version of GNU Radio, located in the `filter` module. This filter takes another block called `lowpass_coeff` as a parameter, containing the following filter taps values:
  - `Gain` = 1.0
  - `Sample Rate` = input_rate
  - `Cutoff Freq` = 100k
  - `Transition Width` = 100k
  - `Window` = Hamming
  - `Beta` = 6.76 (Unchanged, as this value is not used with my window type)
- `usrp.source_c`: The USRP source block, corresponding to `uhd.usrp_source`. Thorsrud used 0 as center frequency for his block, because of his radio attachment, while I set this to 5.8 GHz.
- `gr.file_source`: A file source for reading an input file when not using the USRP.
- `gr.file_sink`: One file sink for the decoded data, plus four file sinks used for logging.

Next, the connections between the blocks were constructed, before the variable blocks were created. The signal processing blocks and their connections are shown in Figure 4.6. The following variables used in Thorsrud's receiver needed to be included in my program:

- `gain`: The USRP gain, set to 1.0.
- `symbol_per_second`: The bit rate, which was set to 250 kbit/s, following the DSRC uplink parameters.
- `usrp_samples_per_second`: The samples per second limit for the ADC, which is 100 MS/s for the N200.

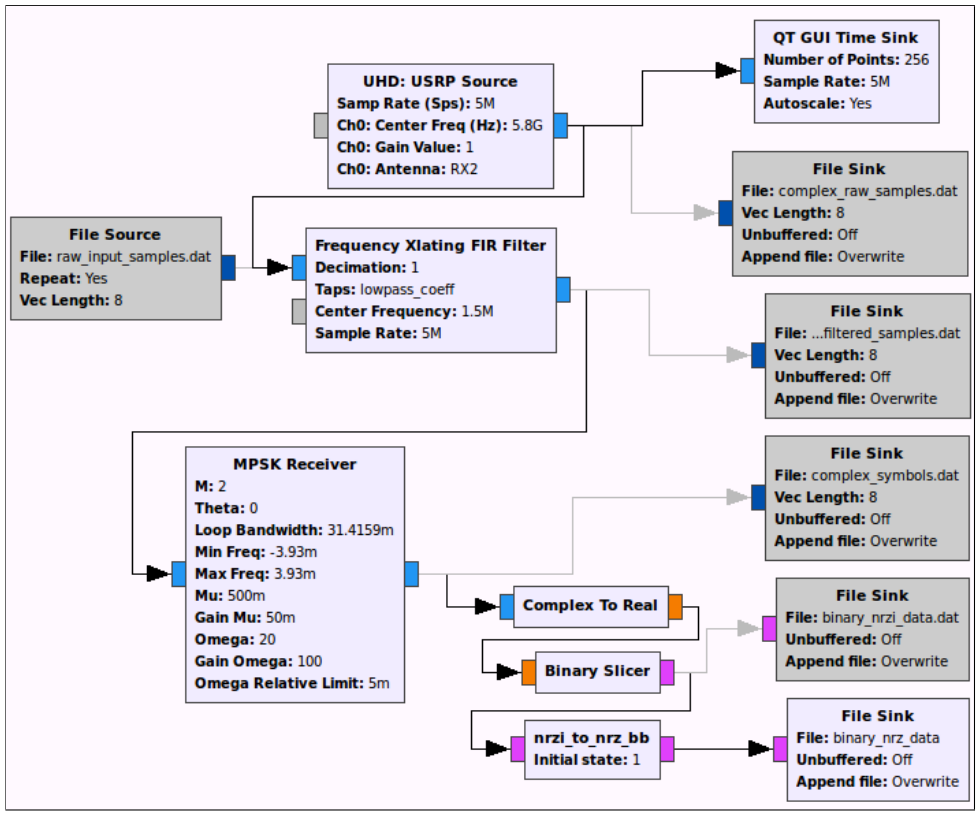


Figure 4.6: The signal processing blocks from the GRC receiver flow chart.

- `input_rate`: The sample rate from the ADC is decimated from 100 MS/s to this input rate. I chose 5 MS/s, to get an even decimation rate of 20 and a fitting number of samples per symbol.
- `samp_per_symb`: The samples per symbol could be calculated from `input_rate` (Samples per second) and `symbol_per_second`, resulting in the value 20. Thorsrud used 16 for this value, but stated that the synchronizing blocks could work with as little as 2 samples per symbol.

Similarly to the transmitter, Thorsrud's receiver took some parameters from the command line. Most of these were not needed in my flow chart, but I added a GUI entry for the parameter `subcarrier_freq`. The variable and GUI blocks from the program are depicted in Figure 4.7. For the complete GRC receiver flow chart, see B.2.

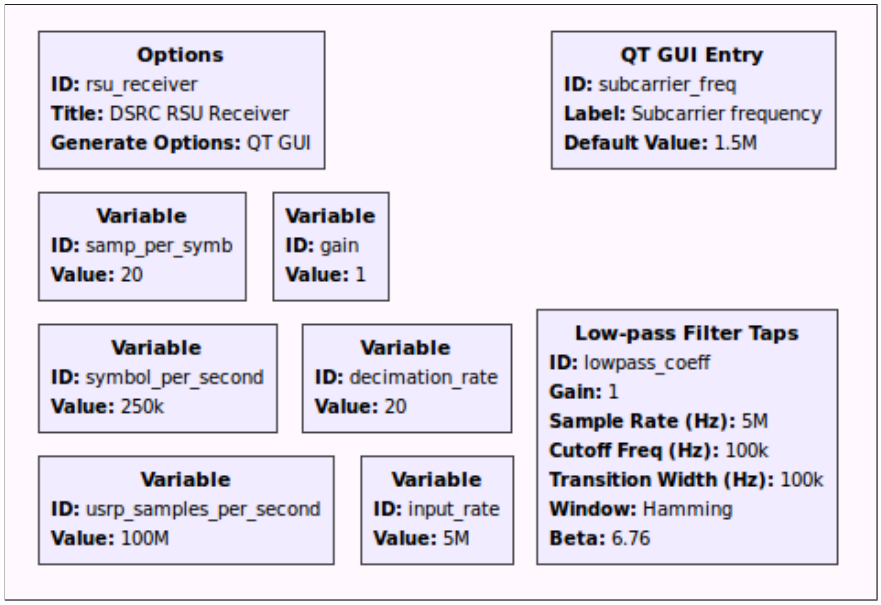


Figure 4.7: The variables and GUI entries from the GRC receiver flow chart.

4.2 Results and Discussion

4.2.1 DSRC Module

All the implemented GNU Radio blocks in the DSRC module have been tested with QA test files. These files are included alongside the source code for the blocks in the module's Python folder. The input and expected output values for the tests were taken from Thorsrud's QA files, to make sure the recreated blocks behaved as the original blocks. Thorough testing showed that the signal processing blocks functioned properly as they were supposed to.

4.2.2 Transmitter Flow Chart

The DSRC transmitter flow chart was tested with the USRP and several different input values. I had an AutoPASS OBU chip available, but this was difficult to use for testing, as it gave no visual feedback to the signals. Thorsrud used a chip with LED lights for testing, which showed when the chip received a modulated carrier wave. This way he could get feedback even when the chip did not recognize any message and send a reply. To be able to look at the signal my program sent, I added a GUI sink to the flow chart. A screenshot of the transmitter program running is included in Figure 4.8. In this screenshot the input was an alternating sequence of

1's and 0's. The signal plot shows that the signal is FM0 encoded correctly, and the similarities to Thorsrud's signal plots in [Tho09] are clear.

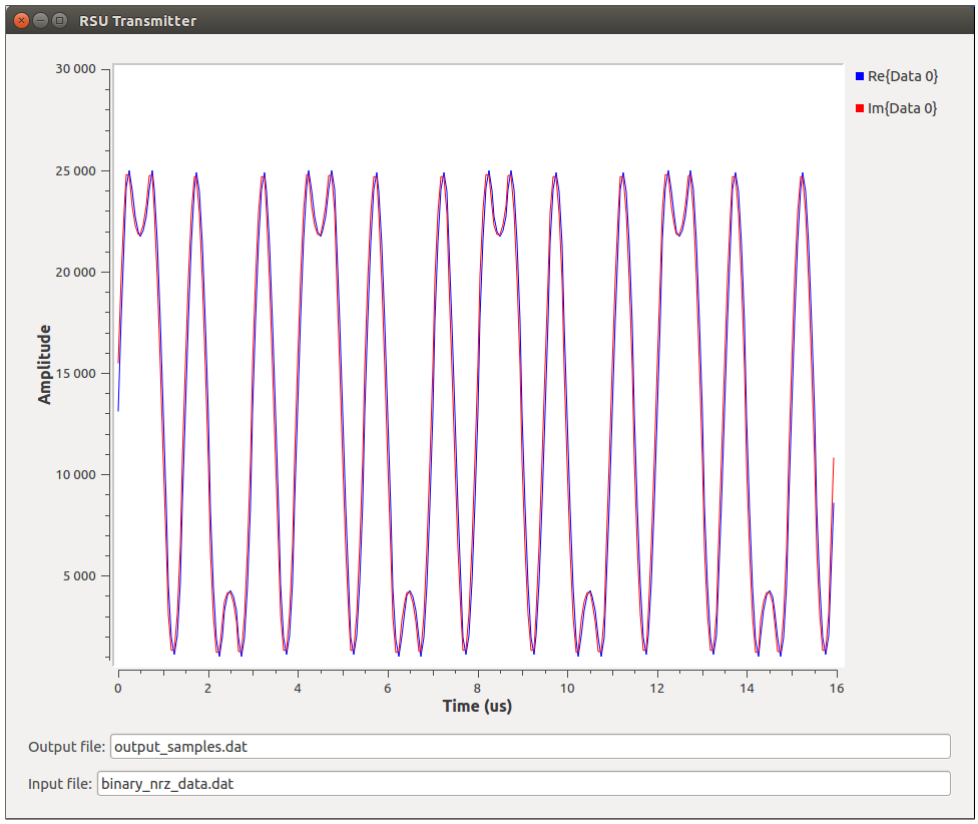


Figure 4.8: Screenshot of the transmitter program running.

However, there were some difficulties with selecting the correct values for the variables in used in the transmitter. The values related to the USRP had to be altered from Thorsrud's program, because of the different hardware. Additionally, many of the used blocks were changed in the newer version of GNU Radio, so the parameters needed were not always the same. For instance, the new USRP sink block required "Samp Rate (Sps)" as a parameter, while the old block required "Interpolation." Both values decide the USRP sample rate, but in different ways. Then the equations used by Thorsrud were not applicable, the values needed to be calculated differently. After a lot of experimenting and different calculations, I ended up with the values presented, which seemed to be suitable.

It should also be mentioned that when running the GNU Radio flow chart for the first time after booting the computer, the UHD warning messages displayed below

showed up each time. This also happened for the receiver flow chart. The messages warn about too small send and receive buffers, and the problem was fixed by running the commands suggested.

```
UHD Warning:
    The recv buffer could not be resized sufficiently.
    Target sock buff size: 50000000 bytes.
    Actual sock buff size: 1000000 bytes.
    See the transport application notes on buffer resizing.
    Please run: sudo sysctl -w net.core.rmem_max=50000000

UHD Warning:
    The send buffer could not be resized sufficiently.
    Target sock buff size: 1048576 bytes.
    Actual sock buff size: 1000000 bytes.
    See the transport application notes on buffer resizing.
    Please run: sudo sysctl -w net.core.wmem_max=1048576
```

4.2.3 Receiver Flow Chart

The receiver was even more difficult to test than the transmitter, since it needed signals from an OBU. QA tests of the blocks showed that they worked as they were supposed to, and running the flow chart resulted in no errors. In order to get some signals for the receiver, one thought was to run the transmitter simultaneously. The USRP supports simultaneous signal transmission and reception, so this could be a way to produce input for testing. However, only one GNU Radio flow chart can be connected to a USRP at the time. Since I only had one USRP available with the correct daughterboard, I needed to combine the transmitter and receiver in one GNU Radio program. This was done by copying the GRC blocks from `rsu_transmitter.grc` and `rsu_receiver.grc` into one chart called `dsrc_transceiver.grc`. The new flow chart was tested with several different parameter values. Since the transmitter and receiver were not created to communicate with each other, but with an OBU, some additional modifications were necessary. The programs were originally set to use different frequencies, bitrates, and encoding.

Some testing with these programs were done, but without producing any concrete, conclusive results. Because of a shortage of time, I was then forced to move on with the next part of the thesis. With more time it would have been possible to do more thorough tests of both the transmitter and the receiver, and more modifications to make them communicate with each other. Other equipment could also be helpful for testing, for instance an oscilloscope for measuring the signals sent from the transmitter. Similarly, a signal generator could be used for creating signals to the

receiver.

Chapter 5
Brute-Force Attacks on DES Keys from EFC

5.1 Literature Review

For the second part of the thesis, the literature review began by looking at the DES algorithm. The history of the standard, the algorithm design and security issues were studied, and presented in 2.6. Next, I looked at different types of brute-force attacks against DES, to see which that could be used against the encryption in EFC. The time-memory trade-off algorithms described in 3.2 were studied in particular, in addition to papers comparing these methods.

Then the security weaknesses of the EFC standards were investigated, especially the use of DES encryption. A CEN standard proposal from November 2015, called "*Electronic Fee Collection - Assessment of security measures for applications using dedicated short-range communication*" [CEN15], was helpful for this part. This technical report includes a detailed threat analysis for EFC systems using DSRC. Additionally, a draft paper from my supervisor Tord I. Reistad [Rei] was reviewed. His paper focuses specifically on the EN 15509 standard, and looks at how the DES encryption can be broken with brute-force attacks.

5.2 Attacks on EFC using customized DSRC devices

The available attack targets relevant to this thesis are the ones where DES encryption is involved. Necessarily, this means the calculations of Message Authentication Code (MAC) values and access credentials described in 2.3.1. The threat analysis in [CEN15] also points out this, and identifies stolen access credential keys and authentication keys by brute-force attacks as threat T1 and T2.

5.2.1 Obtaining MAC Values

MAC values can be obtained from an OBU by using a customizable RSU. The attacker can then send a carefully constructed GET_STAMPED request, and receive

the desired MAC value in the reply. Parameters required in a GET_STAMPED request are described in EN 14906 [CEN11], and presented in Table 5.1. The most important parameters to notice are ActionType, AccessCredentials, AttributeIdList and Nonce. For this attack we use the value 0x00 for both ActionType and AttributeIdList, specifying the GET_STAMPED command and an empty attribute list. AccessCredentials are not needed when AttributeIdList is set to zero.

Parameter name	**ASN.1 type**
Element Identifier EID	Dsrc-EID
ActionType	INTEGER(0..127,...)
AccessCredentials	OCTET STRING
ActionParameter	GetStampedRq ::= SEQUENCE { attributeIdList AttributeIdList, nonce OCTET STRING, keyRef INTEGER(0..255) }
Mode	BOOLEAN

Table 5.1: Overview of parameters in a GET_STAMPED.request

As presented in 2.3.1, the OBU computes the MAC value over a message M containing the AttributeIdList and the RndRSE nonce with padding. When choosing an empty attribute list in the request, the MAC is calculated from the 8 bytes listed in Table 5.2. This MAC value is then included as a part of the GET_STAMPED response to the RSU.

Name	**Value**
AttributeIdList	0x00
RndRSE length	0x04
RndRSE	4 arbitrary bytes
Padding	0x00 0x00

Table 5.2: The parameters used as input for the MAC computation, when an empty attribute list is requested. The 4 bytes for the RndRSE nonce can be chosen by the attacker.

Consequently, the attacker is able to obtain the ciphertext to a chosen plaintext. By choosing different values for the 4 byte nonce, it is even possible to retrieve several plaintext-ciphertext pairs. A chosen plaintext attack like this makes it possible to do precomputations, to speed up a brute-force attack.

5.2.2 Obtaining Access Credentials

By using a customizable OBU, it can be possible to obtain access credentials from a legitimate RSU. This is done by making the OBU send a response to a BST. When transmitting a customized, but still valid, VST, an attacker is able to receive a new command from the RSU containing access credentials. This procedure is illustrated in Figure 5.1, and can be recognized as part of the transaction from Figure 2.2.

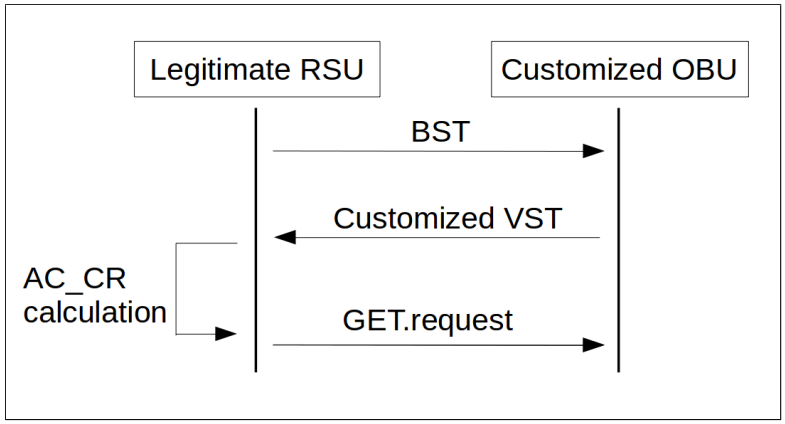


Figure 5.1: The procedure for obtaining access credentials with a fake OBU. The GET.request may also be a GET_STAMPED.request or SET.request.

As explained in 2.3.1, the access credential AC_CR is computed over the RndOBE included in the VST. Annex B in EN 14906 [CEN11] provides a detailed example of the contents of a VST. The most important parameter for this attack is the ApplicationContextMark. Since we want a response with access credentials, security level 1 is used. Then the ApplicationContextMark consists of a 6 byte EFC-ContextMark, a 2 byte AC_CR-KeyReference, and a 4 byte RndOBE. The EFC-ContextMark includes ContractProvider, TypeOfContract, and ContextVersion, and needs to be recognized by the RSU in order to not be discarded. To make sure these values are legitimate, the VST from an authentic OBU should be studied. If the RSU acknowledges the message, it will respond with a GET, SET, or GET_STAMPED request. This request will include the AC_CR, calculated from the RndOBE chosen by the attacker concatenated with four zero-bytes. Similarly to the MAC value, this method makes it possible to obtain chosen plaintexts and their corresponding ciphertexts.

5.2.3 Brute-force Attack against Access Credential and MAC Keys

5.2.1 and 5.2.2 have explained how a chosen plaintext attack is feasible against both the authentication keys and the access credential keys in EFC. For the authentication keys, the plaintext must be of the format '00 04 xx xx xx xx 00 00'H, where xx is a freely chosen byte. Similarly, the plaintext for access credential keys must be of the format 'xx xx xx xx 00 00 00 00'H. By combining these, we see that the plaintext format '00 04 xx xx 00 00 00 00'H is usable for an attack on both the key types. This means that a precomputed hash table like those described in 3.2 can be used for brute-forcing both authentication keys and access credential keys.

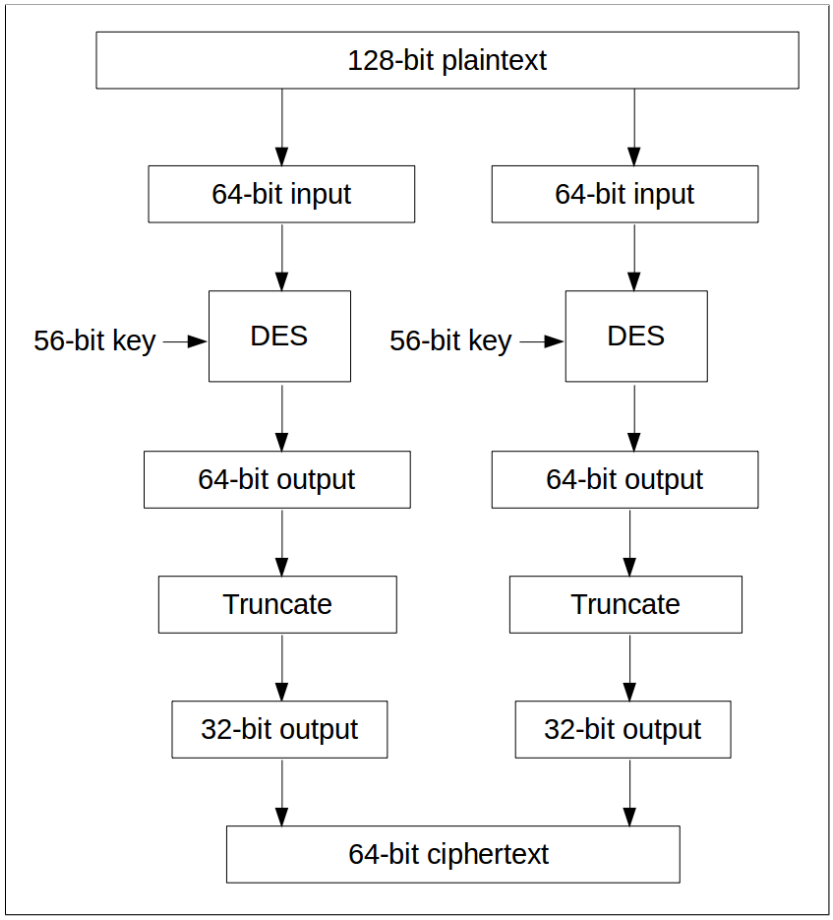


Figure 5.2: Illustration of the DES encryption for a modified brute-force attack on MAC and access credential keys in EFC.

However, a brute-force attack on the DES keys used in EFC is not as straightforward as a standard attack on DES. The reason for this is the truncation of the output after the encryption, which results in a MAC value or access credential consisting of only 4 bytes. Consequently, one plaintext/ciphertext pair is not enough to determine the key. To make up for this, at least two pairs must be obtained, as explained in [Rei]. This is possible, since the third and fourth bytes in the plaintext can be chosen freely. Two DES encryptions with the same key are then performed, and the two outputs are combined to get 64 bits. The input to the computation is 128 bits, containing two 64-bit plaintexts. This procedure is illustrated in Figure 5.2 on the previous page.

The modification of the brute-force attack has to be considered also when doing precomputations for time-memory trade-off algorithms. Because of the double DES encryption, both the creation of hash tables and searching will require extra calculations. However, if the calculations can be done in parallel, it does not necessarily mean that much extra time will be used. To demonstrate how such an attack can be implemented, I created an example using Python. In this example, a method called `double_encrypt` performs the specialized encryption using two plaintexts. The method takes the two 64-bit plaintexts and a 64-bit key as input, and returns a 64-bit ciphertext:

```
def double_encrypt(p0,p1,key):
    #Create DES object with the new key
    k = des(key)

    #First encryption
    c0 = k.encrypt(p0)
    #Truncate output
    c0 = c0.encode("hex")[:8]

    #Second encrytion
    c1 = k.encrypt(p1)
    #Truncate output
    c1 = c1.encode("hex")[:8]

    #Concatenate the two outputs
    c = (c0+c1).decode("hex")

    return c
```

In this code the Python package pyDes [Whi] is used for DES encryption. My code is not written with speed and performance in mind, but rather to give a concrete

example of how the modified brute-force attack against keys in EFC could look. The example creates precomputed hash tables where both the number of chains and the length of the chains can be chosen. This is done in the method `create_table`, which takes `length` and `chainlength` as input:

```
def create_table(length,chainlength):

    table = [None] * length

    for i in range(length):

        #Randomly selected startkey for each chain
        startkey = os.urandom(8)
        key = startkey

        for j in range(chainlength):
            #Plaintexts for the two DES encryptions
            p0 = '\x00\x04\x00\x00\x00\x00\x00\x00'
            p1 = '\x00\x04\xff\xff\x00\x00\x00\x00'

            c = double_encrypt(p0,p1,key)
            key = c

        #Add startkey and endkey to the table
        table[i] = (startkey,key)
        print table

    #Sort table by endkeys
    table = sorted(table, key=lambda pair: pair[1])
    return table
```

In this implementation it is worth noticing that no defined reduction function is used. The 64-bit output is used directly as the next key. This is possible because the DES implementation from pyDes does not consider the parity bits. It requires a 64-bit key, and the 8 parity bits are simply not used in the computation. A reduction from 64 to 56 bits is therefore automatically done, as those 8 bits are removed in practice. The plaintexts `p0` and `p1` are entered directly in the code. They are both valid plaintexts for an attack against access credentials and MAC values, as explained previously. To choose the random startkey for each chain, the Python module `os` is used to pick 8 random bytes. After the complete table is created, is it sorted on the endkeys. For an implementation focusing on speed, the sorting should be done directly when a new table entry is added.

The last method in my example is `reverse_hash`, which takes a ciphertext hash and searches through a precomputed table to find the corresponding key. Along with the hash, the table and the chainlength are also taken as input.

```
def reverse_hash(hash, table,chainlength):

    #Plaintexts for the two DES encryptions
    p0 = '\x00\x04\x00\x00\x00\x00\x00\x00'
    p1 = '\x00\x04\xff\xff\x00\x00\x00\x00'

    nexthash = hash

     #If no matching endkey is found within the chainlength,
     # the table does not contain the wanted key
    for y in range(chainlength):

        key = nexthash

          #Search through all the endkeys
        for i in range(len(table)):

            if key==table[i][1]:
                 #Matching endkey found
                endkey = key
                startkey = table[i][0]

                #Recreate chain from startkey to the wanted key
                for j in range(chainlength-y-1):
                    c = double_encrypt(p0,p1,startkey)
                    startkey = c
                #The wanted key is found
                return c

        nexthash = double_encrypt(p0,p1,key)

    print("Key not found")
    return None
```

This method implements the search procedure explained in 3.2.1. Similarly to the last method, the two plaintexts are entered directly in the code. Since the reduction function is automatically done at encryption, the search can start by directly comparing the given hash against the endkeys. It then continues encrypting

with `double_encrypt`, until a matching endkey is found. When this happens, the hash chain is recreated until the wanted key is reached, and this key is then returned. The complete code for the example is attached in C.1. This code includes more prints for feedback, and the hex values are encoded to be more readable. Additionally, a small precomputed table and a hash from one of the chains are included for testing. The table is of size 100x100 and was cut from this appendix because of space, but is included in the source code accompanying this thesis.

Making The Attack Faster

If an actual attack against these DES keys are to be carried out, the implementation will have to be optimized for speed and performance. Because of all the DES computations involved, the chosen DES implementation is extremely important when it comes to speed. The fastest implementation known is the Bitslice DES [Kwa], which was first presented by Eli Biham in 1997 [Bih97]. This specialized variant of the encryption standard has been improved repeatedly since it was introduced, and people are still working with making faster implementations.

Another element critical for the speed, is the platform the program is implemented for. To achieve the desired performance for an attack like this, it is definitely recommended to use hardware like a graphics processing unit (GPU) or an FPGA. GPU programming is popular for performance dependent implementations, and several guides and frameworks exist for this. NVIDIA CUDA [CUD] is a platform for parallel computing with GPUs, which would be suitable for creating this implementation.

Finally, the optimal time-memory trade-off algorithm to use with the brute-force attack should be chosen. As presented in 3.2, both the distinguished points method and rainbow tables are great improvements to the standard precomputed tables. There exist a lot of research comparing the algorithms to see which method is the most efficient, where the most recent paper is [LH15] from 2015. This article concludes that the perfect rainbow table trade-off is advantageous over perfect distinguished points and the other algorithms. Perfect tables means tables without any identical end points.

Chapter 6
Conclusion

This Master's thesis has looked at the possibilities of using a USRP to create a wireless test-bed for DSRC applications. Software programs for transmitting and receiving DSRC messages have been implemented using the GNU Radio framework. Custom signal processing blocks for these programs have been created, and tests show that these blocks function properly. The transmitter seems to be working when looking at the created signal, but this has not been confirmed with actual DSRC equipment. Similarly, I have not been able to test the receiver with original DSRC signals, so no clear conclusion about the program can be made. Even though no definite results regarding communication with EFC entities have been produced, the work with these programs has given the impression that such communication should definitely be possible. With enough knowledge of signal processing and the DSRC specifications, a USRP in combination with a framework like GNU Radio is all that is needed.

Furthermore, the EFC system and its security has been especially focused on in this thesis. Security weaknesses have been pointed out, concentrating on the use of DES encryption. The DES algorithm design has been presented, and it has been explained why this standard is not secure. Previous brute-force attacks against DES keys have been introduced, and time-memory trade-off algorithms to speed up such attacks are thoroughly described in 3.2.

Chapter 5 specifies concrete attacks against EFC, which take use of customized DSRC transmitters and receivers posing as RSUs and OBUs. These attacks make it possible to perform chosen plaintext attacks against the DES keys used for MAC and access credential calculation in EFC. A specially modified brute-force attack against these keys is described, and a code example implementing a simple version of this attack is presented. Time-memory trade-off algorithms making the attack more efficient are also discussed.

Further work would be to continue testing and development of the DSRC trans-

mitter and receiver, to verify their functionality. As mentioned in 4.2, additional equipment like an oscillator and a signal generator could be helpful for this. For the second part with DES cracking, a possible next step could be to attempt a more substantial implementation of the brute-force attack focusing on speed and performance, using hardware programming and faster trade-off algorithms.

References

[BB97] Eli Biham and Alex Biryukov. An Improvement of Davies' Attack on DES. *Journal of Cryptology*, 10(3):195–206, 1997.

[Bih97] Eli Biham. A fast new DES implementation in software. In *Fast Software Encryption*, pages 260–272. Springer Verlag, 1997.

[Blo04] Eric Blossom. GNU radio: tools for exploring the radio frequency spectrum. *Linux Journal*, 122, June 2004.

[BS93] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer Verlag, 1993.

[CEN03a] CEN. EN 12795:2002 Road transport and traffic telematics - Dedicated short-range communication (DSRC) – DSRC data link layer: medium access and logical link control. March 2003.

[CEN03b] CEN. EN 12834:2002 Road transport and traffic telematics - Dedicated short-range communication (DSRC) – DSRC application layer. November 2003.

[CEN04a] CEN. EN 12253:2004 Road transport and traffic telematics - Dedicated short-range communication – Physical layer using microwave at 5.8 GHz. July 2004.

[CEN04b] CEN. EN 13372:2004 Road transport and traffic telematics (RTTT) - Dedicated short-range communication – Profiles for RTTT applications. July 2004.

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

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

[CEN15] CEN. Electronic Fee Collection - Assessment of security measures for applications using dedicated short-range communication. November 2015.

[CUD] CUDA Parallel Computing Platform. http://www.nvidia.com/object/cuda_home_new.html.

[Den82] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982. p. 100.

[DES] DESCHALL Project. http://www.interhack.net/projects/deschall/. Accessed: 20 May 2016.

[DH77] Whitfield Diffie and Martin E. Hellman. Exhaustive Cryptanalysis of the NBS Data Encryption Standard. *Computer*, 10(6):74–84, July 1977.

[Ett] About Ettus Research. https://www.ettus.com/site/about. Accessed: 12 May 2016.

[GRw] GNU Radio Wiki. http://gnuradio.org/redmine/projects/gnuradio/wiki. Accessed: 12 May 2016.

[Hel80] Martin E. Hellman. A Cryptanalytic Time-Memory Trade-Off. *IEEE Transactions on Information Theory*, 26(4), July 1980.

[ISO94] ISO. ISO/IEC 7498-1:1994 Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model. November 1994.

[Kar15] Nikos Kargas. Gen2 UHF RFID Reader. https://github.com/nikosl21/Gen2-UHF-RFID-Reader, December 2015. Accessed: 7 Mar 2016.

[KN13] A. Kumar and S. Noghanian. Wireless Channel Test-bed for DSRC Applications using USRP Software Defined Radio. *2013 IEEE Antennas and Propagation Society International Symposium (APSURSI)*, July 2013.

[Kwa] Matthew Kwan. Bitslice DES. http://www.darkside.com.au/bitslice/.

[Lab] LabVIEW Communications for USRP (SDR). http://www.ni.com/labview-communications/usrp/. Accessed: 15 May 2016.

[LH15] Ga Won Lee and Jin Hong. Comparison of perfect table cryptanalytic tradeoff algorithms. *Designs, Codes and Cryptography*, pages 1–51, 2015.

[Mat93] Mitsuru Matsui. Linear Cryptanalysis Method for DES Cipher. In *Advances in Cryptology - EUROCRYPT '93*. Springer Verlag, 1993.

[N20] USRP N200 Product details. https://www.ettus.com/product/details/UN200-KIT. Accessed: 14 May 2016.

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

[NIS99] NIST. FIPS 46-3 - Data Encryption Standard (DES). October 1999.

[Oec03] Philippe Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 617–630. Springer Berlin Heidelberg, 2003.

[Rei] Tord Ingolf Reistad. Securing EN 15509 against brute-force attacks. Norwegian Road Administration (Statens Vegvesen).

[RIV] COPACOBANA RIVYERA. http://www.sciengines.com/copacobana/. Accessed: 20 May 2016.

[Tho09] Einar Thorsrud. Programvaredefinert radio - Mulige hyllevareløsninger for DSRC-anvendelser. Master's thesis, NTNU, 2009.

[UHD] USRP Hardware Driver and USRP Manual. http://files.ettus.com/manual/. Accessed: 16 May 2016.

[Whi] Todd Whiteman. pyDes - Pure Python DES encryption algorithm. http://twhiteman.netfirms.com/des.html.

Appendix A
DSRC Module

The following copyright declaration is included at the beginning of all of the created GNU Radio files. It has been excluded from the code in the appendices to save space.

```
#
# Copyright 2016 Jonathan Hansen.
#
# This is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public
# License as published by the Free Software Foundation;
# either version 3, or (at your option) any later version.
#
# This software is distributed in the hope that it will be
# useful, but WITHOUT ANY WARRANTY; without even the implied
# warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
# PURPOSE. See the GNU General Public License for more
# details.
#
# You should have received a copy of the GNU General Public
# License along with this software; see the file COPYING.
# If not, write to the Free Software Foundation, Inc., 51
# Franklin Street, Boston, MA 02110-1301, USA.
#
```

A.1 NRZI to NRZ block

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy
from gnuradio import gr

class nrzi_to_nrz_bb(gr.sync_block):
    """
    docstring for block nrzi_to_nrz_bb
    """
    def __init__(self, preload):
        self.preload = preload
        gr.sync_block.__init__(self,
            name="nrzi_to_nrz_bb",
            in_sig=[numpy.int8],
            out_sig=[numpy.int8])


    def work(self, input_items, output_items):
        in0 = input_items[0]
        out = output_items[0]
        prev_nrzi_bit = self.preload
        nrzi_bit = 0
        nrz_bit = 0

        for i in range(len(out)):
            nrzi_bit = in0[i]
            #Convert NRZI to NRZ
            if(nrzi_bit != prev_nrzi_bit):
                nrz_bit = 0
            else:
                nrz_bit = 1
            out[i] = nrz_bit
            prev_nrzi_bit = nrzi_bit

            return len(output_items[0])
```

A.2 NRZ to NRZI block

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy
from gnuradio import gr

class nrz_to_nrzi_bb(gr.sync_block):
    """
    docstring for block nrz_to_nrzi_bb
    """
    def __init__(self, preload):
        self.preload = preload
        gr.sync_block.__init__(self,
            name="nrz_to_nrzi_bb",
            in_sig=[numpy.int8],
            out_sig=[numpy.int8])


    def work(self, input_items, output_items):
        in0 = input_items[0]
        out = output_items[0]
    nrzi_bit = 0
    nrz_bit = 0
    prev_nrzi_bit = self.preload

    for i in range(len(out)):
        nrz_bit = in0[i]

        #Convert NRZ to NRZI
        if(nrz_bit == 0):
            nrzi_bit = prev_nrzi_bit ^ 1
        else:
            nrzi_bit = prev_nrzi_bit
        out[i] = nrzi_bit
        prev_nrzi_bit = nrzi_bit

        return len(output_items[0])
```

A.3 Pulse shaper block

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy
from gnuradio import gr

class pulse_shaper_bs(gr.interp_block):
    """
    docstring for block pulse_shaper_bs
    """
    def __init__(self, v_min, v_max, phase, pulse_shaper_interpolation):
        gr.interp_block.__init__(self,
            name="pulse_shaper_bs",
            in_sig=[numpy.int8],
            out_sig=[numpy.short], interp=pulse_shaper_interpolation)
    self.min = v_min
    self.max = v_max
    self.phase = phase
    self.ps_interpolation = pulse_shaper_interpolation


    def work(self, input_items, output_items):
        in0 = input_items[0]
        out = output_items[0]
    ninput_items = len(output_items[0]) / self.ps_interpolation

    #print "ps_interpolation = "+str(self.ps_interpolation)
    #print "ninput_items = "+str(ninput_items)

    HALF_SIN_LENGTH = 256
    half_sin = (0,   25,   50,   75,  100,  125,  150,  175, \
    200,  225,  250,  275,  300,  325,  349,  374, \
    399,  423,  448,  472,  497,  521,  545,  569, \
    593,  617,  641,  665,  688,  712,  735,  759, \
    782,  805,  828,  851,  874,  896,  919,  941, \
    963,  985, 1007, 1029, 1050, 1072, 1093, 1114, \
    1135, 1155, 1176, 1196, 1216, 1236, 1256, 1276, \
    1295, 1315, 1334, 1352, 1371, 1389, 1408, 1426, \
    1443, 1461, 1478, 1495, 1512, 1529, 1545, 1561, \
    1577, 1593, 1608, 1624, 1639, 1653, 1668, 1682, \
```

```
1696, 1710, 1723, 1736, 1749, 1762, 1774, 1786, \
1798, 1810, 1821, 1832, 1843, 1853, 1863, 1873, \
1883, 1892, 1901, 1910, 1919, 1927, 1935, 1942, \
1949, 1956, 1963, 1970, 1976, 1981, 1987, 1992, \
1997, 2002, 2006, 2010, 2014, 2017, 2020, 2023, \
2025, 2027, 2029, 2031, 2032, 2033, 2034, 2034, \
2034, 2034, 2033, 2032, 2031, 2029, 2027, 2025, \
2023, 2020, 2017, 2014, 2010, 2006, 2002, 1997, \
1992, 1987, 1981, 1976, 1970, 1963, 1956, 1949, \
1942, 1935, 1927, 1919, 1910, 1901, 1892, 1883, \
1873, 1863, 1853, 1843, 1832, 1821, 1810, 1798, \
1786, 1774, 1762, 1749, 1736, 1723, 1710, 1696, \
1682, 1668, 1653, 1639, 1624, 1608, 1593, 1577, \
1561, 1545, 1529, 1512, 1495, 1478, 1461, 1443, \
1426, 1408, 1389, 1371, 1352, 1334, 1315, 1295, \
1276, 1256, 1236, 1216, 1196, 1176, 1155, 1135, \
1114, 1093, 1072, 1050, 1029, 1007,  985,  963, \
941,  919,  896,  874,  851,  828,  805,  782, \
759,  735,  712,  688,  665,  641,  617,  593, \
569,  545,  521,  497,  472,  448,  423,  399, \
374,  349,  325,  300,  275,  250,  225,  200, \
175,  150,  125,  100,   75,   50,   25,    0)

half_sin2 = (25,   50,   75,  100,  126,  151,  176,  201, \
  226,  251,  276,  301,  325,  350,  375,  400, \
  424,  449,  473,  498,  522,  546,  570,  595, \
  619,  642,  666,  690,  714,  737,  760,  784, \
  807,  830,  853,  876,  898,  921,  943,  965, \
  988, 1009, 1030, 1052, 1073, 1095, 1116, 1137, \
 1158, 1178, 1199, 1219, 1239, 1259, 1279, 1298, \
 1318, 1337, 1356, 1374, 1393, 1411, 1429, 1447, \
 1465, 1482, 1499, 1516, 1533, 1550, 1566, 1582, \
 1598, 1614, 1629, 1644, 1659, 1673, 1688, 1702, \
 1716, 1729, 1743, 1756, 1768, 1781, 1793, 1805, \
 1817, 1828, 1839, 1850, 1861, 1871, 1881, 1891, \
 1901, 1910, 1919, 1927, 1936, 1944, 1951, 1959, \
 1966, 1973, 1979, 1986, 1992, 1997, 2003, 2008, \
 2012, 2017, 2021, 2025, 2028, 2032, 2035, 2037, \
 2039, 2041, 2043, 2045, 2046, 2046, 2047, 2047, \
 2047, 2046, 2046, 2045, 2043, 2041, 2039, 2037, \
 2035, 2032, 2028, 2025, 2021, 2017, 2012, 2008, \
```

```
 2003, 1997, 1992, 1986, 1979, 1973, 1966, 1959, \
 1951, 1944, 1936, 1927, 1919, 1910, 1901, 1891, \
 1881, 1871, 1861, 1850, 1843, 1834, 1826, 1818, \
 1810, 1802, 1793, 1784, 1776, 1769, 1761, 1753, \
 1745, 1738, 1731, 1726, 1720, 1714, 1709, 1704, \
 1698, 1693, 1688, 1682, 1677, 1672, 1667, 1662, \
 1657, 1652, 1648, 1643, 1638, 1633, 1629, 1624, \
 1620, 1616, 1611, 1607, 1603, 1599, 1595, 1591, \
 1587, 1583, 1580, 1576, 1572, 1569, 1565, 1562, \
 1559, 1555, 1552, 1549, 1546, 1543, 1540, 1538, \
 1535, 1532, 1530, 1527, 1525, 1523, 1520, 1518, \
 1516, 1514, 1512, 1511, 1509, 1507, 1506, 1504, \
 1503, 1502, 1500, 1499, 1498, 1497, 1496, 1495, \
 1495, 1494, 1494, 1493, 1493, 1492, 1492, 1492)

#print half_sin
#print half_sin2

step_length = (HALF_SIN_LENGTH * 2) / self.ps_interpolation
#print "Step length (128): "+str(step_length)
TABLE_LEVEL = 2047
scaling = float(self.max - self.min) / (TABLE_LEVEL *2)
#print "Scaling(1): "+str(scaling)
DC_offset = float(self.min) + TABLE_LEVEL * scaling + 0.5
#print "DC_offset(0.5): "+str(DC_offset)

#Verify that the interpolation factor is within range
assert (self.ps_interpolation >= 4 and self.ps_interpolation <= 512)
#Verify that the interpolation factor is dividable by 2
assert (self.ps_interpolation % 2 == 0)
#and that (HALF_SIN_LENGTH * 2) / interpol_fac is a valid int
assert ((HALF_SIN_LENGTH * 2) % self.ps_interpolation == 0)
#Verify that "sign" has a correct value
assert (self.phase == -1 or self.phase == 1)

      # <+signal processing here+>

for i in range(ninput_items):
    if(in0[i]==0):
        for j in range(self.ps_interpolation/2):
```

```
            out[self.ps_interpolation*i+j] = half_sin[step_length*j] *
                self.phase * scaling + DC_offset
            if(out[self.ps_interpolation*i+j] < 0):
                out[self.ps_interpolation*i+j] = out[self.ps_interpolation*i+j]
                    - 1
        for j in range(self.ps_interpolation/2, self.ps_interpolation):
            out[self.ps_interpolation*i+j] = -half_sin[step_length*j -
                HALF_SIN_LENGTH] * self.phase * scaling + DC_offset
            if(out[self.ps_interpolation*i+j] < 0):
                out[self.ps_interpolation*i+j] = out[self.ps_interpolation*i+j]
                    - 1
    elif(in0[i]==1):
        for j in range(self.ps_interpolation/2):
            out[self.ps_interpolation*i+j] = half_sin2[step_length*j] *
                self.phase * scaling + DC_offset
            if(out[self.ps_interpolation*i+j] < 0):
                out[self.ps_interpolation*i+j] = out[self.ps_interpolation*i+j]
                    - 1
        for j in range(self.ps_interpolation/2, self.ps_interpolation):
            out[self.ps_interpolation*i+j] =
                half_sin2[HALF_SIN_LENGTH*2 - step_length*j -1] *
                self.phase * scaling + DC_offset
            if(out[self.ps_interpolation*i+j] < 0):
                out[self.ps_interpolation*i+j] = out[self.ps_interpolation*i+j]
                    - 1
        self.phase = -self.phase

    #out[:] = in0
#print out
    return len(output_items[0])
```

Appendix B

GNU Radio Flow Charts

B.1 DSRC Transmitter

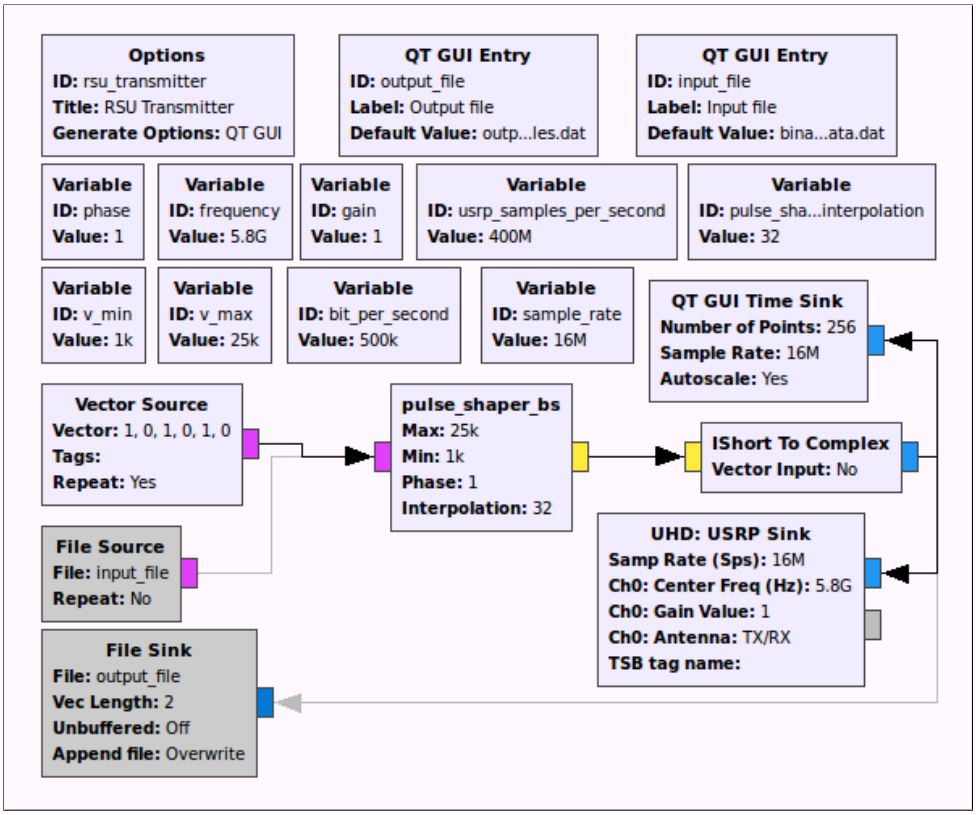


Figure B.1: The GRC transmitter flow chart.

B.2 DSRC Receiver

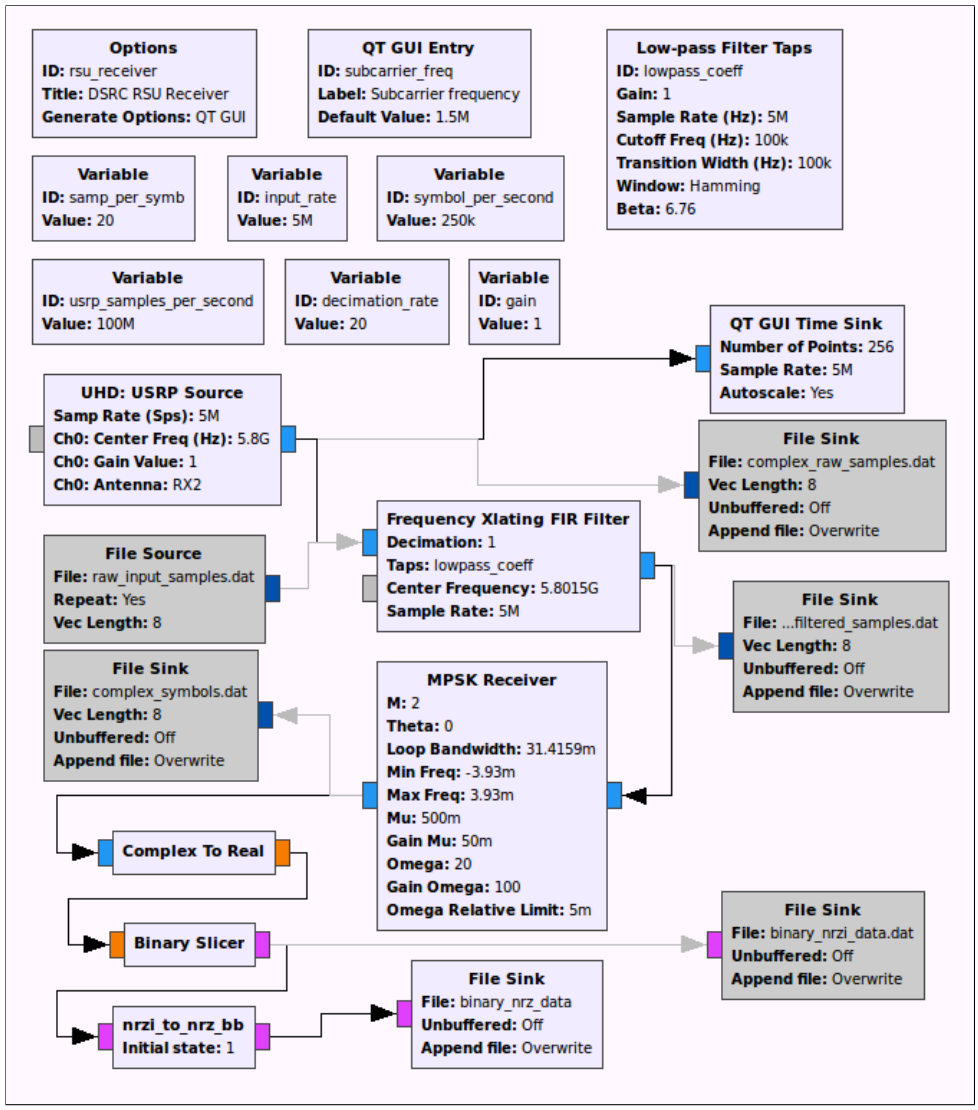


Figure B.2: The complete GRC receiver flow chart.

Appendix C
Brute-Force Example

C.1 modified_bruteforce.py

```
from pyDes import *
from time import time
import os

def double_encrypt(p0,p1,key):
    #Create DES object with the new key
    k = des(key)
    print ("Key: %r" % key.encode("hex"))

    #First encryption
    c0 = k.encrypt(p0)
    #Truncate output
    c0 = c0.encode("hex")[:8]
    print("c0: %r" % c0)

    #Second encrytion
    c1 = k.encrypt(p1)
    #Truncate output
    c1 = c1.encode("hex")[:8]
    print("c1: %r" % c1)

    #Concatenate the two outputs
    c = (c0+c1).decode("hex")
    print("Ciphertext: %r" % c.encode("hex"))

    return c
```

```
def create_table(length,chainlength):

    table = [None] * length

    for i in range(length):

        #Randomly selected startkey for each chain
        startkey = os.urandom(8)
        print("Random startkey: %r" % startkey)
        key = startkey

        for j in range(chainlength):
            #Plaintexts for the two DES encryptions
            p0 = '\x00\x04\x00\x00\x00\x00\x00\x00'
            p1 = '\x00\x04\xff\xff\x00\x00\x00\x00'

            c = double_encrypt(p0,p1,key)
            key = c
            print("")

        #Add startkey and endkey to the table
        table[i] = (startkey.encode("hex"),key.encode("hex"))
        print table

    #Sort table by endkeys
    table = sorted(table, key=lambda pair: pair[1])
    print("")
    print table
    return table

def reverse_hash(hash, table,chainlength):

    #Plaintexts for the two DES encryptions
    p0 = '\x00\x04\x00\x00\x00\x00\x00\x00'
    p1 = '\x00\x04\xff\xff\x00\x00\x00\x00'

    nexthash = hash

    #If no matching endkey is found within the chainlength,
     # the table does not contain the wanted key
    for y in range(chainlength):
```

```
        key = nexthash
        print ('Trying key nr %r: %r' % (y+1,key))

        #Search through all the endkeys
        for i in range(len(table)):
            if key==table[i][1]:
                #Matching endkey found
                print('Endkey found: %r' % key)
                endkey = key
                startkey = table[i][0]
                print('Startkey: %r' % startkey)
                startkey = startkey.decode("hex")

                #Recreate chain from startkey to the wanted key
                for j in range(chainlength-y-1):
                    c = double_encrypt(p0,p1,startkey)
                    startkey = c
                #The wanted key is found
                return c.encode("hex")

        nexthash = double_encrypt(p0,p1,key.decode("hex")).encode("hex")

    print("Key not found")
    return None

# This 100*100 table is added for testing,
# together with the hash value 'e15ed48ca2ab8752'
# and the corresponding key '1945d0157aabb5ea'.
# This hash is located in chain number 76,
# which has the endkey 'c54f238e17ba1fed'.
table = # Cut to save space. Included in the source code with the thesis.
hash = 'e15ed48ca2ab8752'

#t0 = time()
#table = create_table(100,100)
#t1 = time()
#tabletime = t1-t0
#print ('Table created in %ds' % tabletime)

t1 = time()
```

```
key = reverse_hash(hash,table,100)
t2 = time()

keytime = t2-t1
print('Key found: %r, in %ds' % (key,keytime))
```