NTNU
Norwegian University of
Science and Technology

Security Analysis of the Norwegian Toll Road System AutoPASS

Mari Bergendahl

Master of Science in Communication Technology
Submission date: June 2018
Supervisor: Stig Frode Mjølsnes, IIK
Co-supervisor: Tord Reistad, Vegvesenet

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

Title: Security Analysis of the Norwegian Toll Road System AutoPASS
Student: Mari Bergendahl

Problem Description:

AutoPASS is the Norwegian electronic fee collection (EFC) system used for both road tolling and ferry transport payments. The main components in EFC are the on-board unit (OBU) and the Roadside Equipment (RSE). The OBU and RSE uses Dedicated Short Range Communications (DSRC) as communication protocol.

The Data Encryption Standard (DES) is used for authentication in EFCs. This is a cipher that is considered to be insecure for many applications, and can be cracked by doing a brute force search on the key space. However, this is a very time consuming task, especially when the goal of the attack is to break a lot of keys.

For efficiently breaking a lot of keys and with few constraints on the pre-computation, using rainbow tables is a suitable solution. Rainbow tables was introduced in 2003 by Philippe Oechslin when he described a time-memory trade-off method for breaking hashes. Generating these rainbow tables is highly parallelizeble, making an attack suitable for graphical processor units (GPUs).

In this project rainbow tables, CPUs, and parallel GPU computations will be used to perform an attack on the encryption function, DES, and obtain keys in the shortest possible amount of time based on a simulated chosen plaintext attack.

Responsible professor: Stig Frode Mjølsnes, IIK
Supervisor: Tord Ingolf Reistad, Statens Vegvesen

Abstract

Electronic Fee Collection (EFC) systems are used overall to describe ICT solutions that automatically collects road user charges without the need of stopping. EFC systems consists of two main units; the on board unit (OBU) and the roadside equipment (RSE). These units uses the Dedicated Short Range Communications (DSRC) at 5.8 GHz as communication protocol.

The European standard EN 15509:2007 defines requirements for both the OBU, RSE and the DSRC, and it also specifies a set of transactions between the units. The GET_STAMPED messages are a central part of these transactions. It is through these messages the units authenticates themselves. The Message Authentication Code (MAC), included in the GET_STAMPED messages, is calculated based on an AttributeIDList, a Random Number and an Authentication Key.

The Data Encryption Standard (DES) is used in the calculation of the authentication codes. This is a cipher that is considered to be insecure for many applications, and can be cracked by doing a brute force search on the key space. Based on this, MACs are assumed obtained in this thesis, and an attack on these MACs are investigated.

Rainbow tables are used in this thesis in order to make an attack on a lot of message authentication keys efficient. A rainbow table are described as a time-memory trade-off for breaking keys in a chosen plaintext attack. A simple rainbow table are implemented i C++, and generation times are discussed.

Sammendrag

Elektronisk bompengeinnkreving (EFC) blir brukt for å beskrive IKT-løsninger som automatisk samler inn bompenger fra kjøretøy, uten at de trenger å stoppe. EFC systemer består av to hovedkomponenter: en bombrikke (OBU) og en bomstasjon (RSE). Disse komponentene bruker dedikert kortholdslink (DSRC) på 5.8 GHz som kommunikasjonsprotokol.

Den europeiske standarden EN 15509:2007 definerer ulike krav til både bombrikken, bomstasjonen og kommunikasjonsprotokollen. Den spesifiserer også et sett med transaksjoner mellom komponentene. GET_STAMPED-meldingene er en sentral del av disse transaksjonene. Det er gjennom disse meldingene komponentene autentiserer seg. Meldingsautentiseringskoden (MAC), som er inkludert i GET_STAMPED-meldignene er kalkulert baset på en AttributtIDListe, et tilfeldig tall og en autensiteringsnøkkel.

Data Encryption Standard (DES) blir brukt i autentiseringskodene. Dette er en chiffer som blir sett på som mindre sikker for mange applikasjoner, og det er mulig å utføre et brute-force angrep på nøklene. Med dette som bakgrunn vil det i denne oppgaven bli utført et angrep på authentiseringskodene.

Regnbuetabeller er brukt i denne oppgaven for å gjøre et angrep på flere nøkler mer effektivt. En regnbuetabell blir beskrevet som en metode som bruker litt mer tid på prosessering, men som til gjengjeld tar mindre plass i minnet. De blir som oftest brukt når hensikten er å avdekke nøker i et angrep der klarteksten er kjent. En enkel regnbuetablell er i denne oppgaven implementert i C++, og tiden det tar for å utføre et angrep er diskutert.

Preface

This Master's thesis is carried out in the 10th semester of my Master of Science degree in Communication Technology at Norwegian University of Science and Technology. I would like to thank my responsible professor Stig Frode Mjølsnes and my supervisor Tord Ingolf Reistad for their guidance and feedback during this semester.

Trondheim, June 2018

Mari Bergendahl

Contents

List of Tables

List of Figures

List of Acronyms

OBU	=	On Board Unit
EFC	=	Electronic Fee Collection
DSRC	=	Dedicated Short Range Communications
CEN	=	The European Center for Standardization
OSI	=	Open Systems Interconnection
BST	=	Beacon Service Table
VST	=	Vehicle Service Table
AC_CR	=	Access Credentials
RndOBU	=	Random Number from OBU
MAck	=	Master Access Credentials Key
MAuK	=	Master Authenticator Key
AC_CR-KeyReference	=	Access Credential Reference
AcK	=	Access Credentials Key
AuK	=	Authenticator Key
MAC	=	Message Authenticator Code
DES	=	Data Encryption Standard
NIST	=	National Institute of Standards and Technology
IBM	=	International Business Machines Corporation
AES	=	Advanced Encryption Standard
CUDA	=	Compute Unified DeviceArchitecture
LUT	=	Look Up Table
IV	=	Initial Vector

Chapter 1

Introduction

1.1 Motivation

Transport technology is undergoing a rapid change, and the need for technologies like intelligent transport systems (ITS) are increasing [Tay01]. Electronic Fee Collection (EFC) systems is a ITS technology that covers road tolling and ferry transport payments. AutoPASS is the Norwegian toll collection system and had in 2008 1,3 million users/tags, meaning that more than 50% of all vehicles in Norway had a tag [sin13].

The security in the AutoPASS is based on the EN 15509 EFC standard. The Norwegian system uses the security level 1 described in the standard, meaning that the OBU will require valid access credentials from the RSE before enabling access to the attributes in the OBU. In a draft paper [Reied] written by Tord Reistad, he described an attack on the Message Authentication Codes; *by requesting an empty attribute list, it is possible to avoid using the access credentials.* Without security level 1, an attack on the MACs is feasible, and stealing the key used in the calculation is then possible . An attacker will then have the possibility to use this key in the calculation of it's own MAC, hence pretending to have an other ID.

An attacker's goal is to obtain hundreds of different keys, using these keys one by one in it's own OBU. The original owner of the keys will then have to pay each time the attackers OBU passes through a toll station. By changing the stolen keys frequently it is almost impossible for an attacker to get caught.

Approximately 400 EFC transactions are assumed to be wrong each year [Reied]. If some of these transactions are wrong because of minor faults in the system, or because of dishonest people is difficult to know. But what we do know is that it is possible to crack DES, and therefore, it should be possible to obtain keys that can be used in order use someone else identity.

1.2 Scope and Objectives

This master thesis is based on a previous project where a customizable RSE were build in order to obtain MACs. Electronic fee collection systems uses DES in order to calculate these MACs used for authentication. DES is known to be a less secure encryption standard, because of the 56-bit key.

When a MAC is obtained by the attacker, there is possible to crack DES and obtain authentication keys. Rainbow Tables is a tool to perform such an attack, consisting of a precomputed table and an algorithm to later search these tables after an obtained MAC. The objectives of this thesis will be to:

1. Build a Rainbow Table, and make an attack on DES
2. Compare the computational times of a sequential and a theoretical parallel programmed attack

The goal of this thesis is to generate a Rainbow Tables and to use this to perform an attack on the encryption function DES. A second goal is to obtain keys in the shortest possible amount of time.

1.3 Methodology

The intention of this thesis is to make a Rainbow Table for GPUs. Generating the Rainbow Chains that constitutes a Rainbow Table is highly parallelizable. Implementing this properly on GPUs is very time consuming and not an easy task, but in return it will make the generation a lot faster that on CPUs. Because of lack of experience and knowledge in GPU programming a small scale implementation in C++ is done, and also investigating what could be done faster with parallel processing.

1.4 Related Work

This thesis will be a continuation of Sverre Turter Sandvold's Master's thesis from 2017 [ST17]. He aimed to make a customizable RSE. The RSE was intended to communicate with an OBU in order to obtain several MACs from the OBU. Sandvold also aimed to make an attack on these MACs by building a rainbow table, but he did not manage to finish the implementation on GPUs. He's work is still valuable for this Master's thesis.

1.5 Outline

This thesis is divided into 5 chapters. The outline is as following:

Chapter 2 presents an overview of the electronic fee collection, and the most important security information needed to do an attack on DES.

Chapter 3 describes the rainbow table and how it can be used in an attack.

Chapter 4 gives a brief introduction to parallel computing, and describes how a rainbow table can be built using GPUs.

Chapter 5 contains a summary and conclusion of work done in this masters thesis. It also suggests some improvements and ideas for further work on the topic.

Chapter 2

Electronic Fee Collection

Electronic Fee Collection (EFC) systems is an ITS technology that covers road tolling and ferry transport payments. In this chapter, backround informations for the fundamentals in an EFC system are provided. A short description of the DSRC are presented, before the EFC functions, AutoPASS, and their security implementations are discussed in more detail.

2.1 Overview

The main components in EFC systems in most of Europe, is the OBU and the RSE. These units uses the DSRC as communication protocol.

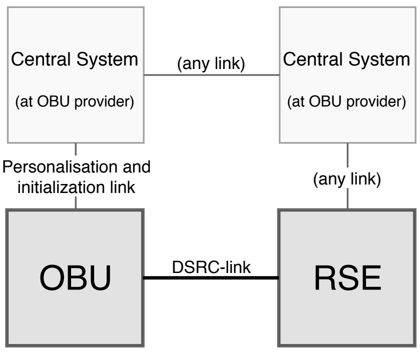

Figure 2.1: EFC Overview

2.2 Dedicated Short Range Communications

The units in an EFC system uses the DSRC at 5.8 GHz as communication protocol [CEN03a]. The European Center for Standardization (CEN) has produced a set of standards that defines a basic level of technical interoperability for these units using DSRC. The DSRC protocol has been defined as a OSI protocol stack consisting of three layers; the physical layer, the data link layer and the application layer, as seen in figure 2.2.

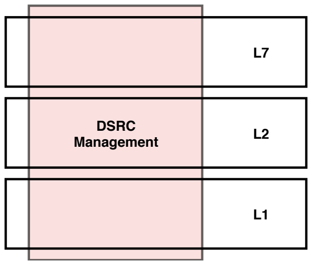

Figure 2.2: DSRC Protocol Architecture

The physical layer was designed to support different media types, and is at the moment using microwave transmission at 5.8GHz . Uplink data is sent at a bit rate of 250 kbit/s, and the bit rate for downlink is 500 kbit/s. These bit rates are the default values for bit transmission, but higher and lower transmission rates can be negotiated.

The data link layer is composed of two sublayer; the logical link control and the medium access control. The beacon sends out a “Beacon Service Table” (BST) with a list of present applications. The BST is sent downlink, followed by a “Public Window Allocation” and several “Public Windows”. One of these public windows are chosen by the vehicle for private uplink transmission. As an answer to the BST, the vehicle sends a “Vehicle Service Table” (VST) along with a list of all the applications that are available for both the vehicle and the beacon.

2.3 EFC Functions and Applications

The European standard EN 15509:2007 [CEN07] defines an Application Profile with the objective to support technical interoperability between EFC DSRC-based systems in Europe. This standard is based upon three base standards:

- EN ISO 14906 on EFC application interface definition for DSRC [CEN11]
- EN ISO 12834 on DSRC application layer [CEN03b]
- EN ISO 13382 on DSRC Profiles [CEN04]

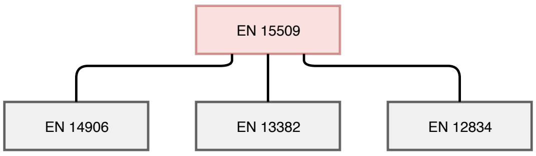

Figure 2.3: Relation between base standards and EN 15509

The EN 15509 specifies requirements for both the OBU, RSE and the DSRC. It states that the OBU shall comply with the underlying DSRC-standards for the three layers. Table 2.1 shows the different DSRC L7 services and EFC functions that the OBU shall support. Figure 2.4 shows the a complete EFC-transaction between the RSE and a vehicle that have entered the communication zone.

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

Table 2.1: DSRC L7 Services and EFC Functions Supported By the OBU

The RSE is at all times sending out BSTs to “wake up” OBUs within range of that RSE. When communication is established, the OBU returns a VST that contains information about an Access Credential Reference (AC_CR-KeyReference) and a random number (RndOBU). From these values a secret master key (MAck) and the secret key (Ack) are derived. The Ack is used for the computation of the AC_CR using the RndOBU as input. The value of the AC_CR are returned to the OBU that compares the value with its own calculation. If the values are equal, the RSE is approved by the OBU as an authorized unit, and is allowed to read data.

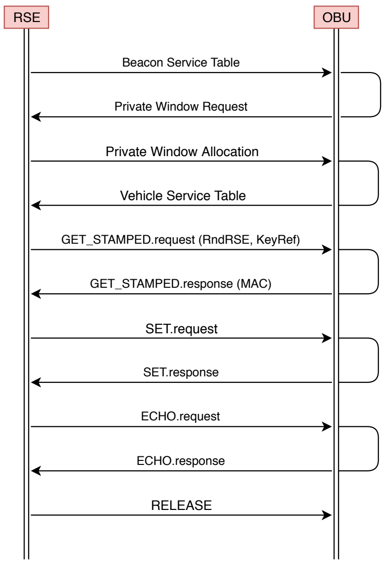

Figure 2.4: EFC-Transaction Between RSE and OBU

2.4 AutoPASS

AutoPASS is the Norwegian system for collection fees, used for both road tolling and ferry transport payments. AutoPASS is implemented with security level 1 as described in EN 15509:2007, and the AutoPASS OBE requires calculation of AC_CR from the RSE enabling access to the EFC attributes in the OBU.

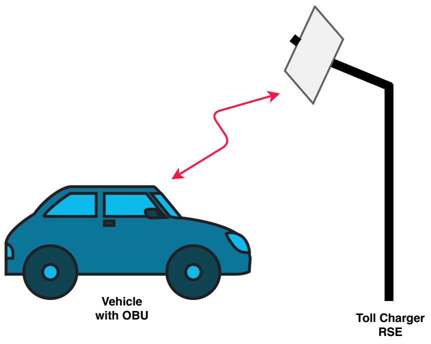

Figure 2.5: AutoPASS OBU and RSE

An OBU communicates with the Toll Charger RSE, and the RSE might communicate with the driver with a green (OK), or white signal (OK, but time to pay) when driving through a toll station. The toll station is responsible for a specific toll domain. The toll domain may be a road network or a specific section of a road (e.g. a bridge, a tunnel or a ferry connection). One OBU is connected to the registration number of one vehicle.

2.5 Security Calculations

EN 15590 standard allows two different security levels; 0 and 1. Security level 0 is mandatory and shall be implemented. The OBU shall support the GET_STAMPED function, meaning that it shall be able to calculate an Authenticator. The RSE shall also be able to calculate Authenticators to validate data integrity and origin of application data. There is no protection of user related data on the OBU with security level 0. In security level 1, level 0 shall be supported along with the support of calculation of Access Credentials. The AC_CR are used in the protection of user related data on the OBU.

2.5.1 Attribute Authenticator

Calculation of Authenticators are mandatory in both security levels. The authentication is done in the calculation and transaction of the GET_STAMPED messages; a random number (RndRSE) is included in the GET_STAMPED.request along with a Key Reference (KeyRef). This RndRSE is concatenated by the Attributelist in the GET_STAMPED.response. This concatenation is split into blocks of 8-octets (D_1 to D_{n-1}). An Authenticator Key (Auk) is derived from one of the eight Master Authenticator Keys (MAuK) stored in the OBU referenced by the KeyRef. A MAC is calculated using Data Encryption Standard in Chained Block Cipher mode with D_n and AuK as input, as seen in figure 2.6. This MAC is included in the GET_STAMPED.response. This is a 4-byte value, meaning that only the first half of the DES output is sent.

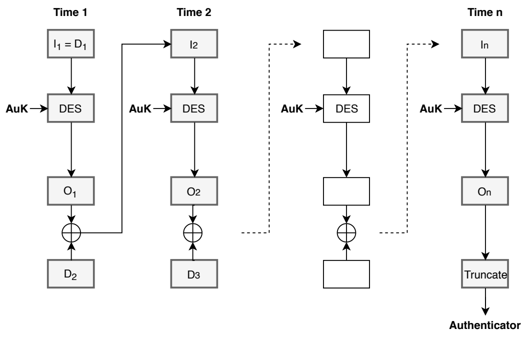

Figure 2.6: Calculation of an Authenticator Using DES in CBC Mode

Algorithm 1: Calculation of an Authenticator

```
(a) Let M = AttributeIDList || RndRSE || 00 00
(b) Let D1 = M1 = Sub(M,0,8), D2 = Sub(M,8,16), and Dn =
    Sub(M,8*(n-1),8*n)
(c) Compute O1 = DES[AuK](I1)
(d) Compute O2 = DES[AuK](I2 = O1 XOR D2)
(e) Compute On = DES[AuK](In = On-1 XOR Dn)
(f) Let MAC = Sub(On,0,4)
```

2.5.2 Access Credentials

The AC_CR are data transmitted to the OBE, in order to authenticate the RSE and is used to protect user related data on the OBU. It can carry passwords as well as cryptographic based information, and is only used when security level 1 is required. A reference to the AC_CR are transmitted in the VST along with a random number (RndOBU). This AC_CR-KeyReference contains a pointer to a secret MAcK that shall be used for the computation of the AcK. This AcK will then be used in the calculation of the AC_CR. The OBU receives the AC_CR computed by the RSA and compares it with its own calculation. If they are equal, reading data from the OBU is allowed. Figure 2.7 shows the process of the calculation of AC_CR.

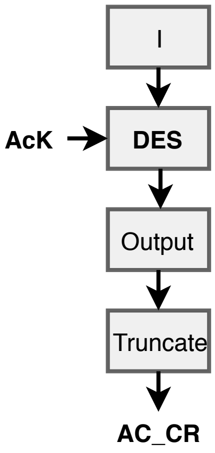

Figure 2.7: Calculation of an AC_CR

Algorithm 2: Calculation of the AC_CR

```
(a) Let I = RndOBU || 00 00 00 00
(b) Let O = DES[AcK](I)
(c) Let AC_CR = Sub(O,0,4)
```

2.5.3 Authentication Key

The Authenticator Key (AuK) is derived from one of the eight Master Authenticator Keys (MAuKs) stored in OBU. A Compact PAN is calculated by taking the leftmost half of the Personal Account Number (PAN) XORed with the rightmost half. The PAN is in this way truncated from 64 bits to 32 bits. Contract Provider is known at both parties, and is transmitted as a part of the VST. An example of how the AuK can be computed from the PAN is shown below:

Algorithm 3: Calculation of the Authentication Key

```
(a) Let Compact PAN = Sub(PAN,0,4) XOR Sub(PAN,4,4)
(b) Let VAL = Compact PAN || ContractProvider || 00
(c) Compute AuK(k) = 3DES[EAuK(k)](VAL)
```

2.5.4 Access Key

An AC_CR-KeyReference is sent to the RSE in the VST, and is used four times in the calculation of the AcK, in addition to work as a reference to the MAcK. Algorithm 4 shows how the AcK are calculated from the VAL and MAcK. Note that triple DES is used as the cryptographic algorithm.

Algorithm 4: Calculation of Access Credentials Key

```
(a) Let VAL = AC_CR-KeyRef || AC_CR-KeyRef || AC_CR-KeyRef
    || AC_CR-KeyRef
(b) Compute AcK = 3DES[MAcK](VAL)
```

2.6 Data Encryption Standard

Data Encryption Standard (DES) are used in EFC in the process of generating keys. It is a widely used standard, even though it can no longer be considered secure. This section will give an overview of the algorithm behind DES.

2.6.1 Brief History

In 1972, the National Institute of Standards and Technology (NIST) identified a need for standard that could be used for encrypting unclassified, sensitive information. The year after, several proposals for a cipher were submitted, but none of them turned out to be suitable. In 1974, IBM submitted a cipher that was based on an already existing algorithm – the Feistel network. The cipher was met with skepticism, especially regarding the shortened key length and the S-Boxes. Despite the criticism, the cipher submitted by IBM was approved as federal standard in 1976, and got the name *Data Encryption Standard* (DES).

Already in 1977 a machine was described that could crack DES in a single day. This was, on the other hand, an extremely expensive machine, but in the following years, less expensive machines were proposed. These suggested attacks remained theoretical, but in 1998 the first brute force attack were demonstrated. This showed that DES could easily be cracked, and a need for a replacement cipher were required. In 2002, DES was superseded by the Advanced Encryption Standard (AES), and it has been withdrawn as a standard by NIST. Despite this, DES is still remains in widespead use [Sta99].

2.6.2 DES Encryption Algorithm

DES is a symmetric block cipher with 64-bit blocks and a 56-bit key. Each block of plaintext is enciphered through rounds of permutation and substitution, to a 64-bit cipher text. The encryption algorithm consists of 16 rounds as shown in figure 2.8.

As a preparation before the encryption, the plaintext is passed through the Initial Permutation where the output is divided in two; one left side, L_0, and one right side, R_0. This marks the beginning of round 1. As in every round the right half, R_n, is passed into the Feistel (F) function together with the round key, K_n. The F-function is followed by an XOR operation between the output of the F-function and the left half, L_n. The result of the XOR makes up the next round's R_{n+1}. Note that L_{n+1} equals R_n. In the last round, the left and right half is passed into the Inverse Initial Permutation without being swapped.

The F-function plays a central role in each round of th 16 rounds of encryption. The function operated only on 32-bits of the input at a time, and it consists of four stages, as seen in 2.9:

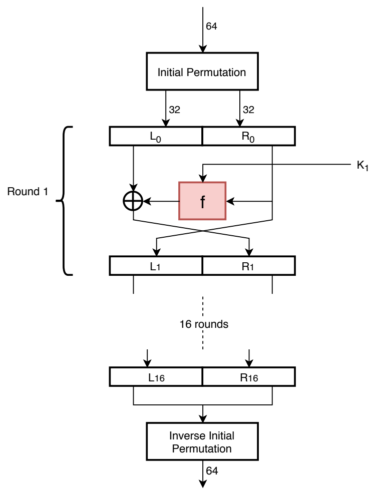

Figure 2.8: DES Algorithm

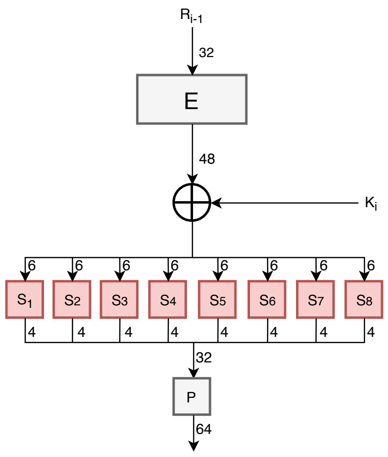

Figure 2.9: F-Function

The first stage is the expansion permutation (E). The 32-bit input is transformed into a 48-bit output, where some of the input bits are duplicated. The second stage is to combine this output with the round key, K_i. 16 round keys are generated from the input key – one for each round. After XORing in the round key, the block is divided into 8 6-bit sub-blocks. Each of these are used as input to eighth different substitution boxes, or S-boxes. The first S-box is shown in table 2.2. These boxes works as lookup tables, transforming the 8-bit input into a 4-bit output. The last stage of the F-function is to permute the 32-bit output from the S-boxes into 64-bit final output.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Table 2.2: S-box Number 1

Chapter 3

Rainbow Tables

Finding the original word of a hash is a difficult task. A hash is a one-way encryption function, which means that it is not possible to invert. One way to find the original data from the ciphertext is to perform a brute force search; to try all the possible plaintexts until the right one is found. This method is very time consuming. An other way to determine the unencrypted data is to create a lookup table in advance. A look up table is a potentially huge table containing all possible plaintexts and their encrypted results, as seen in table 3.1.

Plaintext	Ciphertext
123	37 6D 02 15 79 41 d7 25
456	b1 86 59 48 f1 dd b5 cb
789	63 fe 90 6e 9d e2 96 3d

Table 3.1: Small Lookup Table

3.1 Building a Rainbow Table

A simple lookup table can be huge, but in return, they are fast. If you know the encrypted value, you simply look it up to see what the plaitext/password is. Rainbow Tables are a time-memory trade-off, meaning that they consumes less memory, but uses a bit more processing time. Rainbow Tables makes use of chains and reduction functions to make the table smaller. It is smaller because only the first and last column is stored in the table. The values in between are calculated if needed.

Rainbow Tables was described in an article by Philippe Oechslin in 2003 [Oec03]. Oechslin based his Rainbow Tables on an already existing technique described by Martin Hellman, typically used to recover keys when the plaintext and ciphertext is known. For a system having N keys Hellman's method can recover a key in N2/3 operations using N2/3 words of memory. Given a fixed plaintext, the method aims to precalculate all possible ciphertexts in advance with all N possible keys. The ciphertext are organized in chains, where only the first and last element of every chain is stored in memory.

$$k_{\mathrm{i}} \xrightarrow{\mathrm{S_{k_i}\ (P_0)}} C_{\mathrm{i}} \xrightarrow{\mathrm{Ri\ (C_i)}} k_{\mathrm{i+1}}$$

The chains are created as shown above. A cipher S is applied to the plaintext, $\mathrm{P_0}$, and the first key, $\mathrm{K_i}$. A Reduction function, R, then creates a new key, $\mathrm{K_{i+1}}$ from the ciphertext. The ciphertext is longer than the key, hence the reduction. By applying the cipher S and the reduction function R over and over again, a chain of keys are made. As stated earlier, only the first and last element are stored in memory. Hellman created several tables, but with different reduction function for each table.

Hellman's original method has one limitations; if two chains collide within a table, they will merge. Oechslin proposed a new method that reduces the chance of merging. His method uses t-1 reduction functions; one for each point in the chain. In this case, a merge will only occur if the collision takes place at the same point in both chains. Oechslin called these chains Rainbow Chains. The probability of success (avoid merges) in these Rainbow Tables is:

$$P_{\text{table}} = 1 - \prod_{i=1}^{t}(1 - \frac{b_{\mathrm{i}}}{N}), \text{ where } b_{\mathrm{i}} = 1 \text{ and } b_{\mathrm{i+1}} = N(1 - e^{-\frac{m_{\mathrm{n}}}{N}}) \tag{3.1}$$

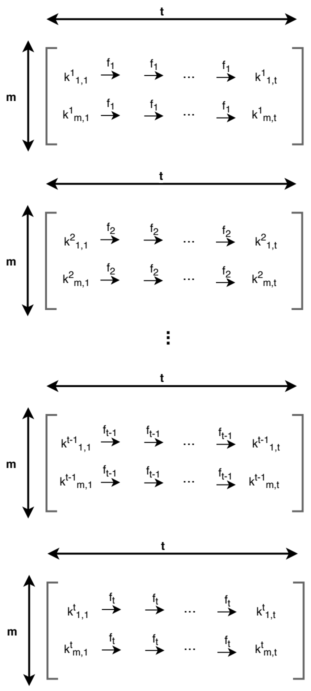
t
m
k¹1,1 f1 f1 ... f1 k¹1,t
k¹m,1 f1 f1 ... f1 k¹m,t
k²1,1 f2 f2 ... f2 k²1,t
k²m,1 f2 f2 ... f2 k²m,t
k^t-1 1,1 ft-1 ft-1 ... ft-1 k^t-1 1,t
k^t-1 m,1 ft-1 ft-1 ... ft-1 k^t-1 m,t
k^t 1,1 ft ft ... ft k^t 1,t
k^t m,1 ft ft ... ft k^t m,t

Figure 3.1: Hellman's Original Tables

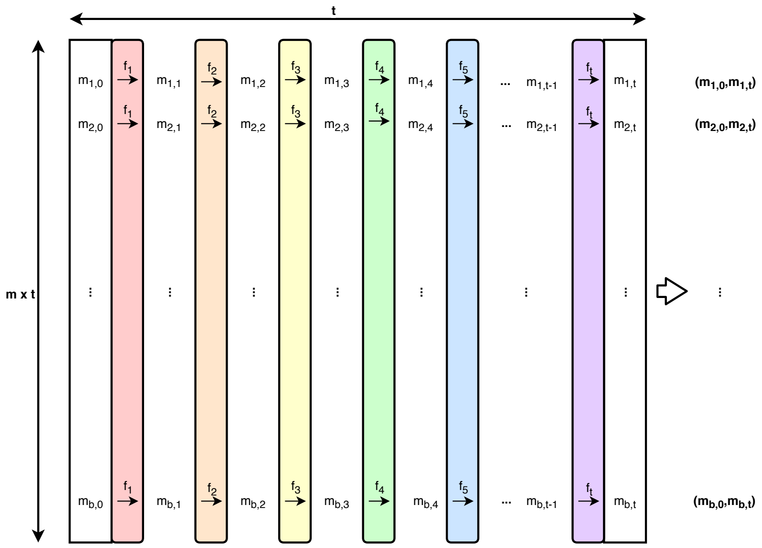
t
m x t
m1,0 f1 m1,1 f2 m1,2 f3 m1,3 f4 m1,4 f5 ... m1,t-1 ft m1,t
m2,0 f1 m2,1 f2 m2,2 f3 m2,3 f4 m2,4 f5 ... m2,t-1 ft m2,t
mb,0 f1 mb,1 f2 mb,2 f3 mb,3 f4 mb,4 f5 ... mb,t-1 ft mb,t
(m1,0,m1,t)
(m2,0,m2,t)
(mb,0,mb,t)

Figure 3.2: Rainbow Chains

3.2 Searching the Rainbow Table

When both the plaintext and the ciphertext are known, the Rainbow Table can be used to find the key. Applying the last reduction function, R_{t-1}, to the ciphertext is giving a result we want to find among the end points in the table. If the right end point is found, we rebuild the chain using the corresponding start point. An example is the ciphertext 5520 and the reduction function, h(). Applying the reduction function, R_{n-1} to our ciphertext will give the result value 55.

Start Point	End Point
3	08
10	50
68	55

Table 3.2: Small Rainbow Table

As seen in Table 3.2 the corresponding start point to the end point, 55 is 68. To find the wanted key, the complete chain needs to be calculated:

k_i	C_i	k_{i+1}	C_{i+1}	k_{i+2}	C_{i+2}	k_{i+3}
68	3131	31	3790	$\bigcirc\!\!\!\!\!\!37$	5520	55

Table 3.3: Complete Chain

First, the start key, 68, is hashed, resulting in 3131. This ciphertext is reduced to a new valid key, 31. This process is continued until the original ciphertext, 5520, is reached. We then know that the preceding key, 37 is the used to encipher the plaintext.

If we do not find the value among the end points, the value is hashed and the reduction function from the last column is applied, R_{t-2}. A new search among the end values is performed. If the value is still not found, the two last reduction functions is applied together with the hash function. This is done over and over again until the value is found.

3.3 Cracking DES with Rainbow Tables

In this thesis Rainbow Tables will be used in attaching the encryption function DES. This can be done when both the plaintext and the ciphertext is known. A fixed DES-key is used as a start value, and the known ciphertext is the end value.

3.3.1 Generating a Rainbow Table in C++

As a part of this thesis, a Rainbow Table is implemented in C++ using CryptoPP as a library for the cryptographic functions, including encryption and decryption tools for sequential DES. The total amount of possible DES keys is 2^{56}, meaning that we need long chains and lot of storage to be able to do an attack. The initial parameters for implementing a Rainbow Table is listed in table 3.4. With a hard drive of size 3TB, we can store approximately 9.42 x 10^{10}. This means that the length of one chain can have the length of 764 587.

Making a perfekt table, without collision, and a 100% chance of finding a MAC, will make the Rainbow Table larger than needed. When attacking MACs in AutoPASS, the goal is not to break every single one of them, but obtaining a significant percentage of them. If one MAC gives no result in the Rainbow Table, we move on to the next MAC.

Total amount of keys	2^{56}
Size of one chain	35 Byte
Size of hard drive	3TB
Total amount of chains	9.42 x 10^{10}
Length of one chain	2^{56} / 9.42 x 10^{10} = 764 587

Table 3.4: Initial Parameters

Initial Key Generation

The first step when generating a Rainbow Table is to build a starting key. This is the first key in every chain in the table, and is dependent of the the chain number. The chain number is converted to binary, and padded with zeroes to make the key 56 bit. The next step is to add 8 error detecting bits. These bits are not used in the encryption. These error detecting bits are added as 1's after every 7th bit, becoming the least significant bit of each byte. Finally, the key are converted back to hexadecimal and padded with zeroes.

```
std::string key_generation(int chainnumber){

      int rawkey = convertDecimalToBinary(chainnumber);
      std::string rawkey_padded = padTo(std::to_string(rawkey),
          56);

      std::string tempkey = "";
      for (int i = 0; i < 56; i=i+7){
            tempkey += rawkey_padded.substr(i, 7);
            tempkey += '1';
      }

      //tempkey to hex
      std::string hex_key = "10101010"+binary_to_hex(tempkey);
```

```
        //add padding
        std::string padded_key = padTo(hex_key, 16);

        return padded_key;
}
```

Encryption Function

When building the encryption function, there is some an initial plaintext and an initial vector (IV) needs to be set. The plaintext is set to be "0004000000000000" and the IV is all zeroes. The library, CryptoPP, is used in encryption function, providing the needed encryption tool used in DES.

```
std::string crypt(std::string inputkey){
        std::cout << "Krypterer med key:\t\t " << inputkey <<
             std::endl;

        long long hex_key = std::stoul(inputkey, 0, 16);

        byte key[ CryptoPP::DES::DEFAULT_KEYLENGTH ], iv[
             CryptoPP::DES::BLOCKSIZE ];
        memset( key, hex_key, CryptoPP::DES::DEFAULT_KEYLENGTH );
        memset( iv, 0x00, CryptoPP::DES::BLOCKSIZE );

        //
        // String and Sink setup
        //

        std::string plaintext = "0004000000000000";
        std::string ciphertext;

        //
        // Create Cipher Text
        //
        CryptoPP::DES::Encryption desEncryption(key,
             CryptoPP::DES::DEFAULT_KEYLENGTH);
        CryptoPP::CBC_Mode_ExternalCipher::Encryption
             cbcEncryption( desEncryption, iv );

        CryptoPP::StreamTransformationFilter
             stfEncryptor(cbcEncryption, new CryptoPP::StringSink(
             ciphertext ) );


        stfEncryptor.Put( reinterpret_cast<const unsigned char*>(
            hex_to_string(plaintext).c_str() ), plaintext.length()
            + 1 );
        stfEncryptor.MessageEnd();

        std::string output1;
        for( int i = 0; i < 8; i++ ) {
                if(int_to_hex((0xFF &
                    static_cast<byte>(ciphertext[i]))).length() == 1){
                        output1 = output1 + "0" +int_to_hex((0xFF &
                            static_cast<byte>(ciphertext[i])));
                }
                else{
                        output1 = output1 + int_to_hex((0xFF &
                            static_cast<byte>(ciphertext[i])));
                }
        }

        //
        // Dump Cipher Text (output)
        //
        std::cout << "From Crypt: " << output1 << std::endl;
        return output1;
}
```

Reduction Function

The purpose of the reduction function is to transform the cipher text to a valid key. The cipher text is often longer that the key, hence the need of reduction. The attacker chooses the reduction function freely, and ideally it should produce unique keys for every cipher text. The goal is to create a Rainbow Table without merges. These merges can easily be detected by their identical endpoints. To reduce the chance of merges Oehsling suggests using chains within a certain range of length [Oec03]. He also implemented his Rainbow Tables with different reduction functions on each point in the chains. This means that a merge can only happen when a collision occurs at the same point in the chain.

In DES both the plain text, cipher text and the key are of length 64 bits, meaning that the output of the encryption can almost be used directly as a key. The code below shoes how the reduction function is implemented for this Rainbow Table. This function takes the MAC and the column number as input. The column number implies the column in the chain where the reduction occurs. The column number is converted to hexadecimal and merged with the first bits of the MAC to form a new key of size 64 bits.

```
std::string reduce(std::string hash, int column){

      //Converting hash and column to hexadecimal
      std::string hash_hex = string_to_hex(hash);
      std::string column_hex = int_to_hex(column);

      //The length of the column in hex
      int column_len = column_hex.size();

      //Remove N last characters of the hash (N = column_len)
      std::string sub = hash.substr(0, hash.size()-column_len);

      //Merge hash and column to a new key
      std::string reduced_hash = sub+column_hex;

      return reduced_hash;
}
```

Generating Rainbow Chains

The Rainbow Chains are the building blocks in a Rainbow Table. The length of these chains are chosen by the attacker. The probability for merges increases with the length of the chain, and the key is to choose the chain length within a certain range. When this is decided, the chain length and the chain number is taken as input into the chain_maker() function, as shown in the code below. The starting key is an encrypted version of the chain number, and is different for every chain. The next step is to use the starting key as input to create the next key in the chain. This procedure is explained in section 3.1. The last hash that is generated are stored together with the start key as a pair in an array.

```
std::pair<std::string, std::string> chain_maker(int chainNumber,
   int chainlength){

      std::string startkey = key_generation(chainNumber);
      std::string endhash = startkey;

      for (int i = 0; i < chainlength; i=i+1){
            if(i<chainlength-1){
                  endhash = reduce(crypt(endhash), i);
            }
            else{
                  endhash = crypt(endhash);
            }
      }
      return make_pair(startkey, endhash);
}
```

To store the start key and the last hash, and an array of pair objects are used. The Rainbow Table are generated as shown in the code below:

```
void make_chains(int chains, int chainlength){

      //Creating an array of rainbow pairs
      std::pair<std::string, std::string> rt[chains];

      //Creating pairs of {starkey, endkey} and storing them in
          rt[]
      for (int i = 0; i < chains; i=i+1){
            rt[i] = chain_maker(i, chainlength);
      }
      //Printing the Rainbow Table
      for (int i = 0; i < chains; i=i+1){
            std::cout << "{" << rt[i].first << "," <<
                rt[i].second <<
            "}" << std::endl;
      }
}
```

3.3.2 Searching for a MAC

A huge part of attacking DES is to generating the Rainbow Table. This is usually done before the actual attack begins, and saves a lot of time when performing the "look up"-phase. The function search(), shown below, performs a search among the end hashes in the table, searching to find an end hash that matches the input MAC. If there is no match among the end hashes, the reduction function for the last column is applied to the MAC before it is encrypted. The search among the end hashes are performed again with the reduced and encrypted MAC. This is done over and over again until either the hash is found or not found.

```
void search(std::string mac, std::pair<std::string, std::string>
    table[],
   int chains, int chainlength){
      std::cout << "\nLooking for MAC: " << mac << std::endl;
      std::string startkey;
      int found = 0;
      for (int i = 0; i < chains; i=i+1){
         if(table[i].second == mac){

            std::cout << "\n" << mac << " found in chain number "
            << i << std::endl;

            std::cout << "The corresponding startkey is "
            << table[i].first << std::endl;
```

```
                startkey = table[i].first;
                found = 1;
                recreate_chain(startkey, mac, chainlength);
                break;

                }
            if(i == chains-1 && !found){
                hash = crypt(reduce(hash, chainlength-i-1));
            }
        }
}
```

When the MAC is found, the corresponding start key is used to recreate the chain. The recreation is stopped when the MAC is found, and key that was used to encrypt the MAC is obtained.

```
std::string recreate_chain(std::string startkey, std::string mac,
    int chainlength){

        std::cout << "Searching for key to MAC " << mac << "\n"
        << std::endl;

        std::string hash = crypt(startkey);
        std::string key = startkey;
        std::string wanted;
        int column;


        for (int i = 0; i < chainlength; i=i+1){
                column = i;
                if(hash == mac){
                        wanted = key;
                        break;
                }
                else{
                        key = reduce(hash, i);
                        hash = crypt(reduce(hash, i));
                }
        }

        std::cout << "Wanted key: " << wanted << " found in column "
        << column << std::endl;

return wanted;
}
```

When creating a small Rainbow Table with 10 chains of length 3, the output is as below. As seen, the table is not without merges.

```
{ 0101010101010101, 3ef1a1b05740648f }
{ 0101010101010103, 97aae0e8769fe83a }
{ 0101010101010105, 0db095dae115298d }
{ 0101010101010107, efc1a075b4d03af6 }
{ 0101010101010109, 97aae0e8769fe83a }
{ 010101010101010b, 3331ffa6d4da129d }
{ 010101010101010d, c13518c3e3d43035 }
{ 010101010101010f, 3ef1a1b05740648f }
{ 0101010101010111, 96b1e7b3b9344fde }
{ 0101010101010113, 97aae0e8769fe83a }
Generated 10 chains of length 3 in: 0.003542sec

Looking for MAC: 0db095dae115298d

0db095dae115298d found in chain number 2
The corresponding startkey is 0101010101010105
Searching for key to MAC 0db095dae115298d:

Key:     0101010101010105
Hash:    c18a86debe45b9e4
Key:     c18a86debe45b9e0
Hash:    511c4aa9d0ff2610
Key:     511c4aa9d0ff2611
Hash:    0db095dae115298d

Wanted key: 511c4aa9d0ff2611
```

3.3.3 Improvements to the Attack

Currently, the CPU is regarded as the computing component in today's traditional computer systems [LCG+10], meaning that an attack that runs on CPU is suitable for most attackers. On the other hand, a code that runs sequentially on a CPU, will use a lot of time on both building a Rainbow Table, and on searching for the right MAC. An attack programmed to run on Graphical Processing Units (GPUs) will be faster, an is discussed in chapter 4.

If the goal is to perform an attack using a CPU, there is still some improvements that can be done to the program presented earlier in this chapter. The first suggestion is the use of threads, to speed up the generation of chains. This enables the possibility to do work in parallel. The computer used in this thesis has an Intel Xeon E5-2620V4 processor with the specifications shown in table 3.5 below. As seen in the table, this server contains two sockets each, with the total of 32 threads that can run run parallel. Computing 32 Rainbow

Chains in parallel means that the generation of a Rainbow Table can be done faster, but is still using years in generating a full scale Rainbow Table [ST17].

Architecture:	x86_64
CPU(s):	32
Thread(s) per core:	2
Core(s) per socket:	8
Socket(s):	2

Table 3.5: Intel Xeon E5-2620V4

To speed up the regeneration of chains, checkpoints can be used as some intermediate point to detect false alarms, before regenerating the whole chain from start to end. Avoine et al. [AJO08] introduced a new technique in 2008 based on checkpoints that would reduce the processing times by ruling out false alarms. The checkpoints are usually a position in the chain where a parity check is performed.

To avoid searching through all of the chains when performing the attack, the list of startkeys and end hashes should be sorted on the end hashes. In this way the search can be stopped when the position where the MAC should have been, is passed.

Chapter 4

Parallel Programming

This chapter gives an introduction to parallel programming, and how GPUs can speed up the generation of rainbow tables. It also shows how the program CUDA can be used to cotroll the GPUs.

4.1 Graphical Processing Unit

CPUs consists of a number of cores that are optimized for sequential serial programming, while a GPU consists of thousands of more efficient cores designed for handling multiple tasks simultaneously. GPUs differs from CPUs in the way they parallelize instructions. When running instructions in parallel on a CPU, pipeline technology is used, and the CPU have to guess on what data it can perform several instructions simultaneously [Mey11]. GPUs uses Single Instruction Multiple Data (SIMD), meaning that it parallelizes data on the same instruction.

CUDA is a parallel computing platform developed by NVIDIA for programming on GPUs. A CUDA program consists of one *host* part and one *kernel* part. The host code is run on the CPU and uses the CPU memory, while the kernel code runs on the GPU and uses the GPU memory. The host is responsible for running the program, and are launching the kernel code on the GPU when needed.

NVIDIA's CUDA is the most popular framework among several programming frameworks that are build to harness GPU cores. CUDA stands for Compute Unified Device Architecture, and is a toolkit that accesses an extention of C++ combined with its own programming model. This combination lets programmers run their code on the GPUs. Five classic subtasks of CUDA code are:

1. Create the kernel
2. Set up device memory
3. Execute the kernel on the device
4. Transfer the result from the device to host
5. Free the device memory

The first task is to create the kernel that performs the calculations. The kernel program is the heart of a CUDA program, and is executed by GPUs in parallel. It is executed by an array of CUDA threads. All threads run the same code, and each thread has an ID that it uses to compute memory addresses and make control decisions. CUDA organizes these threads in a grid hierarchi of thread blocks. A grid is a set of thread blocks that can be processed on the device in parallel. Each thread is a set of concurrent threads that can cooperate amongs themselves and access a shared memory space private to the block. Its the programmers job to specify the grid blocks organization on each kernel call, since it can be different each time witin the limit set by their specific GPU.

The second subtask is to allocate memory on the GPU, and also for the CPU. CUDA uses the concept of unified memory to make this easy. It provides a single memory space accessible by all GPUs and CPUs on your system. This will allocate data in unified memory, and returns a pointer that we can access from CPU or GPU code. The host code, run on the CPU, allocates both CPU and GPU memory through malloc and cudaMalloc in the main function:

```
cudaMallocManaged(&x, N*sizeof(float));
```

The next step is to launch the kernel with a thread of grid blocks. CUDA GPUs runs kernels using blocks of threads that are a multiple of 32 in size, for instance 256. But doing this alone will causing it to do the computation once per thread, rather than spreading the computation among parallel threads. To do it properly we get the indexes of the running threads using the threadID keyword for the current thread in its block. We use the blockDim keyword to find the number of threads in the block.

```
int index = threadIdx.x;
int stride = blockDim.x;
for(int i = index; i<n; i+= stride){
   /*Do something*/
}

myKernel <<<numBlocks,threadsPerBlock>>>(/*params for the kernel
    func*/);
```

After running the kernel, the results needs to transfered from the device to host. We do this by copying the function cudaMemcpy(). When the results are copied back to the host, the last step that needs to be done is to free the device memory.

```
cudaFree(x);
```

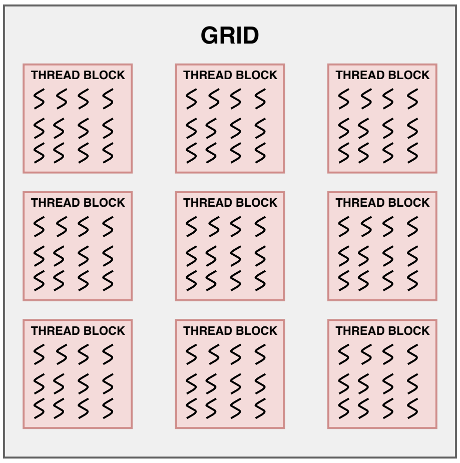

Figure 4.1: CUDA Grid

For the experiments in this thesis NTNU's newly built computer with 4 NVDIA GTX 1080 GPUs will be examined. One such GPU contains 20 streaming multiprocessors (SMs), and has 2560 CUDA cores. Each GPU core can run 16 threads simultaneously, meaning that 16*2560 threads can be run in parallel. These threads are organized in blocks, and each block is executed by a multiprocessing unit. At the highest level each kernel creates a single grid, as seen in figure 4.1, consisting of several thread blocks. Threads in the same block can share data, but are otherwise independent. Synchronization between thread blocks can only happen when the kernel is terminated.

GPU	GeForce GTX 1080
SMs	20
CUDA cores	2560
Texture Units	160

Table 4.1: GPU Specifications

A challenge when programming on GPU is the balance between computational intensity and memory bandwidth [BCK11]. The host and the device have their own memory address space. When a program runs on the GPU, the data is first allocated on host memory and then copied to the device memory for the threads running on the GPU to access. The data are copied back to host memory when the execution is finished. The threads has an hierarchical ordering, and both the computations and the data transfer between CPU and GPU must be done by the same hierarchical threads.

As stated earlier, threads are scheduled as a collection of 32 threads. This is also called a warp. A callenge in CUDA is to avoide so called "warp serialization". When waprs are scheduled, they are scheduled together with other warps in a multiprocessor. Each processor has processors to execute threads and an execution controller [LWT12]. All the thread processors in one warp executes the same instruction at the same time. To achieve maximum efficiency it is important that all of the threads in one warp follows the same execution flow. If one flow suddenly takes different directions, for instance in an if-else-statement, if can potentially increase the execution time. When this happens, new thread groups are made with the same execution flow. These thread groups are executed sequentially by the multiprocessor.

4.2 Building a Rainbow Table using GPUs

Building a Rainbow Table includes sub tasks that can be done in parallel. Because of lack of experience and knowledge in working with GPUs and CUDA, a theoretical analysis of how this can be done in parallel, is done.

The chains that constitutes a Rainbow Table is calculated independently of the other chains, meaning that each chain can be generated in parallel. A make_chain() procedure will be performed on each thread, *i*, and it will make a chain of length, *t-1*.

As shown in the CUDA code below, the first step is to allocate memory on both the CPU and on the GPU. Variables denoted with device_, refers to the variables defined in the global memory of the GPU.

```
int main() {

   unsigned int* device_plaintext;

   cudaMalloc((void**)&device_plaintext, sizeof(unsigned
       int)*(mem_length));

   cudaMemcpy(device_plaintext, plaintext, sizeof(unsigned int)
   *mem_length, cudaMemcpyHostToDevice);

   // Create int array on the CPU.
   int host_rainbowtable[N];

   // Create a corresponding int array on the GPU.
   int *device_rainbowtable;
   cudaMalloc((void **)&device_rainbowtable, N*sizeof(int));

   // Launch GPU code with N threads, one per array element.
   chain_maker<<<N, 1>>>(device_rainbowtable, device_plaintext);

   // Copy output array from GPU back to CPU.
   cudaMemcpy(host_rainbowtable, device_rainbowtable,
       N*sizeof(int), cudaMemcpyDeviceToHost);

   for (int i = 0; i<N; ++i) {
      printf("%d\n", host_rainbowtable[i]);
   }

   // Free up the arrays on the GPU.
   cudaFree(device_rainbowtable);

return 0;
```

4.3 Parallelized DES

If we break down the DES algorithm, we have a number of sub-processes in each of the 16 rounds: an XOR, expansion and permutation operations, and the S-boxes. The latter process is the most time consuming operation in the algorithm. It is normally handled like a look up table, where 8 bits, one from each input block, are combined to a 4 bit output. In 1997, Eli Biham suggested a faster implementation of S-boxes using logical gate circuits [Bih97]. This implementation was called *bitsliced DES*, and includes AND, OR, XOR gates, run as machine instructions. The main idea is that encryption is done on one bit at a time. When parallel processing is supported, Biham's implementation results in 64 DES encryptions run in parallel [Kwa00].

When using bitslicing each bit is treated as its own variable, meaning that each S-box takes 6 boolean variables as input. A digital circuit is created using logical gates. The time it takes to simulate the operation of this circuit, using CPU instructions, is roughly proportional to the number of gates. Without any optimization the 16 functions of two variables can be generated with 12 gates. For the selection, 4x2x(8+4+2+1) gates are needed. This gives a total of 132 gates. Since the bottleneck in bitsliced DES is the number of gates required to implement the logical circuit, a number of scientists have tried to reduce the number of gates. By optimizing the implementation, Biham was able to reduce the number of gates used in the S-boxes down to an average 100. Matthew Kwan showed in his paper [Kwa00] that he was able to reduce the average number of gates to 51, by also including non-standard gates:

S-Box	S1	S2	S3	S4	S5	S6	S7	S8
Gates	56	50	53	39	56	53	51	50

Table 4.2: Matthew Kwan's Optimization to Bitsliced DES

```
static byte sbox[8][64] = {
   /* S1 */
   14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7,
   0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
   4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
   15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13,

   /* S2 */
   15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10,
   3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
   0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
   13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9,

   /* S3 */
   10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8,
   13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
   13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
   1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12,

   /* S4 */
   7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15,
   13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
   10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
   3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14,

  /* S5 */
  2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,
  14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
  4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
  11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3,

  /* S6 */
  12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11,
  10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
  9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
  4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13,

  /* S7 */
  4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1,
  13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
  1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
  6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12,

   /* S8 */
  13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7,
  1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
  7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,
  2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11
};
```

Biside the S-boxes, showed as C++ code in the previous page, it is seven for-loops in the key generation part of the DES-algorith that is suitable for running in parallel. Detecting the parallelizable loops can be seen on as the first important step in the parallelization of the DES algoritm. One of these loops is found in the first bit permutation (PC-1):

```
const byte pc1[] = {
      57, 49, 41, 33, 25, 17, 9,
       1, 58, 50, 42, 34, 26, 18,
      10, 2, 59, 51, 43, 35, 27,
      19, 11, 3, 60, 52, 44, 36,
      63, 55, 47, 39, 31, 23, 15,
       7, 62, 54, 46, 38, 30, 22,
      14, 6, 61, 53, 45, 37, 29,
      21, 13, 5, 28, 20, 12, 4
};
```

PC-1 is used to remove parity bits in the 64-bit input key. It permutes the bits, and outputs a 56-bit key. There is no dependencies in this for-loop, and it can be done in parallel. The serialized version of the loop is shown below:

```
for (j=0; j<56; j++) {      /* convert pc1 to bits of key */
   l=pc1[j]-1;             /* integer bit location */
   m = l & 07;             /* find bit        */
   pc1m[j]=(key[l>>3] &    /* find which key byte l is in */
       bytebit[m])         /* and which bit of that byte */
       ? 1 : 0;            /* and store 1-bit result */
```

This loop can be written in parallel by applying the substitution technique in order to eliminate dependencies [BB05]:

```
for ( j = 0; j < 56; j++) {
    pc1m[j] = (key[pc1[j] >> 3] & bytebit[pc1[j] & 07]) ? 1 : 0;
}
```

In addition to the for-loops included in the key generation function, the for-loops in the encryption and decryption function can also be implemented as parallel processes. Every encryption or decryption are only dependent of a predefined key, and can be run as an individual task.

4.4 Serial vs. Parallel

CPU can only operate with a small number of cores to stay within the power and thermal limitations, meaning that the degree of parallelism have to stay within boundaries. GPUs on the other hand are made for running small processing elements in parallel [LKC+10].

Giovanni Agosta et al. [ABDSP10] showed in 2010 that it is possible to recover DES keys in as little as 18 days with the use of bitslicing on GPUs. The throughput in a standard DES implementation on GPUs is not dependent on the number of blocks per kernel. This is because is uses only a small part of the GPU memory. When using a plain DES implementation on a single CPU the number of keys that is tested every second is $13.32 * 10^6$. When GPUs and parallel processes is used, 6 times as many keys can be tested every second. In this speed, a key can be recovered in 58 days, as seen in table 4.3.

$$\frac{2^{64} \text{ possible keys}}{13.32 * 10^6 \text{ keys per second}} = 160 \text{ days} \tag{4.1}$$

	DES Breaking [days]
CPU	160
CPU with bitslicind DES	90
GPU	58
GPU with bitsliced DES	18

Table 4.3: Breaking Time Achieved

Chapter 5

Conclusion

This Master's thesis has looked at the possibilities of conducting an attack on the encryption function, DES, and obtain keys in a short amount of time. The goal was to investigate both CPU and GPU approaches to the attack.

The first objective was to build a Rainbow Table that could be used in attacking DES. Chapter 3 describes why a Rainbow Table is valuable when attacking a lot of MACs. A simple Rainbow Table was implemented in C++, and improvements to the attack were suggested. Due to lack of experience in CUDA and parallel programming, a theoretical solution to how the Rainbow Table can be implemented using GPUs were discussed.

The second objective was to make a comparison between the computational times of a sequential and a theoretical parallel programmed attack. Eli Biham's implementation with bitsliced DES were found as the best solution when speed is an important factor in the attack. An optimization of the original bitliced DES, states that it is possible to reduce the average number of gates used in the S-boxes down to 51. In chapter 4, it was shown that an attack using a bitsliced DES implementations on GPUs, compared to a plain DES implementation on one CPU, can be done 9 times faster.

This thesis has shown that it is possible to break DES using Rainbow Tables. This is possible when the attacker manages to avoid security level 1, and the AC_CR is not used. If this is possible to do without being detected, an attack on how to obtain the keys from the stolen MACs are described in section 3.3.

5.1 Further Work

When working further with this project, it will be natural to implement the Rainbow Table in full scale on a CPU, working with the suggested improvements discussed in chapter 3.3.3. By understanding the occurrence of merges and false alarms, and implementing checkpoints, I believe that generating Rainbow Tables can be done a lot faster.

The next natural step will be to implement the same project on GPUs. As discussed in section 4.3, one way to speed up the DES implementation is to use bitslicing. An interesting goal in the future will be to perform an attack on DES with less than an average of 51 gates used in the S-boxes. To achieve this, the NVIDIA LOP3.LUT instruction can be used. This takes a 3 input look up table (LUT), that can produce output of any 3 input true table. LOP3.LUT will be interesting to spend more time studying if moving forward with bitslicing.

Finally, it would be interesting to spend more time studying the EFC system, especially since it has been proved that DES is less secure, and that it is possible to obtain message authentication codes. More weaknesses in the system might be present, and would be fun to investigate.

Bibliography

[ABDSP10] Giovanni Agosta, Alessandro Barenghi, Fabrizio De Santis, and Gerardo Pelosi. Record setting software implementation of des using cuda. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 748–755. IEEE, 2010.

[AJO08] Gildas Avoine, Pascal Junod, and Philippe Oechslin. Characterization and improvement of time-memory trade-off based on perfect tables. *ACM Transactions on Information and System Security (TISSEC)*, 11(4):17, 2008.

[BB05] V Beletskyy and D Burak. Parallelization of the data encryption standard (des) algorithm. In *Enhanced methods in computer security, biometric and artificial intelligence systems*, pages 23–33. Springer, 2005.

[BCK11] Michael Bauer, Henry Cook, and Brucek Khailany. Cudadma: optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, page 12. ACM, 2011.

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

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

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

[CEN04] CEN. EN 13372:2004 – Road transport and traffic telematics – Profiles for RTTT applications. Standard, European Committee for Standardization, July 2004.

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

[CEN11] CEN. EN 14906:2004 – Road transport and traffic telematics – Electronic fee collection – Application interface definition for dedicated short-range communication. Standard, European Committee for Standardization, October 2011.

[Kwa00] Matthew Kwan. Reducing the gate count of bitslice des. *IACR Cryptology ePrint Archive*, 2000:51, 2000.

[LCG+10] Deguang Le, Jinyi Chang, Xingdou Gou, Ankang Zhang, and Conglan Lu. Parallel aes algorithm for fast data encryption on gpu. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 6, pages V6–1. IEEE, 2010.

[LKC+10] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH computer architecture news*, 38(3):451–460, 2010.

[LWT12] Luis HA Lourenco, Daniel Weingaertner, and Eduardo Todt. Efficient implementation of canny edge detection filter for itk using cuda. In *Computer Systems (WSCAD-SSC), 2012 13th Symposium on*, pages 33–40. IEEE, 2012.

[Mey11] Steven Meyer. Breaking 53 bits passwords with rainbow tables using gpus. page 45, 2011.

[Oec03] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Annual International Cryptology Conference*, pages 617–630. Springer, 2003.

[Reied] Tord Ingolf Reistad. Securing en 15509 against brute-force attacks. Unpublished.

[sin13] Informasjonssikkerhet i AutoPASS-brikker. Report, SINTEF - Teknologi og Samfunn, January 2013.

[ST17] Sandvold Sverre Turter. Attacking message authentication codes in efc using rainbow tables. *Master of Science in Communication Technology*, page 106, 2017.

[Sta99] Data Encryption Standard. Data encryption standard. *Federal Information Processing Standards Publication*, 1999.

[Tay01] Michael AP Taylor. Intelligent transport systems. In *Handbook of transport systems and traffic control*, pages 461–475. Emerald Group Publishing Limited, 2001.

Appendix

```
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include <bitset>
#include <ctime>

#include "cryptopp/modes.h"
#include "cryptopp/des.h"
#include "cryptopp/filters.h"

int convertBinaryToDecimal(long long n){

   int decimalNumber = 0, i = 0, remainder;
   while (n!=0)
   {
      remainder = n%10;
      n /= 10;
      decimalNumber += remainder*pow(2,i);
      ++i;
   }
   return decimalNumber;
}

long long convertDecimalToBinary(int n){
   long long binaryNumber = 0;
   int remainder, i = 1, step = 1;

   while (n!=0)
   {
      remainder = n%2;
      n /= 2;
      binaryNumber += remainder*i;
      i *= 10;
   }
   return binaryNumber;
}
```

```
std::string string_to_hex(const std::string& input){
   static const char* const lut = "0123456789ABCDEF";
   size_t len = input.length();

   std::string output;
   output.reserve(2 * len);
   for (size_t i = 0; i < len; ++i)
   {
      const unsigned char c = input[i];
      output.push_back(lut[c >> 4]);
      output.push_back(lut[c & 15]);
   }
   return output;
}

std::string hex_to_string(const std::string& input){
   static const char* const lut = "0123456789ABCDEF";
   size_t len = input.length();
   if (len & 1) throw std::invalid_argument("odd length");

   std::string output;
   output.reserve(len / 2);
   for (size_t i = 0; i < len; i += 2)
   {
      char a = input[i];
      const char* p = std::lower_bound(lut, lut + 16, a);
      if (*p != a) throw std::invalid_argument("not a hex digit");

      char b = input[i + 1];
      const char* q = std::lower_bound(lut, lut + 16, b);
      if (*q != b) throw std::invalid_argument("not a hex digit");

      output.push_back(((p - lut) << 4) | (q - lut));
   }
   return output;
}

std::string binint_to_hex(int input){
      int decimal = convertBinaryToDecimal(input);
      std::stringstream sstream;
      sstream << std::hex << decimal;
      std::string column_hex = sstream.str();
      return column_hex;
}
```

```
std::string binary_to_hex(const std::string& s){
      std::bitset<64> bs(s);
      unsigned n = bs.to_ulong();
      std::stringstream ss;
      ss << std::hex << n;
      return ss.str();
}

std::string int_to_hex(int input){
      int decimal = input;
      std::stringstream sstream;
      sstream << std::hex << decimal;
      std::string column_hex = sstream.str();
      return column_hex;
}

std::string reduce(std::string hash, int column){

      //Converting hash and column to hexadecimal
      std::string hash_hex = string_to_hex(hash);
      std::string column_hex = int_to_hex(column);

      //The length of the column in hex
      int column_len = column_hex.size();

      //Remove N last characters of the hash (N = column_len)
      std::string sub = hash.substr(0, hash.size()-column_len);

      //Merge hash and column to a new key
      std::string reduced_hash = sub+column_hex;

      return reduced_hash;
}

std::string padTo(std::string str, int num,
   char paddingChar = '0'){
   if(num > str.size())
      return str.insert(0, num - str.size(), paddingChar);
}

std::string key_generation(int keynumber){
      //to binary (long long)
      int rawkey = convertDecimalToBinary(keynumber);
      std::string rawkey_padded = padTo(std::to_string(rawkey),
          56);
      std::string tempkey = "";
      for (int i = 0; i < 56; i=i+7){
            tempkey += rawkey_padded.substr(i, 7);
            tempkey += '1';
      }
      //tempkey to hex
      std::string hex_key = "10101010"+binary_to_hex(tempkey);
      //add padding
      std::string padded_key = padTo(hex_key, 16);
      return padded_key;
}

std::string crypt(std::string inputkey){

      long long hex_key = std::stoul(inputkey, 0, 16);

      byte key[ CryptoPP::DES::DEFAULT_KEYLENGTH ], iv[
          CryptoPP::DES::BLOCKSIZE ];
      memset( key, hex_key, CryptoPP::DES::DEFAULT_KEYLENGTH );
      memset( iv, 0x00, CryptoPP::DES::BLOCKSIZE );

      // String setup
      std::string plaintext = "0004000000000000";
      std::string ciphertext;

      // Create Cipher Text
      CryptoPP::DES::Encryption desEncryption(key,
          CryptoPP::DES::DEFAULT_KEYLENGTH);
      CryptoPP::CBC_Mode_ExternalCipher::Encryption
          cbcEncryption( desEncryption, iv );

      CryptoPP::StreamTransformationFilter
          stfEncryptor(cbcEncryption, new CryptoPP::StringSink(
          ciphertext ) );

      stfEncryptor.Put( reinterpret_cast<const unsigned char*>(
          hex_to_string(plaintext).c_str() ), plaintext.length()
          + 1 );
      stfEncryptor.MessageEnd();

      std::string output1;
      for( int i = 0; i < 8; i++ ) {
            if(int_to_hex((0xFF &
                static_cast<byte>(ciphertext[i]))).length() == 1){
                  output1 = output1 + "0" +int_to_hex((0xFF &
                      static_cast<byte>(ciphertext[i])));
            }
            else{
                  output1 = output1 + int_to_hex((0xFF &
                      static_cast<byte>(ciphertext[i])));
            }
      }
      // Dump Cipher Text (output)
      return output1;
}
```

```
std::pair<std::string, std::string> chain_maker(int chainNumber,
    int chainlength){

      std::string startkey = key_generation(chainNumber);
      std::string endhash = startkey;

      for (int i = 0; i < chainlength; i=i+1){
            if(i<chainlength-1){

                  endhash = reduce(crypt(endhash), i);
            }
            else{
                  endhash = crypt(endhash);
            }
      }
      return make_pair(startkey, endhash);
}

std::string recreate_chain(std::string startkey, std::string mac,
    int chainlength){

      std::cout << "Searching for key to MAC " << mac << "\n" <<
          std::endl;

      std::string hash = crypt(startkey);
      std::string key = startkey;
      std::string wanted;
      int column;

      for (int i = 0; i < chainlength; i=i+1){
            column = i;
            if(hash == mac){
                  wanted = key;
            }
            else{
                  key = reduce(hash, i);
                  hash = crypt(reduce(hash, i));
            }
      }

      std::cout << "Wanted key: " << wanted << std::endl;

   return wanted;
}
```

```
void search(std::string mac, std::pair<std::string, std::string>
    table[], int chains, int chainlength){
      std::cout << "\nLooking for MAC: " << mac << std::endl;
      std::string hash = mac;
      std::string startkey;
      int found = 0;
      for (int i = 0; i < chains; i=i+1){
            if(table[i].second == hash){
                  std::cout << "\n" << hash << " found in chain
                      number " << i << std::endl;
                  std::cout << "The corresponding startkey is "
                      << table[i].first << std::endl;
                  startkey = table[i].first;
                  found = 1;
                  recreate_chain(startkey, hash, chainlength);
                  break;
            }
            if(i == chains-1 && !found){
                  hash = crypt(reduce(hash, chainlength-i-1));
            }
      }
}

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

      int number_of_chains = 10;
      int length_of_chain = 3;

      std::clock_t start;
      double duration;

      start = std::clock();

      std::string mac = "0db095dae115298d";

      //Creating an array of rainbow pairs
      std::pair<std::string, std::string> rt[number_of_chains];

      //Creating pairs of {starkey, endkey} and storing them in
          rt[]
      for (int i = 0; i < number_of_chains; i=i+1){
            rt[i] = chain_maker(i,length_of_chain);
      }
      //Printing the Rainbow Table
      for (int i = 0; i < number_of_chains; i=i+1){
            std::cout << "{ " << rt[i].first << ", " <<
                rt[i].second << " }" << std::endl;
      }
      duration = ( std::clock() - start ) /
          (double)CLOCKS_PER_SEC;
      std::cout<<"Generated " << number_of_chains << " chains of
          length " << length_of_chain << " in: "<< duration <<
          "sec" << std::endl;

      search(mac, rt, number_of_chains, length_of_chain);
}
```