Bachelor thesis

Radboud University

Comparison of chain merge behaviour of TMTO methods

Author:
Ko Stoffelen
Computer Science
s3030547
KoStoffelen@student.ru.nl

First supervisor/assessor:
Dr.ir. Erik Poll
E.Poll@cs.ru.nl

Second supervisor:
Fabian van den Broek, MSc
F.vandenBroek@cs.ru.nl

Second assessor:
Dr. Peter Schwabe
P.Schwabe@cs.ru.nl

January 29, 2013

Abstract

Time-memory trade-off (TMTO) attacks are used to speed up the brute-forcing of encrypted data. Attempts have been made to formally compare four TMTO methods (Hellman, distinguished points, rainbow tables and Kraken), but not all factors could have been included in these comparisons. Chain merges for instance, are suspected to have a significant influence on the percentage of the search space that is covered by a TMTO method. However, it is hard to formally reason about when chain merges might occur. This bachelor thesis aims to gather more empirical data on how the TMTO methods perform compared to each other, mainly with respect to chain merges. It turns out that chain merges indeed have a significant impact. Also, with respect to chain merges, the Kraken method currently performs best.

Contents

Chapter 1

Introduction

During the 26th Chaos Communication Congress in 2009, Karsten Nohl revealed a new approach to cracking the A5/1 cipher that is used in GSM. The approach is used by a piece of software called Kraken. Usually, brute-forcing an encryption costs a lot of time as the entire key space has to be explored. Time-memory trade-off (TMTO) methods have been introduced to accelerate this process. Nohl's new approach made it possible to break the GSM encryption in minutes using nothing but ordinarily available hardware [15]. The approach uses a combination of two TMTO methods: distinguished points [6] and rainbow tables [17]. This combination is said to be more efficient [15, 16]. However, there are reasons to doubt this statement. That is why I am going to measure how efficient the combination actually is, in comparison to the well-established methods, namely Hellman's method, distinguished points and rainbow tables.

Kraken makes it possible for everyone to brute-force the A5/1 cipher with ordinary hardware. However, the statement that its resulting TMTO tables are more efficient than the ones from other methods is a bold one, as no proof is being given. It is also hard to formally disprove the statement, because it is difficult to formally compare some aspects of the methods. It might even be impossible for some aspects, as little progress has been made with the older methods over the past decades. It has been some years and it is still not completely clear as to which method performs best in practice. It is actually an unknown problem, which makes it interesting for research. My measurements will only give an indication for a certain set of ciphers and boundary conditions, but it might end up in an extra argument in favour of some method.

Chapter 2

Theoretical framework

To address the question mentioned above, a bit of theory is required before doing research. First the different categories of cipher algorithms are explained, then brute-forcing and time-memory trade-offs to optimize the brute-forcing, and finally it will be discussed how to measure and compare the efficiency of different TMTO methods.

2.1 Cipher algorithms and hash functions

First of all, hash functions will be discussed and then ciphers. A cipher is an algorithm for performing encryption and/or decryption. There are thousands of different ciphers that can be categorized as shown in Figure 2.1. The two main cipher categories of importance that will be discussed are block ciphers and stream ciphers, where special attention is given to AES and A5/1. In section 2.2.5, it is explained how the kind of cipher influences the use of time-memory trade-off attacks.

2.1.1 Hash functions

A hash function is a function $H\colon D \rightarrow \{0,1\}^N$ that takes data with an arbitrary length and returns a fixed-size bit string, the hash, such that a change to the input will most likely change the hash value. Cryptographic hash functions can be used to for instance verify the integrity of files or messages, or to verify passwords. A useful cryptographic hash function satisfies some important properties [20].

- Preimage resistancy: when given a hash value $H(m) = h$, it is infeasible to find its preimage m. In other words, it is hard to find an inverse H^{-1} such that $H^{-1}(h) = m$.
- Second-preimage resistancy: when given an input m_1, it is infeasible to find a second input m_2 such that $m_1 \neq m_2 \wedge H(m_1) = H(m_2)$.

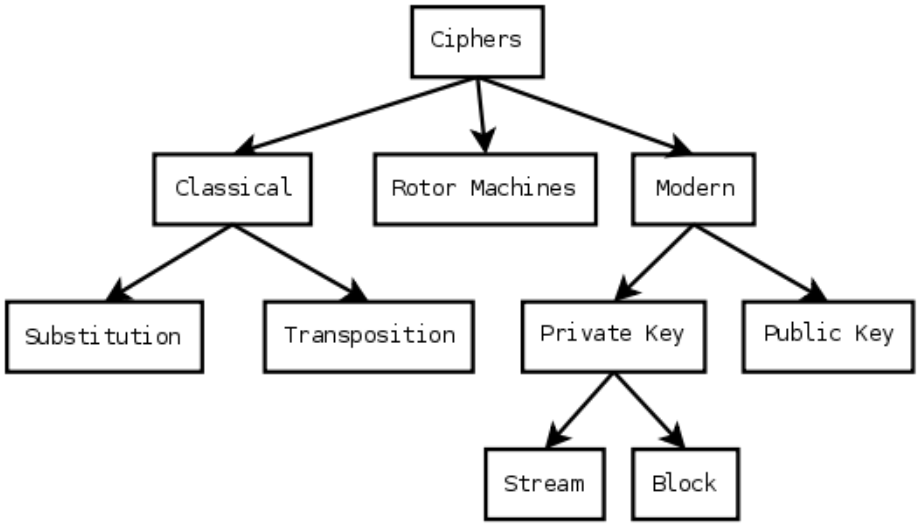

Figure 2.1: Cipher taxonomy [13]

- Collision resistancy: it is infeasible to find two input messages m_1 and m_2 such that $m_1 \neq m_2 \wedge H(m_1) = H(m_2)$.
- A final practical property is that it is easy (i.e. fast) to calculate the hash value of a given input message.

Popular hash functions are for instance MD5 [19] and the functions in the SHA family [8].

2.1.2 Block ciphers

A block cipher is an algorithm that operates on a fixed-length input with a constant transformation that is specified by a symmetric key. Usually, the input, key and output all have the same fixed length, although this is not strictly necessary. Graphically, a block cipher looks as shown in Figure 2.2, where K denotes the key, I the input and O the output.

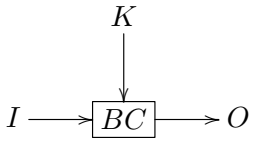

Figure 2.2: Block cipher schema

A larger input message is split in blocks of the required length. The last block is usually padded if it does not meet the required length yet. Each

block is transformed as determined by the key and the cipher algorithm and the resulting ouput blocks are put together again to form the output message. There are a few methods, called modes of operation, on how to handle these multi-block messages.

AES

The Advanced Encryption Standard (AES) is a widely used block cipher. Originally, it was called Rijndael [5] and it was submitted for a NIST contest to supersede the Data Encryption Standard (DES). There are a couple of variants on Rijndael, each with a different key size. AES uses the 128-bit block size version. The Rijndael algorithm repeats a certain cycle a number of times. Each round consists out of four operations, except for the first and the last round. These operations are `SubBytes` (non-linear substitution), `ShiftRows` (transposition), `MixColumns` (mixing) and `AddRoundKey` (XOR state with round key).

2.1.3 Stream ciphers

Stream ciphers are more suitable for variable-length input messages. Using a key K, a stream cipher algorithm generates a keystream of the same length as the input message I. They are then combined (in practice: XORed) to deliver the output message O. Graphically, the situation looks as shown in Figure 2.3.

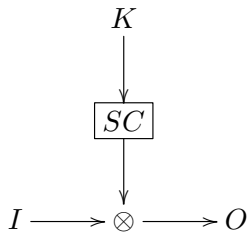

Figure 2.3: Stream cipher schema

A5/1

One of the most widely deployed stream ciphers in the world is the A5/1 cipher that is used in GSM (Globale System for Mobile communication). Initially = it was kept a secret, but reverse engineering has revealed how the encryption works [3]. A5/1 uses three unequally sized linear feedback shift registers (LFSRs) which contain 64 bits in total to generate the keystream. The internal state is initially set with a 64-bit private session key and a publicly known 22-bit frame number. The registers are clocked irregularly

through a majority function over three clock bits, as shown in orange in Figure 2.4.

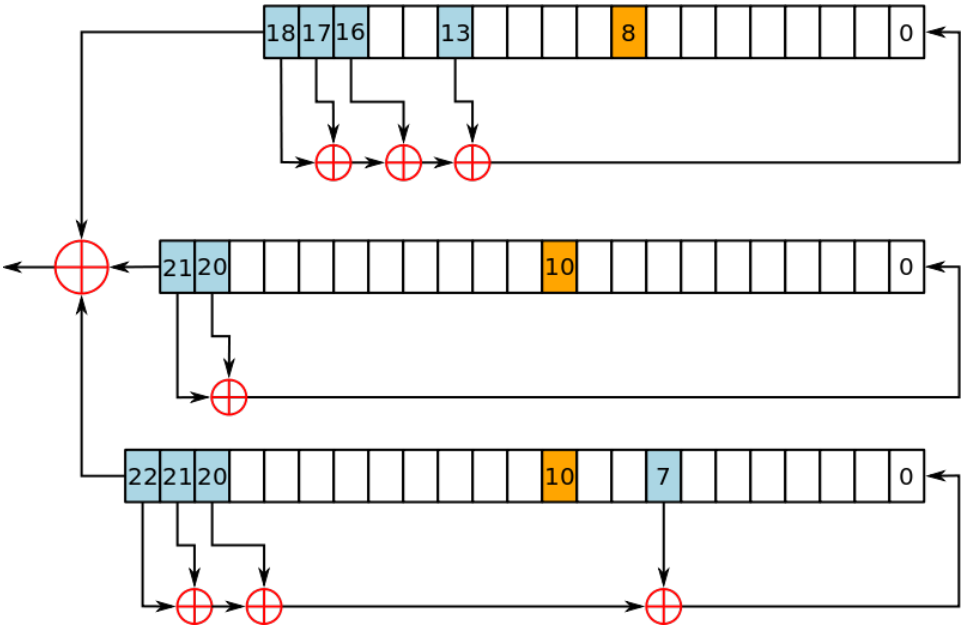

Figure 2.4: A5/1 stream cipher [14]

2.2 Brute-forcing and TMTOs

Assume an attacker tries to crack an encrypted message m'. His goal is to compute the m for which $f(m) = m'$, where f can be an (unsalted) hash function, a block cipher or a stream cipher. In the case of a hash function, m represents the plaintext input. With a block cipher, m usually represents the key and f is applied to a chosen-plaintext input. Finally in the case of a stream cipher, m represents the key that determined the internal state of a cipher f that generated a keystream that was XORed with a chosen plaintext. Even though the attacker knows f, a good cryptosystem should not make it possible to directly compute the preimage. For simplicity, only the word 'key' is used when referring to m from now on.

One thing the attacker can always do is to start brute-forcing: trying out all possible m and see if applying f results in m'. Brute-forcing an encryption can be time-consuming though. For an n-bit key cryptosystem, there are typically $N = 2^n$ possible keys. When N is sufficiently large, it becomes infeasible to try all possibilities.

By being smart, it is possible to greatly reduce the time spent on cracking for repeated attacks. For instance, it is possible to precompute the encryption results of all N possible keys and to store these in a table as tuples

$< m, f(m) >$. This is also called the *offline* phase of the attack. Now when one wants to crack an encrypted message, during the so-called *online* phase, you only have to look it up in your table and you find the key. The downside is that this method requires an enormous amount of memory space and it takes a long time to create the tables.

Something in between would be nice. A time-memory trade-off is where we can speed up an algorithm at the cost of more memory and vice versa.

2.2.1 Hellman

In 1980, Hellman introduced his probabilistic TMTO attack that can brute-force any cryptosystem with N possible keys in on average $N^{2/3}$ operations and $N^{2/3}$ words of memory, after a precompution that requires N operations [9]. This is accomplished by computing $m \times t$ matrices as shown below, m iterative chains of length t, where it is only necessary to store the first and last point of each chain in a table with tuples $< x, f_i^t(x) >$. Usually, m and t are chosen such that $mt^2 = N$, as Hellman did in his article. However, a $m \times t$ matrix satisfying $mt^2 = N$ only covers $1/t$-th of the search space N. That is why a of these matrices are computed, each occupying a different part of the search space. Each of these matrices uses a slightly different f_i, defined as $f_i = h_i \circ f$, where h_i is a simple output modification that differs for each i. A typical example is $h_i(x) = x \oplus i$. The goal of using different functions f_i is to reduce the amount of duplicate chains between tables.

$$\begin{array}{ccccccccc}
x_0 & \rightarrow & f_i(x_0) & \rightarrow & f_i(f_i(x_0)) & \rightarrow & \cdots & \rightarrow & f_i^t(x_0) \\
x_1 & \rightarrow & f_i(x_1) & \rightarrow & f_i(f_i(x_1)) & \rightarrow & \cdots & \rightarrow & f_i^t(x_1) \\
x_2 & \rightarrow & f_i(x_2) & \rightarrow & f_i(f_i(x_2)) & \rightarrow & \cdots & \rightarrow & f_i^t(x_2) \\
\vdots & & \vdots & & \vdots & & \ddots & & \vdots \\
x_m & \rightarrow & f_i(x_m) & \rightarrow & f_i(f_i(x_m)) & \rightarrow & \cdots & \rightarrow & f_i^t(x_m)
\end{array}$$

The online phase consists of the attacker iterating f_i for at most t times over his ciphertext, where he each time checks if the result occurs in the table. If it matches $f_i^t(x_j)$, the attacker can iterate f_i over x_j until he finds his ciphertext. Now he has also computed the preimage of the ciphertext.

2.2.2 Distinguished points

Not much later, Rivest suggested an improvement on Hellman's method [6]. It aims to decrease the number of disk memory accesses required and therefore speeds up the brute-forcing, as disk memory access is usually way more time expensive than computing f_i. This is done by defining an end point DP as a point that satisfies a certain special property, like the first k bits being zero. Now a key has to be looked up in disk memory only if it possesses this

property. The downside is that chains now have a variable length, which makes it hard to make predictions on how much of the keyspace is covered.

Again, a matrices are computed and a function $f_i = h_i \circ f$ is used to iterate over the variable-length chains. The endpoints have a prefix of k bits being zero, which does not have to be stored.

$$\begin{array}{lllllllll}
x_0 & \rightarrow & \cdots & \rightarrow & f_i^p(x_0) = \mathrm{DP}_0 & & & & \\
x_1 & \rightarrow & \cdots & \rightarrow & \cdots & \rightarrow & \cdots & \rightarrow & f_i^q(x_1) = \mathrm{DP}_1 \\
x_2 & \rightarrow & \cdots & \rightarrow & \cdots & \rightarrow & f_i^r(x_2) = \mathrm{DP}_2 & & \\
\vdots & & \ddots & & & & & & \\
x_m & \rightarrow & \cdots & \rightarrow & f_i^*(x_m) = \mathrm{DP}_m & & & &
\end{array}$$

Because it might take a while (or even forever in the case of cycles) before reaching a distinguished point, a constant t_{max} is defined as the maximum chain length. If a chain reaches t_{max} before reaching a distinguished point, the chain is thrown away.

2.2.3 Rainbow tables

For a long time, Rivest's method has been the only improvement to Hellman's TMTO attack. Further work mainly concentrated on optimizing parameters, such as [12]. It was not until 2003 that a new improvement to the algorithm was found by Oechslin, called rainbow tables [17]. A downside of Hellman's algorithm was that collisions within a table resulted in merging of chains. More about that will be said in section 2.3.1. Oechslin combats this by adding an output modification to each step in the chain, instead of just reapplying a cipher function. In this way, each 'column' applies its own f_i. Now, a collision does not result in the merging of chains anymore, unless the collision occurs in the same column. This should prevent a large part of the chain merges.

This time, only one big $m \times t$ matrix is computed, as it is not necessary to compute a different matrices. Again, only the first and last point of each chain have to be stored.

$$\begin{array}{lllllllllll}
x_0 & \rightarrow & f_0(x_0) & \rightarrow & f_1(f_0(x_0)) & \rightarrow & \cdots & \rightarrow & f_t(f_{t-1}(\cdots f_0(x_0)\cdots)) \\
x_1 & \rightarrow & f_0(x_1) & \rightarrow & f_1(f_0(x_1)) & \rightarrow & \cdots & \rightarrow & f_t(f_{t-1}(\cdots f_0(x_1)\cdots)) \\
x_2 & \rightarrow & f_0(x_2) & \rightarrow & f_1(f_0(x_2)) & \rightarrow & \cdots & \rightarrow & f_t(f_{t-1}(\cdots f_0(x_2)\cdots)) \\
\vdots & & \vdots & & \vdots & & \ddots & & \vdots \\
x_m & \rightarrow & f_0(x_m) & \rightarrow & f_1(f_0(x_m)) & \rightarrow & \cdots & \rightarrow & f_t(f_{t-1}(\cdots f_0(x_m)\cdots))
\end{array}$$

2.2.4 Kraken

The methods mentioned above can also be applied to stream ciphers [11]. However to brute-force a stream cipher with a larger key size such as A5/1, lots of further optimizations were required to make this attainable. As said before, Nohl announced that it was more efficient to combine distinguished points and rainbow tables [16] and this was then implemented in the tool called Kraken. An output modification and the cipher function (so together f_i) would be iterated over a start point x_0 until (t_{max} or) a distinguished point DP_{00} was reached, where the first k bits are zero. Then this would be repeated s times with other output modifications, until the end of the chain, DP_{0s}. A single chain now consists of s distinguished point chains.

$$
\begin{array}{ccccccccccc}
x_0 & \to & f_0(x_0) & \to^* & \mathrm{DP}_{00} & \to & f_1(\mathrm{DP}_{00}) & \to^* & \cdots & \to & f_s(\cdots) = \mathrm{DP}_{0s} \\
x_1 & \to & f_0(x_1) & \to^* & \mathrm{DP}_{10} & \to & f_1(\mathrm{DP}_{10}) & \to^* & \cdots & \to & f_s(\cdots) = \mathrm{DP}_{1s} \\
x_2 & \to & f_0(x_2) & \to^* & \mathrm{DP}_{20} & \to & f_1(\mathrm{DP}_{20}) & \to^* & \cdots & \to & f_s(\cdots) = \mathrm{DP}_{2s} \\
\vdots & & \vdots & & \vdots & & \vdots & & \ddots & & \vdots \\
x_m & \to & f_0(x_m) & \to^* & \mathrm{DP}_{m0} & \to & f_1(\mathrm{DP}_{m0}) & \to^* & \cdots & \to & f_s(\cdots) = \mathrm{DP}_{ms}
\end{array}
$$

2.2.5 TMTO on different cipher categories

In general, all TMTO methods can be applied to all aforementioned categories of ciphers. In the following paragraphs, the differences between these applications will be explained in a bit more detail.

Hash functions

When f is some kind of cryptographic hash function, the tables can be used to greatly speed up the search for the preimage x of a hash value $f(x)$. As x's, one chooses all possible plaintexts. A disadvantage here is that hash functions accept an input of arbitrary length. Apart from that, the application of a TMTO attack is self-explanatory.

Block ciphers

Now f is not only dependent on an input x, but also on a certain key K. One of these parameters has to be filled in in order to carry out a TMTO attack. Usually it is more interesting to find K, because then all others parts that were encrypted with the same key can also be decrypted. To do this, one can perform a so-called chosen-plaintext attack, where the attacker already knows the input – perhaps he could enter it somewhere by him- or herself – and the corresponding output. Then a TMTO method can be used to find K.

Stream ciphers

For stream ciphers, the inputs and outputs are more or less the same as with block ciphers. A chosen-plaintext attack can be carried out in order to apply a TMTO method to find K. Or actually, to find the internal state of the cipher which leads to an initialization vector or key. Furthermore, stream ciphers have an extra property that makes TMTO attacks even more useful. When reversing a hash function or block cipher, separate tables have to be created for every part of known plaintext. However when reversing stream ciphers, generic TMTO tables can be used for all pieces of known keystream. So if there are D pieces of known keystream, then the attacker only has to compute a/D tables instead of a to approximate the same probability on success. For a hash function or block cipher, obviously $D = 1$.

2.3 Efficiency

All TMTO methods mentioned earlier have been compared formally in for instance [1, 2, 7, 10, 21]. Some properties are easier to compare than others. Table 2.1 summarizes the comparisons that have been made. All symbols correspond to the symbols that have been used before. t_{avg} is the average actual chain length, somewhere between 1 and t_{max}. A rough estimate is $t_{avg} = t_{max}/2$.

	Hellman	DP	Rainbow	Kraken
Precomputed matrix dimensions	a of $m \times t$	a of $m \times t_{avg}$	$m' \times t$	a of $m \times st_{avg}$
Memory used to store the table(s)	$2ma$	$2ma$	$2m'$	$2ma$
Number of computations of f_i	taD	$t_{avg}aD$	$\frac{t(t+1)}{2}D$	$\frac{s(s+1)}{2}t_{avg}aD$
Number of table seeks	taD	aD	tD	saD

Table 2.1: Formal comparison of TMTO methods [21]

Note the use of m' for rainbow tables instead of m. This is because for fair comparisons, one has to use $m' = ma$. When applying this, $mt^2 = N$ and the attackers want to cover approximately the entire search space N, table 2.1 transforms to table 2.2.

2.3.1 Chain merges and duplicate points

Table 2.2 gives a nice overview on how the different TMTO methods would perform compared to each other. However, not all factors that play a role are

	Hellman	DP	Rainbow	Kraken
Precomputed matrix dimensions	a of $m \times t$	a of $m \times t_{avg}$	$mt \times t$	a of $m \times st_{avg}$
Memory used to store the table(s)	$2mt/D$	$2mt_{avg}/D$	$2mt/D$	$2mt_{avg}/sD$
Number of computations of f_i	t^2	t_{avg}^2	$\frac{t(t+1)}{2}D$	$\frac{s+1}{2}t_{avg}^2$
Number of table seeks	t^2	t_{avg}	tD	t_{avg}

Table 2.2: Formal comparison of TMTO methods satisfying $mt^2 = N$ [21]

included in the calculations. As has been mentioned before briefly, duplicate points and chain merges influence the coverage of the precomputed matrices and thus the performance of the different methods. A chain merge is when the iteration of f_i over two duplicate points in the matrix means that all other subsequent points in the chain are also the same, until at least one of the chains ends. Chain merges are caused by for instance cipher collisions and this strongly depends on the specific cipher topology, which is often unknown or hard to formally reason with. The merges create an overlap in the precomputed matrix, which means that less (unique) points are covered in the TMTO table, thereby influencing the performance of the used method. Of course also the amount of precomputation that is required is influenced by the number of chain merges.

Hellman

When enough chains have been computed using Hellman's method, a new chain will eventually cover parts of the search space that were already covered in an earlier chain. Duplicate points in different chains will always cause a chain merge, if it were not for the output modifications. In fact, this was the whole reason for introducing a tables instead of 1 and applying f_i instead of just f. With these modifications, a pair of duplicate points do not have to cause a chain merge if the points reside in different tables. Of course chain merges are still possible within a single table. The appearance of duplicate points is also where the rule of thumb $mt^2 = N$ comes from. Hellman has computed a lower bound on the number of unique points that are added when computing a new chain, for the case where $mt^2 = N$ is satisfied [9].

Theorem 1. *Within a single Hellman TMTO table that is computed by a random function f_i and that satisfies $mt^2 = N$, each chain contains on average at least $3/4$-th points that are unique up to that point.*

The corresponding proof is found in [9].

Distinguished points

Just as with Hellman's method, the distinguished points method also computes a tables and uses f_i in order to reduce the number of chain merges. Again, they can only occur within the same table. The distinguished point method has the advantage that it is very easy to spot chain merges, because a merge will cause the endpoints of the two chains to be exactly the same. One only has to look at the endpoints. It is also possible to eventually remove the shortest of these two chains (because it covers the smallest part of the search space) and to compute a new one.

Rainbow tables

Both Hellman's method and distinguished points may have chain merges within the same table. The rainbow table method was invented to circumvent this problem by using the different 'colours' for each 'column'. A pair of duplicate points will now only cause a chain merge if they reside in the same column. Because this is a lot less likely, it is no longer necessary to precompute multiple tables. This method is suspected to have fewer chain merges than the two above.

Kraken

The Kraken method combines the distinguished points and the rainbow table method. It also combines their behaviour regarding chain merges. Chain merges are easy to spot, because only the endpoints have to be taken into account. They also only occur within the same table and when duplicate points reside in the same rainbow 'colour'. However, the reason for suspecting that the Kraken method might actually perform worse with respect to chain merges, is that the distinguished points property makes it more likely for duplicate points to occur within the same 'colour'. When more iterations are carried out before reaching a distinguished point, more points are covered and thus the chance on duplicate points increases. Duplicate points within two subchains in the same 'colour' will cause the corresponding distinguished endpoints to be equivalent, which causes a chain merge. It is unclear if and how this outweighs the advantages of the Kraken method.

Chapter 3

Empirical comparison of TMTO methods

The goal is to measure how the different TMTO methods perform compared to each other on a chosen-plaintext attack, specifically with respect to the number of chain merges. In order to carry out this research, the four aforementioned TMTO methods were implemented and used on a small 16-bit variant of the block cipher AES [4, 18]. The fact that it is a very small cipher makes it easier to experiment with. One can now simply count the number of chain merges after precomputation and the resulting table can be stored in RAM.

3.1 Programming environment

Everything was implemented in C++, a personal preference, and compiled with the MinGW suite. Of course any programming language and platform can be chosen as long as it has sufficient memory. The program has a modular set-up, where each TMTO method consists of two functions (`precompute_table` and `count_chain_merges`) in a separate source code file. The whole program includes running these two functions and printing some statistics, and usually terminates in under 100 milliseconds.

3.2 Point coverage

For this 16-bit cipher, $N = 2^{16}$. A simple chosen plaintext attack for a block cipher means that $D = 1$. About N (not necessarily unique) points are covered in the precomputation, in order to make a clean comparison of the four TMTO methods. For the Hellman method for instance, this means that $mta = N$. A few possible combinations for m, t, a (and s) are tried with each TMTO method.

The exact number of computed points ($N_{computed}$), the number of unique points (N_{unique}) and the number of chain merges are measured by counting them. In this case, being a unique points means being unique among all points that have been computed so far within a single execution of the program. For two duplicate points, the first appearance does count as a unique point, but the second does not. The number of unique points does not necessarily say anything about the number of chain merges, because a pair of duplicate points do not have to cause a chain merge if, for instance, they subsequently use a different f_i. However, it it still an interesting number to keep track of. The number of chain merges is measured as a ratio with respect to the total number of chains (ma for Hellman's method). Again, the first appearance of a partly duplicate chain does not count as a merge, but the second does. A chain that merges in itself (i.e. contains a cycle) is also counted as a merge.

3.3 Other parameters

There are a few other parameters that have to be chosen. One of them being the output modification h_i in order to construct $f_i = h_i \circ f$. $h_i(x) = x \oplus i$ is being applied to each byte of the ciphertext. This h_i has to be different for each table and/or column, which in practice means that a (Hellman, distinguished points), t (rainbow tables) or as (Kraken) may not supersede 2^8. For a 16-bit cipher, this seems acceptable.

Also, a choice has to been made regarding how to mark a distinguished point. For the measurements, k ranges around $n/4$. This means that the expected $t_{avg} \approx 2^{n/4} = 2^4$ when $n = 16$. This is an acceptable value for comparisons with the other methods.

Chapter 4

Results and analysis

The experimental results will be presented per TMTO method. Conclusions will be drawn in the form of a series of conjectures. Everything in this chapter, including the aforementioned conjectures, only applies to the 16-bit AES-like cipher that has been used and can not trivially be generalized.

4.1 Hellman

a	m	t	$N_{computed}$	N_{unique}	Chain merge ratio
2^3	2^3	2^{10}	2^{16}	7349	$64/64 = 1$
2^3	2^7	2^6	2^{16}	25009	$841/1024 \approx 0.821$
2^3	2^{10}	2^3	2^{16}	38919	$2937/8192 \approx 0.359$
2^4	2^8	2^4	2^{16}	38226	$1457/4096 \approx 0.356$
2^5	2^6	2^5	2^{16}	37886	$719/2048 \approx 0.351$
2^6	2^3	2^7	2^{16}	35549	$265/512 \approx 0.518$
2^6	2^4	2^6	2^{16}	37632	$362/1024 \approx 0.354$
2^7	2^2	2^7	2^{16}	38476	$167/512 \approx 0.326$
2^7	2^6	2^3	2^{16}	41417	$270/8192 \approx 0.033$
2^7	2^7	2^2	2^{16}	42237	$293/16384 \approx 0.018$

Table 4.1: Experimental results Hellman's method

First of all, there is no important reason for choosing powers of 2 as values of the parameters. It's just that it's easy to calculate with.

Second, it is interesting to see how the configurations with $mt^2 = N$ performed compared to the other settings. The $mt^2 = N$ tests all performed about equally well. The other tests ended up with completely different results, where mainly the final two tests, with $mt^2 < N$, scored really good.

Conjecture 2. *$mt^2 = N$ is not the most efficient relation with respect to storage of unique points and chain merges.*

The major downside is that these $mt^2 < N$ cases require far more memory ($2ma$) to store the tables. Taking that into account, the cases with a relatively low value for mt performed best. This makes sense for chain merges, because they can only occur within the same table. If that table is small, one can expect a low amount of chain merges. This means that the output modifications are of significant importance.

When looking at tests with the same number of output modification classes (i.e. having the same a), it can easily be seen that it is better to have a high m compared to t. Increasing t seems to linearly increase the chain merge ratio. This is an interesting property, though again, this increases memory usage.

4.2 Distinguished points

a	m	t_{max}	k	$N_{computed}$	N_{unique}	Chain merge ratio
2^4	136	2^7	5	65563	31033	$856/2176 \approx 0.393$
2^4	184	2^6	5	65632	33260	$1166/2944 \approx 0.396$
2^4	263	2^7	4	65550	34968	$1133/4208 \approx 0.269$
2^4	291	2^5	5	65557	33905	$1847/4656 \approx 0.397$
2^4	351	2^5	4	65481	36484	$1542/5776 \approx 0.267$
2^5	94	2^6	5	65519	36398	$826/3008 \approx 0.275$
2^5	145	2^5	5	65737	37175	$1179/4640 \approx 0.254$
2^6	46	2^6	5	66224	38801	$484/2944 \approx 0.164$
2^6	69	2^6	4	65536	39386	$440/4416 \approx 0.100$
2^6	72	2^5	5	65147	38490	$746/4608 \approx 0.162$
2^6	123	2^4	5	65335	37814	$1191/7872 \approx 0.151$
2^6	137	2^4	4	65401	39587	$763/8768 \approx 0.087$

Table 4.2: Experimental results distinguished points

With distinguished points, an extra variable k is available to tune with. Again the test cases with a high value for a perform really good and within the same a, it is better to have a high m compared to t.

When taking memory usage into account, tests with a low value for mt scored better with respect to chain merges compared to tests with an approximately equal value for $2ma$. The reason is of course the same as with Hellman's method.

When k increases by 1, the expected chain length before reaching a distinguished point multiplies with a factor 2. However, when t_{max} is not increased by the same factor, more chains will never reach a distinguished point and will be thrown away. This increases the time that is required for computation and thus makes the configuration less efficient in that sense. Increasing k is also accompanied by a lower m in order to compute approximately the same number of points.

Conjecture 3. *Take $k' = k + l$, then t'_{max} should be $t'_{max} = 2^l t_{max}$.*

It is somewhat hard to compare, but it seems that the distinguished point method performs about equally well as Hellman's method with respect to unique points and chain merges for tests where $2ma$ is about equal. This is expected, because the only differences between the two methods do not influence the way points are computed.

Conjecture 4. *The distinguished point method performs equally well as Hellman's method with respect to unique points and chain merges.*

4.3 Rainbow tables

m	t	$N_{computed}$	N_{unique}	Chain merge ratio
2^6	2^{10}	2^{16}	29532	$25/64 \approx 0.391$
2^7	2^9	2^{16}	33011	$46/128 \approx 0.359$
2^8	2^8	2^{16}	35710	$90/256 \approx 0.352$
2^9	2^7	2^{16}	35940	$178/512 \approx 0.348$
2^{10}	2^6	2^{16}	36294	$358/1024 \approx 0.350$
2^{11}	2^5	2^{16}	36522	$728/2048 \approx 0.356$
2^{12}	2^4	2^{16}	37272	$1449/4096 \approx 0.354$
2^{13}	2^3	2^{16}	38132	$3022/8192 \approx 0.369$

Table 4.3: Experimental results rainbow tables

All the points for which $m > t$ naturally also yield $mt^2 = N$, if one substitutes m with ma for fair comparisons. One can then always find a value for a such that $a = t$, because $m > t$. From $mta = N$, it follows that $mt^2 = N$. These configurations perform approximately equally well. They are also comparable to the Hellman $mt^2 = N$ cases. A lower t means more unique points – which makes sense – and oddly enough also more chain merges – which is strange –, and it costs more memory usage due to a higher m.

However, below a certain threshold (around $m = t$) a higher value for t will mean a higher chance on eventually causing a merge with some other chain and this correlation starts to show up.

The next conjecture, Conjecture 5, is quite interesting, as the invention of rainbow tables was supposed to decrease the appearance of chain merges.

Conjecture 5. *When around N points are computed, a rainbow table is not more efficient than using Hellman's method with respect to chain merges.*

However, rainbow tables still have a small (constant) speed-up in the online phase.

4.4 Kraken

a	m	t_{max}	s	k	$N_{computed}$	N_{unique}	Chain merge ratio
2^3	258	2^4	2^4	5	65555	38907	$294/2064 \approx 0.142$
2^4	129	2^5	2^4	5	65081	37866	$342/2064 \approx 0.166$
2^4	130	2^5	2^3	5	66065	38192	$337/2080 \approx 0.162$
2^4	132	2^6	2^4	5	65448	33977	$611/2112 \approx 0.289$
2^4	150	2^2	2^4	6	65396	40393	$277/2400 \approx 0.115$
2^4	181	2^2	2^4	5	65844	41177	$176/2896 \approx 0.061$
2^4	258	2^5	2^4	4	65446	36644	$750/4128 \approx 0.182$
2^4	289	2^5	1	5	65554	34167	$1784/4624 \approx 0.386$
2^4	290	2	2^4	5	65591	40375	$436/4640 \approx 0.094$
2^4	292	2^3	2^2	5	65537	39863	$607/4672 \approx 0.130$

Table 4.4: Experimental results Kraken

It is interesting to note that when $s = 1$, the test performs about equally well as the distinguished point method. This was expected, because the methods are then theoretically exactly equal. A negligible difference is caused by the current implementation of the output modification, which is slightly different for the two methods.

Second, for the bottom three cases where $a = 2^4$, $k = 5$ and $m \approx 290$, it can be seen that is it better to choose a higher s. The suspected reason is that then simply more different f_i's are used.

When looking at equal memory consumption, again $2ma$, having a lower value for t seems to lower the chain merge ratio.

And the most interesting one:

Conjecture 6. *The Kraken method is more efficient compared to Hellman, distinguished points and rainbow tables with respect to chain merges.*

The Kraken method delivers a chain merge ratio that is approximately a factor 2 lower than with the other methods. Perhaps this improvement is simply due to the introduction of an extra parameter s. We've seen before that cases with a low value for mt scored relatively good with respect to chain merges. Under the assumption that approximately N points are computed, the introduction of s might make it easier to keep mt relatively low, without influencing the disk memory consumption: $N = mta$ vs. $N = mst_{avg}a$.

4.5 General results

Chain merges indeed appeared to be of major importance when it comes to the coverage of a TMTO matrix, with chain merge ratios ranging from 6% to 40%. A general strategy to minimize this ratio is to minimize the number of times that the same f_i is applied when iterating. Basically there are two ways of doing so.

Conjecture 7. *By decreasing the length of a chain, the chain merge ratio decreases.*

This is really part of the trade-off, for a decrease in required computation time causes an increase of required memory if one wants to stick to the same coverage. For instance for Hellman's method, if $mta = N$, decreasing t means that m or a have to be increased, thus also increasing $2ma$.

Conjecture 8. *By decreasing the number of points that are computed by the same f_i, the chain merge ratio decreases.*

Note that this number of points is equal to mt_{avg} for the Kraken method. Decreasing this value is where the difference can perhaps be made. For instance by including a globally unique chain number in the output modification, or setting $m = 1$, the f_i's become unique for every chain, making it impossible for a merge to occur. The downside is that this chain number has to be stored with each chain, as it can not be retrieved once the TMTO table gets sorted on end points, and that it causes a huge increase of the time required for the online phase. This example is highly impractical of course, but it raises a focus point on which future TMTO algorithms could perhaps improve on the current ones.

Chapter 5

Conclusion and future work

5.1 Conclusion

This empirical research has had some interesting outcomes. For the 16-bit AES-like cipher, it turns out that chain merges are of significant importance, with merge ratios ranging from 6% to 40%. Hellman's rule of thumb of $mt^2 = N$ does not seem to be the most efficient relation with respect to storage of unique points and chain merges. Hellman's method, the distinguished point method and rainbow tables all perform about equally well with respect to these chain merges. This is surprising, because rainbow tables are often said to prevent a big part of the chain merges. The Kraken method performs better by approximately a factor 2. This is probably due to the decreased number of points that were computed with the same output modification. The fewer points per f_i, the fewer possibilities for a chain merge to occur.

5.2 Future work

Due to the scope of the project and/or practical limitations, there are a number of things worth mentioning that might be interesting for further investigations.

First of all, there are (at least) two papers that each explain a different 16-bit version of AES. One of these ([18]) has been used here, but it is worth verifying whether the results are consistent with the other version. In a more general sense, it is definitely worth verifying whether the results are consistent with completely different ciphers and/or different output modifications. The appearance of duplicate points in TMTO tables is a property of the specific cipher topology. For practical reasons only one small cipher has been put to the test, but this does not say anything about other cases.

Second, while running a test using the distinguished point method, it turned out that t_{max} had not been implemented yet. The iteration of f_i

caused a lot of relatively short cycles within a chain, some even without a distinguished point, causing an infinite loop. Of course t_{max} prevents this and the chances on cycles are smaller when using a cipher with a longer key length, but this behaviour might be interesting. Perhaps an f_i with lots of cycles might even be more secure, because it is more 'resistant' against a TMTO attack.

Third, the implementation of cryptographic algorithms is rather hard. The tiniest mistake is critical and usually there is not a simple way to verify whether the algorithm actually does what it is supposed to do. It has been the same here. Although reasonable thinking created the thought that the code that has been used to perform these measurements is semantically correct, there are no guarantees. This is why even verification with the same cipher is desirable.

Fourth, the correlation between the number of unique points and the number of chain merges has not been made clear yet. It would be interesting to gather empirical data on the number of duplicate points that are actually caused by chain merges. I.e. how far within two chains does a merge usually occur.

Finally, and perhaps the most interesting one, it looks like it might be possible to improve the TMTO methods by adjusting the output modification such that more unique f_i's are used.

Bibliography

[1] Elad Barkan, Eli Biham, and Adi Shamir. Rigorous Bounds on Cryptanalytic Time/Memory Tradeoffs. In *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2006.

[2] Alex Biryukov and Adi Shamir. Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers. In *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, 2000.

[3] M. Briceno, I. Golberg, and D. Wagner. A Pedagogical Implementation of the GSM A5/1 and A5/2 "voice privacy" Encryption Algorithms. Retrieved from http://cryptome.org/gsm-a512.htm, 1999.

[4] C. Cid, S. Murphy, and M. Robshaw. Small Scale Variants of the AES. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption – FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 2005.

[5] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer, 1st edition, 2002.

[6] D. Denning. *Cryptography and Data Security*, page 100. Addison-Wesley Publishing Company, Boston, 1st edition, 1982.

[7] Imran Erguler and Emin Anarim. A New Cryptanalytic Time-Memory Trade-Off for Stream Ciphers. In *Computer and Information Sciences – ISCIS 2005*.

[8] Patrick Gallagher and Carlos M. Gutierrez. FIPS PUB 180-3: Secure Hash Standard (shs). Technical report, National Institute of Standards and Technology, October 2008.

[9] M.E. Hellman. A Cryptanalytic Time-Memory Trade-Off. *IEEE Transactions on Information Theory*, IT-26(4):401–406, July 1980.

[10] Jin Hong, Kyung Chul Jeong, Eun Young Kwon, In-Sok Lee, and Daegun Ma. Variants of the Distinguished Point Method for Cryptanalytic Time Memory Trade-offs. In *Information Security Practice and Experience Conference – ISPEC 2008*, volume 4991 of *Lecture Notes in Computer Science*, pages 131–145. Springer, 2008.

[11] Jan Krhovják, Ondřej Šiler, Paul Leyland, and Jiří Kůr. TMTO attacks on stream ciphers – theory and practice. In *Security and Protection of Information 2011*, pages 66–78, Brno, 2011. Brno University of Defence.

[12] Koji Kusuda and Tsutomu Matsumoto. Optimization of Time-Memory Trade-Off Cryptanalysis and Its Applications to DES, FEAL-32, and Skipjuck. *IEICE Transactions on Fundamentals*, E79-A(1):35–48, January 1996.

[13] Matt_Crypto and Quasar Jarosz, 2007. Retrieved from Wikipedia: `http://en.wikipedia.org/wiki/File:Cipher-taxonomy.svg`.

[14] Matt_Crypto and Tsaitgaist, 2009. Retrieved from Wikimedia Commons: `http://commons.wikimedia.org/w/index.php?title=File:A5-1_GSM_cipher.svg`.

[15] Karsten Nohl. Attacking phone privacy. Convention slides during BlackHat USA 2010, retrieved from `https://media.blackhat.com/bh-us-10/whitepapers/Nohl/BlackHat-USA-2010-Nohl-Attacking.Phone.Privacy-wp.pdf`, 2010.

[16] Karsten Nohl and Chris Paget. GSM – SRSLY? Congress slides during CCC 2009, retrieved from `http://events.ccc.de/congress/2009/Fahrplan/attachments/1519_26C3.Karsten.Nohl.GSM.pdf`, December 2009.

[17] P. Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2003.

[18] Raphael Chung-Wei Phan. Mini Advanced Encryption Standard (Mini-AES): A Testbed of Cryptanalysis Students. Technical report, Swinburne Sarawak Institute of Technology, Malaysia, 2002.

[19] R. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. Internet Engineering Task Force, 1992.

[20] Phillip Rogaway and Thomas Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance.

In Bimal Roy and Willi Meier, editors, *Fast Software Encryption*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

[21] Fabian van den Broek. Comparison of Time-Memory Trade-Off Attacks. To appear, title may change.

Appendix A

Source code

A.1 common.h

```
#ifndef COMMON_H
#define COMMON_H

//general includes
#include <cstring>
#include <climits>
#include <fstream>
#include <cstdlib>
#include <iostream>
#include <vector>

//common settings
//a (for rainbow, a = 1)
#define NUMBER_OF_TABLES 16
//m
#define TABLE_LENGTH 292
//t (or tmax for DP/Kraken)
#define CHAIN_LENGTH 8
//s (used only with Kraken)
#define NUMBER_OF_CHAINS 4
//k (unused with Hellman/rainbow, k < CHAR_BIT, pow(2,k) <= a)
#define DP_SIZE 5
//in bytes (CIPHER_SIZE <= sizeof(int))
#define CIPHER_SIZE 2
//method (METHOD_HELLMAN, METHOD_DP, METHOD_KRAKEN, METHOD_RAINBOW)
#define METHOD_KRAKEN
//cipher (CIPHER_AES)
#define CIPHER_AES
//N
const unsigned int SEARCH_SPACE = 1u << CIPHER_SIZE*CHAR_BIT;

//typedefs
typedef unsigned char byte;
typedef byte TMTOTable[TABLE_LENGTH][2][CIPHER_SIZE];
typedef TMTOTable TMTOTables[NUMBER_OF_TABLES];
struct NoOfPoints
{
    unsigned int total, unique;
};

//functiondefs
byte *cipher(const byte input[], const unsigned len);
void next_plaintext(byte keyword[]);
unsigned int ciphertext_to_uint(const byte[]);

//include logic
#ifdef CIPHER_AES
#include "aes.h"
#endif
```

```
#ifdef METHOD_HELLMAN
#include "hellman.h"
#endif
#ifdef METHOD_DP
#include "dp.h"
#endif
#ifdef METHOD_KRAKEN
#include "kraken.h"
#endif
#ifdef METHOD_RAINBOW
#include "rainbow.h"
#endif

#endif
```

A.2 main.cpp

```
/*
 * TMTO method comparator
 *             by Ko Stoffelen
 *
 * v1.0     29-01-2013
 *
 */
#include "common.h"

/**
 * Apply the cipher to an arbitrarty-length input
 * Currently uses 16-bit AES in ECB mode
 */
byte *cipher(const byte input[], const unsigned int len)
{
    #ifdef CIPHER_AES
    byte chosen_plaintext[CIPHER_SIZE] = {0xC3, 0xF1};
    byte *output = new byte[len];
    for(unsigned int i = 0; i < len; i += CIPHER_SIZE)
    {
        byte AESkey[CIPHER_SIZE];
        memset(AESkey, 0, CIPHER_SIZE);
        for(unsigned int j = 0; j < CIPHER_SIZE && i*CIPHER_SIZE+j < len; ++j)
            AESkey[j] = input[i*CIPHER_SIZE+j];
        AES(chosen_plaintext, AESkey);
        for(unsigned int j = 0; j < CIPHER_SIZE && i*CIPHER_SIZE+j < len; ++j)
            output[i*CIPHER_SIZE+j] = chosen_plaintext[j];
    }
    return output;
    #endif
}

/**
 * Generate the next plaintext by 'counting'
 * Currently the first byte has to be thought of as containing the least significant bits
 */
void next_plaintext(byte keyword[])
{
    ++keyword[0];
    for(unsigned int i = 0; keyword[i] == 0 && i < CIPHER_SIZE-1; ++i)
        ++keyword[i+1];
}

/**
 * Assumes 0 < CIPHER_SIZE <= sizeof(unsigned int)
 */
unsigned int ciphertext_to_uint(const byte ciphertext[])
{
    unsigned int result = ciphertext[0];
    for(unsigned int i = 1; i < CIPHER_SIZE; ++i)
    {
```

```
        result <<= CHAR_BIT;
        result |= ciphertext[i];
    }
    return result;
}

/**
 * For testing purposes
 */
void write_table_to_file(TMTOTables table, const char filename[])
{
    std::ofstream file(filename);
    if(file)
    {
        file << std::hex;
        for(unsigned int i = 0; i < NUMBER_OF_TABLES; ++i)
            for(unsigned int j = 0; j < TABLE_LENGTH; ++j)
            {
                for(unsigned int k = 0; k < CIPHER_SIZE; ++k)
                    file << 'x' << (int)table[i][j][0][k];
                file << '\t';
                for(unsigned int k = 0; k < CIPHER_SIZE; ++k)
                    file << 'x' << (int)table[i][j][1][k];
                file << std::endl;
            }
        file << std::dec;
        file.close();
    }
}

/**
 * Main function
 */
int main()
{
    TMTOTables table;
    std::cout << "> Precompute the TMTO table" << std::endl;
    const NoOfPoints counter = precompute_table(table);
    std::cout << "< Done with " << counter.total << " points (" << counter.unique << "
        unique)" << std::endl;

    //write_table_to_file(table, "table.txt");

    std::cout << "> Analyze the TMTO table" << std::endl;
    unsigned int chain_merges = count_chain_merges(table);
    std::cout << "< " << chain_merges << " chain merges out of " << (NUMBER_OF_TABLES*
        TABLE_LENGTH) << " chains (ratio: " << ((double)chain_merges/(double)(
        NUMBER_OF_TABLES*TABLE_LENGTH)) << ")" << std::endl;

    std::cout << "> Done!" << std::endl << std::endl;

    return 0;
}
```

A.3 aes.h

```
#ifndef AES_H
#define AES_H

#include "common.h"

extern void AES(byte[], byte[]);

/**
 * Modular multiplication table for GF(2^4)
 */
const byte MultiplicationTable[16][16] =
    {{(byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0,(byte)
```

```
        0x0,(byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0},
    {(byte)0x0,(byte)0x1,(byte)0x2,(byte)0x3,(byte)0x4,(byte)0x5,(byte)0x6,(byte)0x7,(byte)
        0x8,(byte)0x9,(byte)0xA,(byte)0xB,(byte)0xC,(byte)0xD,(byte)0xE,(byte)0xF},
    {(byte)0x0,(byte)0x2,(byte)0x4,(byte)0x6,(byte)0x8,(byte)0xA,(byte)0xC,(byte)0xE,(byte)
        0x3,(byte)0x1,(byte)0x7,(byte)0x5,(byte)0xB,(byte)0x9,(byte)0xF,(byte)0xD},
    {(byte)0x0,(byte)0x3,(byte)0x6,(byte)0x5,(byte)0xC,(byte)0xF,(byte)0xA,(byte)0x9,(byte)
        0xB,(byte)0x8,(byte)0xD,(byte)0xE,(byte)0x7,(byte)0x4,(byte)0x1,(byte)0x2},
    {(byte)0x0,(byte)0x4,(byte)0x8,(byte)0xC,(byte)0x3,(byte)0x7,(byte)0xB,(byte)0xF,(byte)
        0x6,(byte)0x2,(byte)0xE,(byte)0xA,(byte)0x5,(byte)0x1,(byte)0xD,(byte)0x9},
    {(byte)0x0,(byte)0x5,(byte)0xA,(byte)0xF,(byte)0x7,(byte)0x2,(byte)0xD,(byte)0x8,(byte)
        0xE,(byte)0xB,(byte)0x4,(byte)0x1,(byte)0x9,(byte)0xC,(byte)0x3,(byte)0x6},
    {(byte)0x0,(byte)0x6,(byte)0xC,(byte)0xA,(byte)0xB,(byte)0xD,(byte)0x7,(byte)0x1,(byte)
        0x5,(byte)0x3,(byte)0x9,(byte)0xF,(byte)0xE,(byte)0x8,(byte)0x2,(byte)0x4},
    {(byte)0x0,(byte)0x7,(byte)0xE,(byte)0x9,(byte)0xF,(byte)0x8,(byte)0x1,(byte)0x6,(byte)
        0xD,(byte)0xA,(byte)0x3,(byte)0x4,(byte)0x2,(byte)0x5,(byte)0xC,(byte)0xB},
    {(byte)0x0,(byte)0x8,(byte)0x3,(byte)0xB,(byte)0x6,(byte)0xE,(byte)0x5,(byte)0xD,(byte)
        0xC,(byte)0x4,(byte)0xF,(byte)0x7,(byte)0xA,(byte)0x2,(byte)0x9,(byte)0x1},
    {(byte)0x0,(byte)0x9,(byte)0x1,(byte)0x8,(byte)0x2,(byte)0xB,(byte)0x3,(byte)0xA,(byte)
        0x4,(byte)0xD,(byte)0x5,(byte)0xC,(byte)0x6,(byte)0xF,(byte)0x7,(byte)0xE},
    {(byte)0x0,(byte)0xA,(byte)0x7,(byte)0xD,(byte)0xE,(byte)0x4,(byte)0x9,(byte)0x3,(byte)
        0xF,(byte)0x5,(byte)0x8,(byte)0x2,(byte)0x1,(byte)0xB,(byte)0x6,(byte)0xC},
    {(byte)0x0,(byte)0xB,(byte)0x5,(byte)0xE,(byte)0xA,(byte)0x1,(byte)0xF,(byte)0x4,(byte)
        0x7,(byte)0xC,(byte)0x2,(byte)0x9,(byte)0xD,(byte)0x6,(byte)0x8,(byte)0x3},
    {(byte)0x0,(byte)0xC,(byte)0xB,(byte)0x7,(byte)0x5,(byte)0x9,(byte)0xE,(byte)0x2,(byte)
        0xA,(byte)0x6,(byte)0x1,(byte)0xD,(byte)0xF,(byte)0x3,(byte)0x4,(byte)0x8},
    {(byte)0x0,(byte)0xD,(byte)0x9,(byte)0x4,(byte)0x1,(byte)0xC,(byte)0x8,(byte)0x5,(byte)
        0x2,(byte)0xF,(byte)0xB,(byte)0x6,(byte)0x3,(byte)0xE,(byte)0xA,(byte)0x7},
    {(byte)0x0,(byte)0xE,(byte)0xF,(byte)0x1,(byte)0xD,(byte)0x3,(byte)0x2,(byte)0xC,(byte)
        0x9,(byte)0x7,(byte)0x6,(byte)0x8,(byte)0x4,(byte)0xA,(byte)0xB,(byte)0x5},
    {(byte)0x0,(byte)0xF,(byte)0xD,(byte)0x2,(byte)0x9,(byte)0x6,(byte)0x4,(byte)0x8,(byte)
        0x1,(byte)0xE,(byte)0xC,(byte)0x3,(byte)0x8,(byte)0x7,(byte)0x5,(byte)0xA}
};

#endif
```

A.4 aes.cpp

```
#include "aes.h"
//assume r = 2, c = 2, e = 4
//src: http://www.ma.rhul.ac.uk/~sean/smallAES-fse05.pdf
//src: http://staff.guilan.ac.ir/staff/users/rebrahimi/fckeditor_repo/file/mini-aes-spec.pdf

/**
 * Lookup table for the S-box corresponding to GF(pow(2,4))
 */
byte SBoxLookup(const byte data)
{
    switch(data & 0xF)
    {
        case 0x0: return (byte)0xE;
        case 0x1: return (byte)0x4;
        case 0x2: return (byte)0xD;
        case 0x3: return (byte)0x1;
        case 0x4: return (byte)0x2;
        case 0x5: return (byte)0xF;
        case 0x6: return (byte)0xB;
        case 0x7: return (byte)0x8;
        case 0x8: return (byte)0x3;
        case 0x9: return (byte)0xA;
        case 0xA: return (byte)0x6;
        case 0xB: return (byte)0xC;
        case 0xC: return (byte)0x5;
        case 0xD: return (byte)0x9;
        case 0xE: return (byte)0x0;
        case 0xF: return (byte)0x7;
        default: std::cerr << "Error in SBoxLookup with data " << data << std::endl;
    }
    return -1;
```

```
}

/**
 * First of the small scale round operations
 * Apply the S-boxes
 */
void SubBytes(byte dataasarray[][2])
{
    dataasarray[0][0] = SBoxLookup(dataasarray[0][0]);
    dataasarray[0][1] = SBoxLookup(dataasarray[0][1]);
    dataasarray[1][0] = SBoxLookup(dataasarray[1][0]);
    dataasarray[1][1] = SBoxLookup(dataasarray[1][1]);
}

/**
 * Second of the small scale round operations
 * Left shift row i with i places
 */
void ShiftRows(byte dataasarray[][2])
{
    const byte h = dataasarray[1][0];
    dataasarray[1][0] = dataasarray[1][1];
    dataasarray[1][1] = h;
}

/**
 * Third of the small scale round operations
 * Matrix multiplication with [2+1    2]
 *                            [2    2+1]
 */
void MixColumns(byte dataasarray[][2])
{
    byte h = dataasarray[0][0];
    dataasarray[0][0] = MultiplicationTable[3][dataasarray[0][0]] ^ MultiplicationTable[2][
        dataasarray[1][0]];
    dataasarray[1][0] = MultiplicationTable[2][h] ^ MultiplicationTable[3][dataasarray
        [1][0]];
    h = dataasarray[0][1];
    dataasarray[0][1] = MultiplicationTable[3][dataasarray[0][1]] ^ MultiplicationTable[2][
        dataasarray[1][1]];
    dataasarray[1][1] = MultiplicationTable[2][h] ^ MultiplicationTable[3][dataasarray
        [1][1]];
}

/**
 * Fourth of the small scale round operations
 * XOR with key and generate new round key
 */
void AddRoundKey(byte dataasarray[][2], byte keyasarray[][2], const byte round)
{
    dataasarray[0][0] ^= keyasarray[0][0];
    dataasarray[0][1] ^= keyasarray[0][1];
    dataasarray[1][0] ^= keyasarray[1][0];
    dataasarray[1][1] ^= keyasarray[1][1];

    byte newkeyasarray[2][2];
    newkeyasarray[0][0] = keyasarray[0][0] ^ SBoxLookup(keyasarray[1][1]) ^ round;
    newkeyasarray[1][0] = keyasarray[1][0] ^ newkeyasarray[0][0];
    newkeyasarray[0][1] = keyasarray[0][1] ^ newkeyasarray[1][0];
    newkeyasarray[1][1] = keyasarray[1][1] ^ newkeyasarray[0][1];

    keyasarray[0][0] = newkeyasarray[0][0];
    keyasarray[0][1] = newkeyasarray[0][1];
    keyasarray[1][0] = newkeyasarray[1][0];
    keyasarray[1][1] = newkeyasarray[1][1];
}

/**
 * Implements a 16-bit AES variant
```

```
 * Assumes length(data) == length(key) == CIPHER_SIZE == 2
 * Converts data/key temporarily to a 2x2 matrix, which makes it easier to apply all round
    operations
 */
void AES(byte data[], byte key[])
{
    byte dataasarray[2][2] = {{data[0] >> 4, data[1] >> 4},{data[0] & 0xF, data[1] & 0xF}};
    byte keyasarray[2][2] = {{key[0] >> 4, key[1] >> 4},{key[0] & 0xF, key[1] & 0xF}};

    AddRoundKey(dataasarray, keyasarray, 1);
    SubBytes(dataasarray);
    ShiftRows(dataasarray);
    MixColumns(dataasarray);
    AddRoundKey(dataasarray, keyasarray, 2);
    SubBytes(dataasarray);
    ShiftRows(dataasarray);
    AddRoundKey(dataasarray, keyasarray, 3);


    data[0] = (dataasarray[0][0] << 4) | dataasarray[1][0];
    data[1] = (dataasarray[0][1] << 4) | dataasarray[1][1];
}
```

A.5 hellman.h

```
#ifndef HELLMAN_H
#define HELLMAN_H

#include "common.h"

extern NoOfPoints precompute_table(TMTOTables);
extern unsigned int count_chain_merges(const TMTOTables);

#endif
```

A.6 hellman.cpp

```
#include "hellman.h"

#ifdef METHOD_HELLMAN

/**
 * Precompute a Hellman TMTO table by starting at plaintext 0x0000
 * Return struct of total number of visisted points and number of unique visited points
 */
NoOfPoints precompute_table(TMTOTables table)
{
    NoOfPoints counter = {0, 0};
    bool checkarray[SEARCH_SPACE];
    memset(checkarray, false, SEARCH_SPACE);
    byte plaintext[CIPHER_SIZE];
    byte ciphered[CIPHER_SIZE];
    memset(plaintext, 0, CIPHER_SIZE);
    for(unsigned int i = 0; i < NUMBER_OF_TABLES; ++i)
        for(unsigned int j = 0; j < TABLE_LENGTH; ++j)
        {
            //start with new plaintext
            memcpy(ciphered, plaintext, CIPHER_SIZE);
            for(unsigned int k = 0; k < CHAIN_LENGTH; ++k)
            {
                //increase point counters
                ++counter.total;
                unsigned int index = ciphertext_to_uint(ciphered);
                if(!checkarray[index])
                {
                    checkarray[index] = true;
                    ++counter.unique;
                }
```

```
                //iterate and apply the output modification
                memcpy(ciphered, cipher(ciphered, CIPHER_SIZE), CIPHER_SIZE);
                for(unsigned int l = 0; l < CIPHER_SIZE; ++l)
                    ciphered[l] ^= (byte)i;
            }

            //store begin and endpoints
            memcpy(table[i][j][0], plaintext, CIPHER_SIZE);
            memcpy(table[i][j][1], ciphered, CIPHER_SIZE);

            next_plaintext(plaintext);
        }
    return counter;
}

/**
 * Return the number of chain merges
 * Two duplicate points within the same table cause a chain merge
 */
unsigned int count_chain_merges(const TMTOTables table)
{
    unsigned int counter = 0;
    bool checkarray[SEARCH_SPACE];
    for(unsigned int i = 0; i < NUMBER_OF_TABLES; ++i)
    {
        memset(checkarray, false, SEARCH_SPACE);
        for(unsigned int j = 0; j < TABLE_LENGTH; ++j)
        {
            byte tmp[CIPHER_SIZE];
            memcpy(tmp, table[i][j][0], CIPHER_SIZE);
            unsigned int index = ciphertext_to_uint(tmp);
            for(unsigned int k = 0; k < CHAIN_LENGTH && !checkarray[index]; ++k)
            {
                checkarray[index] = true;

                memcpy(tmp, cipher(tmp, CIPHER_SIZE), CIPHER_SIZE);
                for(unsigned int l = 0; l < CIPHER_SIZE; ++l)
                    tmp[l] ^= (byte)i;

                index = ciphertext_to_uint(tmp);
            }
            if(checkarray[index])
                ++counter;
        }
    }
    return counter;
}

#endif
```

A.7 dp.h

```
#ifndef DP_H
#define DP_H

#include "common.h"

extern NoOfPoints precompute_table(TMTOTables);
extern unsigned int count_chain_merges(const TMTOTables);

#endif
```

A.8 dp.cpp

```
#include "dp.h"

#ifdef METHOD_DP
```

```
/**
 * Precompute a distinguished points TMTO table by starting at plaintext 0x0000
 * Return struct of total number of visisted points and number of unique visited points
 */
NoOfPoints precompute_table(TMTOTables table)
{
    NoOfPoints counter = {0, 0};
    bool checkarray[SEARCH_SPACE];
    memset(checkarray, false, SEARCH_SPACE);
    byte plaintext[CIPHER_SIZE];
    byte ciphered[CIPHER_SIZE];
    memset(plaintext, 0, CIPHER_SIZE);
    std::vector<unsigned int> checkindexarray;
    for(unsigned int i = 0; i < NUMBER_OF_TABLES; ++i)
        for(unsigned int j = 0; j < TABLE_LENGTH; ++j)
        {
            //start with new plaintext
            memcpy(ciphered, plaintext, CIPHER_SIZE);
            checkindexarray.clear();
            checkindexarray.push_back(ciphertext_to_uint(ciphered));
            for(unsigned int k = 0; ciphered[0] >> (CHAR_BIT - DP_SIZE) != (byte)0x0 && k <
                CHAIN_LENGTH; ++k)
            {
                //iterate and apply the output modification
                memcpy(ciphered, cipher(ciphered, CIPHER_SIZE), CIPHER_SIZE);
                for(unsigned int l = 0; l < CIPHER_SIZE; ++l)
                    ciphered[l] ^= (byte)i;

                //temporarily store point counters
                checkindexarray.push_back(ciphertext_to_uint(ciphered));
            }

            //store begin and endpoints if loop actually ended in DP
            if(ciphered[0] >> (CHAR_BIT - DP_SIZE) == (byte)0x0)
            {
                memcpy(table[i][j][0], plaintext, CIPHER_SIZE);
                memcpy(table[i][j][1], ciphered, CIPHER_SIZE);

                //only then increase point counters
                for(unsigned int k = 0; k < checkindexarray.size(); ++k)
                {
                    ++counter.total;
                    if(!checkarray[checkindexarray[k]])
                    {
                        checkarray[checkindexarray[k]] = true;
                        ++counter.unique;
                    }
                }
            }
            else
                --j;

            next_plaintext(plaintext);
        }
    return counter;
}

/**
 * Return the number of chain merges
 * Two duplicate endpoints within the same table cause a chain merge
 */
unsigned int count_chain_merges(const TMTOTables table)
{
    unsigned int counter = 0;
    bool checkarray[SEARCH_SPACE];
    for(unsigned int i = 0; i < NUMBER_OF_TABLES; ++i)
    {
        memset(checkarray, false, SEARCH_SPACE);
```

```
        for(unsigned int j = 0; j < TABLE_LENGTH; ++j)
        {
            const unsigned index = ciphertext_to_uint(table[i][j][1]);
            if(checkarray[index])
                ++counter;
            else
                checkarray[index] = true;
        }
    }
    return counter;
}

#endif
```

A.9 rainbow.h

```
#ifndef RAINBOW_H
#define RAINBOW_H

#include "common.h"
#include <algorithm>

struct RainbowPoint
{
    unsigned int ciphertext, row;
};

extern NoOfPoints precompute_table(TMTOTables);
extern unsigned int count_chain_merges(const TMTOTables);

#endif
```

A.10 rainbow.cpp

```
#include "rainbow.h"

#ifdef METHOD_RAINBOW

/**
 * Precompute a rainbow TMTO table by starting at plaintext 0x0000
 * Return struct of total number of visisted points and number of unique visited points
 */
NoOfPoints precompute_table(TMTOTables table)
{
    NoOfPoints counter = {0, 0};
    bool checkarray[SEARCH_SPACE];
    memset(checkarray, false, SEARCH_SPACE);
    byte plaintext[CIPHER_SIZE];
    byte ciphered[CIPHER_SIZE];
    memset(plaintext, 0, CIPHER_SIZE);
    for(unsigned int i = 0; i < TABLE_LENGTH; ++i)
    {
        //start with new plaintext
        memcpy(ciphered, plaintext, CIPHER_SIZE);
        for(unsigned int j = 0; j < CHAIN_LENGTH; ++j)
        {
            //increase point counters
            ++counter.total;
            unsigned int index = ciphertext_to_uint(ciphered);
            if(!checkarray[index])
            {
                checkarray[index] = true;
                ++counter.unique;
            }

            //iterate and apply the output modification
            memcpy(ciphered, cipher(ciphered, CIPHER_SIZE), CIPHER_SIZE);
            for(unsigned int k = 0; k < CIPHER_SIZE; ++k)
```

```
                ciphered[k] ^= (byte)j;
        }

        //store begin and endpoints
        memcpy(table[0][i][0], plaintext, CIPHER_SIZE);
        memcpy(table[0][i][1], ciphered, CIPHER_SIZE);

        next_plaintext(plaintext);
    }
    return counter;
}

/**
 * Equality operator for RainbowPoint type
 */
bool operator==(const RainbowPoint& a, const RainbowPoint& b)
{
    return a.ciphertext == b.ciphertext && a.row == b.row;
}

/**
 * Return the number of chain merges
 * Two duplicate points within the same column cause a chain merge
 * RainbowPoints are used to also store the column number
 */
unsigned int count_chain_merges(const TMTOTables table)
{
    unsigned int counter = 0;
    std::vector<RainbowPoint> checkarray;
    for(unsigned int i = 0; i < TABLE_LENGTH; ++i)
    {
        byte tmp[CIPHER_SIZE];
        memcpy(tmp, table[0][i][0], CIPHER_SIZE);
        RainbowPoint rp = {ciphertext_to_uint(tmp), 0};
        for(unsigned int j = 0; j < CHAIN_LENGTH && std::find(checkarray.begin(), checkarray
            .end(), rp) == checkarray.end(); ++j)
        {
            checkarray.push_back(rp);

            memcpy(tmp, cipher(tmp, CIPHER_SIZE), CIPHER_SIZE);
            for(unsigned int k = 0; k < CIPHER_SIZE; ++k)
                tmp[k] ^= (byte)j;

            rp.ciphertext = ciphertext_to_uint(tmp);
            rp.row = j;
        }
        if(std::find(checkarray.begin(), checkarray.end(), rp) != checkarray.end())
            ++counter;
    }
    return counter;

}

#endif
```

A.11 kraken.h

```
#ifndef KRAKEN_H
#define KRAKEN_H

#include "common.h"

extern NoOfPoints precompute_table(TMTOTables);
extern unsigned int count_chain_merges(const TMTOTables);

#endif
```

A.12 kraken.cpp

```
#include "kraken.h"

#ifdef METHOD_KRAKEN

/**
 * Precompute a Kraken TMTO table by starting at plaintext 0x0000
 * Return struct of total number of visisted points and number of unique visited points
 */
NoOfPoints precompute_table(TMTOTables table)
{
    NoOfPoints counter = {0, 0};
    bool checkarray[SEARCH_SPACE];
    memset(checkarray, false, SEARCH_SPACE);
    byte plaintext[CIPHER_SIZE];
    byte ciphered[CIPHER_SIZE];
    memset(plaintext, 0, CIPHER_SIZE);
    std::vector<unsigned int> checkindexarray;
    for(unsigned int i = 0; i < NUMBER_OF_TABLES; ++i)
        for(unsigned int j = 0; j < TABLE_LENGTH; ++j)
        {
            //start with new plaintext
            memcpy(ciphered, plaintext, CIPHER_SIZE);
            checkindexarray.clear();
            checkindexarray.push_back(ciphertext_to_uint(ciphered));
            for(unsigned int k = 0; ciphered[0] >> (CHAR_BIT - DP_SIZE) != (byte)0x0 && k <
                NUMBER_OF_CHAINS; ++k)
                for(unsigned int l = 0; ciphered[0] >> (CHAR_BIT - DP_SIZE) != (byte)0x0 &&
                    l < CHAIN_LENGTH; ++l)
                {
                    //iterate and apply the output modification
                    memcpy(ciphered, cipher(ciphered, CIPHER_SIZE), CIPHER_SIZE);
                    for(unsigned int m = 0; m < CIPHER_SIZE; ++m)
                        ciphered[m] ^= (byte)((i << 4) | k); //assume a,s <= 16

                    //temporarily store points because of counters
                    checkindexarray.push_back(ciphertext_to_uint(ciphered));
                }

            //store begin and endpoints if loop actually ended in DP
            if(ciphered[0] >> (CHAR_BIT - DP_SIZE) == (byte)0x0)
            {
                memcpy(table[i][j][0], plaintext, CIPHER_SIZE);
                memcpy(table[i][j][1], ciphered, CIPHER_SIZE);

                //only then increase point counters
                for(unsigned int k = 0; k < checkindexarray.size(); ++k)
                {
                    ++counter.total;
                    if(!checkarray[checkindexarray[k]])
                    {
                        checkarray[checkindexarray[k]] = true;
                        ++counter.unique;
                    }
                }
            }
            else
                --j;

            next_plaintext(plaintext);
        }
    return counter;
}

/**
 * Return the number of chain merges
 * Two duplicate endpoints within the same table cause a chain merge
```

```
 */
unsigned int count_chain_merges(const TMTOTables table)
{
    unsigned int counter = 0;
    bool checkarray[SEARCH_SPACE];
    for(unsigned int i = 0; i < NUMBER_OF_TABLES; ++i)
    {
        memset(checkarray, false, SEARCH_SPACE);
        for(unsigned int j = 0; j < TABLE_LENGTH; ++j)
        {
            const unsigned index = ciphertext_to_uint(table[i][j][1]);
            if(checkarray[index])
                ++counter;
            else
                checkarray[index] = true;
        }
    }
    return counter;
}

#endif
```