Time Memory Tradeoff Implementation on Copacobana

Stefan Spitz

27.06.2007

Master-Thesis
Ruhr-Universität Bochum

Tutors:
Dipl. Ing. Tim Güneysu, Dipl. Inf. Andy Rupp
Chair for Communication Security
Prof. Dr.-Ing. Christof Paar

Gesetzt am November 17, 2007 um 16:26 Uhr.

Contents

1 Abstract

1.1 none

The purpose of this thesis is to present and implement a method for decreasing the time which is neccesary to obtain a key used by a block cipher. This is done for the commonly known block cipher *Data Encryption Standard* DES, but the methods and implementation can easily be adjusted for any block cipher. The reduction of the neccesary time is achieved by utilizing the special purpose hardware *COPACOBANA* which will be introduced in chapter **??**.

Chapters two to four introduce the theory about time memory tradeoffs and the COPACOBANA, while chapters five and six address the more practical aspects of the attack like the analysis of the methods presented in chapter three in regard to the hardware limitations of COPACOBANA as well as the chosen method with its pros and cons. Chapter seven deals with the implementation on both sides, COPACOBANA and PC, and points out some of the most important aspects which came up during the design process as well as some first results during a reduced attack scenario. The final chapter eight introduces means to further optimize the results using a modified method, multiple computers or possible improvements for the COPACOBANA which would all reduce the time for the attack considerably.

2 Basics

This chapter presents the required conditions for the attack, the basics of block ciphers and the functionality of DES. In preparation to the next chapter, the two basic methods on which a time memory tradeoff is based, the *brute force* and *table lookup* attack, are introduced as well.

2.1 Environment

The general set-up consists of two areas, the required hardware and the attack scenario in which the time memory tradeoff attack takes place. The first piece of hardware to mention is the COPACOBANA, which stands for *Cost Optimized Parallel Code Breaker*. It consists of an array of FPGAs *Field Programmable Gate Array* which can be designed to compute encryptions multiple times faster than a normal retail computer and a more detailed introduction to the COPACOBANA is done in chapter 4. In addition to this a computer is needed to control the COPACOBANA during the computations and to execute the attack later on. For this, an Intel Core 2 Quad, each Core with a frequency of 2.4 GHz, is used. The storage capacity amounts to 2.5 terabyte, provided by 5 hard disks with 500 gigabyte of memory each.
The second area relates to the scenario in which the attack takes place. The different attack scenarios are the *ciphertext only* attack where the attacker only knows the ciphertext, a *known plaintext* attack in which the attacker knows the ciphertext and the corresponding plaintext. Similar to this is the *chosen plaintext* scenario in which the attacker can even choose the plaintext which is then encrypted into the ciphertext (which he receives as well).
A different approach is presented with a *chosen ciphertext* attack in which the attacker receives the corresponding plaintext for a ciphertext he obtained earlier. The scenarios called *adaptive chosen plaintext* and *adaptive chosen ciphertext* assume that the attacker not only has access to the encryption or decryption device once, but multiple times, hence he can use small modifications on successive plain/ciphertexts to gain more information about the used key as with only a single pair.
Relevant for the time memory tradeoff attack in this theses are the scenarios in which the plaintext can be chosen, therefore the known and (adaptive) chosen plaintext scenarios. In theory, the chosen chipertext attack would work as well,

but the time spent for the computations would be wasted since the probability to get two identical ciphertexts is marginally small.
One may think that if the plaintext is already known, the effort to find the key is wasted. If the whole plaintext is known and the plaintext is only used once, then an attack would indeed be a waste of time. But with "'known plaintext"' only 64 bits are known. These 64 bit translate into only 8 ASCII characters which not neccesarily give much information about the whole plaintext. But these 64 bit can contain exactly the informations needed for a successful attack. Consider an encrypted textfile which starts with: "'Dear Ladies and Gentlemen"' which is a common form of address provides more than the neccesary 8 characters. Or a picture in the .jpg format uses more than 64 bit for their header which again provides enough "'known plaintext"'. Almost any header information which changes rarely and uses at least 64 bits of information can be used for a time memory tradeoff attack, e.g. word or pdf files, pictures, zip files and many more. Any of these plaintexts can be used and it is up to the person who executes the time memory tradeoff to decide which one to choose from.

2.2 Block Ciphers

A block cipher consists of a symmetric cipher with a fixed input and output length. The key which is used in the transformation of a plaintext into a ciphertext (or a ciphertext into a plaintext) is known to both the sender and receiver of the ciphertext, hence called a *symmetric key*. One well known block cipher is the *Data Encryption Standard* DES.

2.2.1 Data Encryption Standard

The DES was first published in 1975 by IBM and approved as an federal US standard in 1976. It uses a 56 bit symmetric key to encrypt a 64 bit plaintext into 64 bit ciphertext and vice versa and is built upon the basic structure of an *feistel network* which is shown in 2.1.

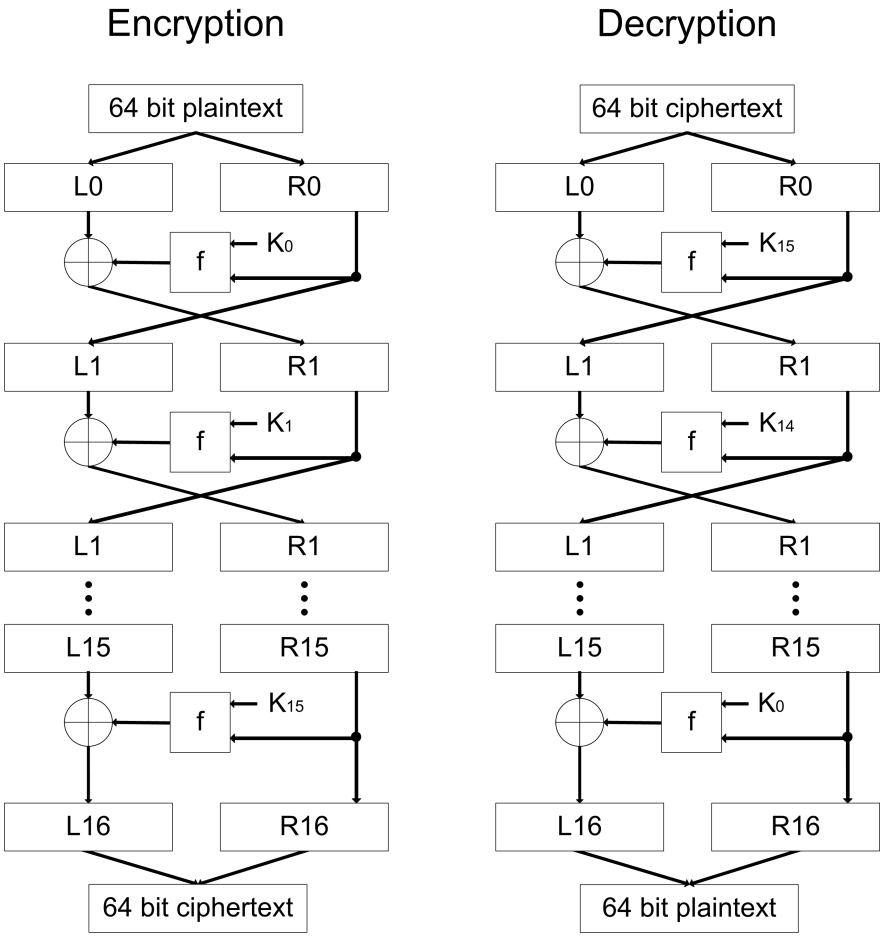

Figure 2.1: Feistel Network used for the DES

K_0 to K_{15} represent the different subkeys used in each round of the DES encryption process. These 16 subkeys are derived from the 56 bit symmetric key by using a *key scheduling* function which shifts the 56 bit in each round and returns only 48 bits for each subkey K_i. The function f is shown in 2.2.

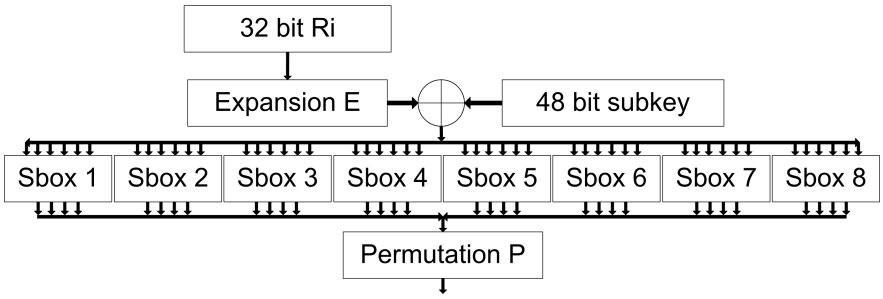

Figure 2.2: f-function of DES

The input for f are R_i and the subkey, hence the expansion function which expands the 32 bits from R_i to match the 48 bits from the subkey. Each of the 8 S-boxes maps a 6 bit value onto a 4 bit value, therfore the 48 input bits are mapped to 32 output bits which are then permuted before being xored with the 32 bits from L_i. The 8 S-boxes represent the primary security against cryptanalytical attacks like the differential or linar attack. Especially the differential attack is promising while less than 16 rounds of DES are used, but a full 16 round DES, as stated in the standard, is as secure against the differential attack as against a brute force attack in a practical manner since the differential attack on a 16 round DES needs about 2^{47} plaintext/cipertext pairs which translates into about 1 petabyte of known plaintext.

2.3 Brute Force

The purpose of an *brute force* attack is to obtain the key K which was used in encrypting a given plaintext P into the associated ciphertext C. This is done by trying every possible key out of the keyspace N with the function

$$enc_{K_i}(P) = C', K_i \in N$$

and the check if $C = C'$ until a match is found. The complexity of this attack is related directly to the number of possible keys, therefore the size of the keyspace N. Figure 2.3 illustrates the number of encryptions which are done to carry out a brute force attack on a cipher with a keyspace of 2^{56} in a *Worst Case* scenario which is later used in the explanation of the time memory tradeoff method.

Each step (trying a key) in the brute force attack is equivalent to a single encryption resulting in 2^{56} encryptions in a worst case scenario. If a Pentium 4

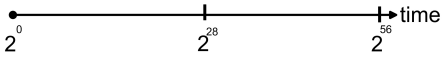

Figure 2.3: Brute force time usage

with 3.2 GHz is used for the attack then about 2 million ($\approx 2^{21}$) encryptions can be done per second. This results in a total of about $2^{56-21} = 2^{35}$ seconds ≈ 1000 years which pass before each key is checked with a single computer, and every time a new key is used to encrypt a plaintext another brute force attack needs to be started. Therefore, brute forcing a 56 bit cipher with a single PC isn't a good idea.

2.4 Table Lookup

Different to the brute force attack is the table lookup or dictionary attack. This attack is fixed for a certain plaintext which is chosen before the generation of the dictionary, hence called a *chosen or known plaintext* attack. Similar to the brute force attack, every possible key is used for the encryption of the plaintext. The difference to the brute force is that the resulting ciphertext is not checked and discarded but stored in a table together with the used key. After the table is generated, the attack uses this table to search for a match between the obtained ciphertext C and all stored ciphertexts C_i' within the table, returning the corresponding key for the matched C_i'. The graphic below shows the space which is used for storing the tables, and again it is used later for explaining the time memory tradeoff method.

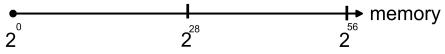

Figure 2.4: Table lookup memory usage

Considering the 56 bit cipher from before, the generated table consists of 2^{56} different entries, each representing a key and the corresponding ciphertext. Since each key is used for the table generation only the resulting ciphertexts need to be stored using the 56 bit value of a key as the position within the table. Each ciphertext is 64 bit long, so 8 byte of memory are needed storing a single ciphertext. In addition to the fact that generating the table takes as much time as a brute force attack, storing 2^{56} ciphertexts uses up 512 petabyte which translates into 512000 hard disks, each with a capacity of 1 terabyte. The advantage compared to the brute force attack is that once the table is stored finding a key for the fixed plaintext is a simple search within the table in $log_2(2^{56}) = 56$ steps.

Like the brute force attack a table loopkup is not viable for common use because memory solutions reaching into the petabytes are quite expensive.

3 TMTO Methods

A time-memory tradeoff is a compromise between the time needed for a brute force attack and the memory requirement of a table lookup method and like the table lookup mezhod, the time memory tradeoff is used in a known or chosen plaintext scenario.

The combination of both the brute force attack with its many computations and the table lookup with the huge memory requirements is illustrated as in figure 3.1.

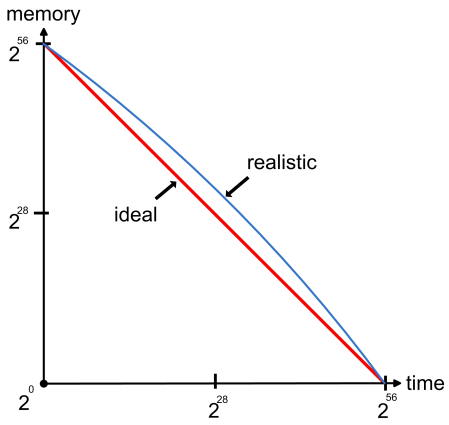

Figure 3.1: Time and Memory Complexity of a TMTO

The attack with a table lookup symbolizes the vertical axis. It has a memory complexity of 2^{56} with only a single encryption for verifying the key candidate, therefore a time complexity of 1. The brute force method is located on the horizontal axis with a time complexity of 2^{56} encryptions and a memory usage of only 1, which is taken up by the currently computed ciphertext during the attack.
An ideal tradeoff between those two methods moves along the red (ideal) line. If, for example, the memory usage is cut by half (to 2^{55}) regarding a table lookup, the number of computations during the time memory tradeoff attack rises to 2. The more realistic approach is symbolized by the blue line which means that halving the memory usage increases the time complexity by at least 2. The run of this curve is affected by the efficiency of the method used for the tradeoff. It depends on how many calculations are wasted during the generation of the table with the details of *wasted computations* being presented during the respective

sections of the different methods.

The procedure of all time memory tradeoffs is as follows: A TMTO consists of two phases, the *precomputation phase* and the *online phase*. During the precomputation phase a table is built by computing so-called *chains*. These chains contain several keys which are connected in a certain way that depends on the used TMTO method and block cipher.
Each generated table consists of multiple chains. The total number of used keys in a tables is called the *coverage*. Each generated table uses up some memory, but only the first and last entry of each chain needs to be stored with a memory usage denoted as $Memory_{method}$ for all tables while the number of computations to generate them is given as $Time_{method}$.
The success rate $Success_{method}$ depends on the coverage of all tables related to the number of all possible keys N for the chosen block cipher. In the following figure 3.2 a basic scheme of an TMTO table is shown:

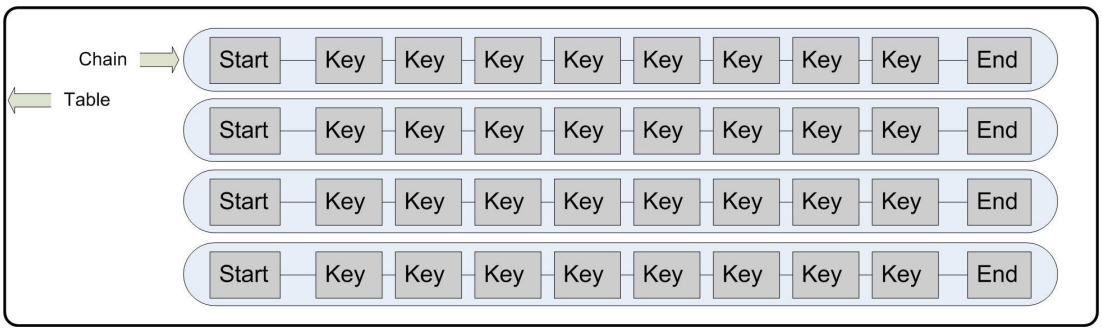

Figure 3.2: Basic scheme of a single TMTO table

Based on the examples given in chapter 2, halving the memory usage to 2^{55} would generate a table with only this amount of entries with each chain within covering 2 keys. On the other hand a reduction of the time complexity to 2^{55} generates a table with only 2 entries with each chain covering 2^{55} keys. In any case the precomputation needs at least as much computations as a brute force attack, but with an reduced memory usage of an table lookup.

The search for a key which was used to encrypt some ciphertext is done during the online phase. The stored tables are searched through for this key which takes up $Time'_{method}$ calculations. The advantage of this effort is that this table needs to be generated only once for a fixed plaintext and that the time and memory for the attack can be greatly reduced, compared to a brute force approach, respectively the table lookup. Each of the following methods use a slightly different approach on both the online and precomputation phase which will now be described in more detail.

3.1 Hellman's Method

In 1980 Martin Hellman published his famous TMTO attack in [Hel80]. His main idea can roughly be described as follows: Similar to the naive table lookup method described in Section 2, this method tries to precompute all possible key-ciphertext pairs in advance by encrypting P with all N possible keys. However, to reduce memory requirements these pairs are organized in several chains of a fixed length. These chains are generated deterministically and are uniquely identified by their respective start and end points. In this way, it suffices to save its start and end point to restore a chain later on. In the online phase of the attack, one then simply needs to identify the right chain containing the given ciphertext to get the wanted key by restoring the chain. The details of the two phases of Hellman's TMTO attack are described in the following.

3.1.1 Precomputation Phase

The time memory tradeoff table is generated during the precomputation phase. This table is fixed for a single plaintext P which is chosen at the beginning of the table generation. This plaintext is encrypted by using the encryption function E from our one-way function, while the key k used during this encryption can be chosen arbitrary out of all possible N keys. The result of this encryption is the ciphertext C. This ciphertext is to be used as key for the next encryption of our plaintext. To this end, a so-called *reduction function* R is applied to C. The role of R is to reduce the bit length of C to the bit length of a key for the cipher E (if this is necessary) and to perform a re-randomization of the output. For instance, in the case of DES Hellman suggests to simply choose a fixed subset of 56 bits from C. However, there are many other possibilities for realizing the reduction function R. The reduced ciphertext is called intermediate point X and is used as the key for the next encryption or round with $X_i = R\big(E_{X_{i-1}}(P)\big)$. The composition of E and R is called *step-function* F with $F_k := R\big(E_k(P)\big)$. This step function is applied as often as t times producing a chain which starts with the chosen key k and ends with the last intermediate point X. Only the first key or start point SP and the last intermediate point or end point EP needs to be stored in the table to reproduce the chain during the online phase. The generation of a single chain within the table is shown in figure 3.3.

After a chain is stored, the next key or start point is chosen. This is done m times, each time with a different SP. At the end the table contains m different chains each represented by a stored (SP, EP)-tuple. In every chain $t + 1$ keys were used which results in $m \times (t + 1)$ keys throughout the table generation. Despite having $t + 1$ keys in each chain only t keys can actually be counted for the coverage because the first key SP has no reproducable preimage, therefore this key is useless regarding our online phase. Ideally our coverage for a table

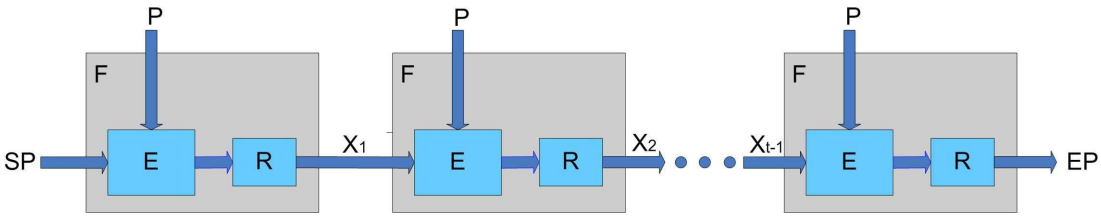

Figure 3.3: Precomputation scheme of Hellman's table generation

is $m \times t$ but unfortunately, the occurence of a key in a table is not necessarily unique since there is a chance that two chains collide and merge or that a chain runs in a loop.

Merges and Loops

A merge is an event where at some point during the table generation two different chains contain the same intermediate point. This can happen if the one-way function uses a key which is shorter than the ciphertext. Since the reduction function R is not a bijection an intermediate point produced from ciphertext 1 can be equal to an intermediate point which was produced from ciphertext 2. AES 128 would not produce any merges since the key length of 128 bit is equal to the length of the ciphertext. DES on the other hand uses 56 bit keys and 64 bit ciphertexts, therefore merges are possible. If a merge happens then the two chains produce the same intermediate point until the maximum chainlength $t+1$ is reached. Figure 3.4 illustrates the event of an merge.

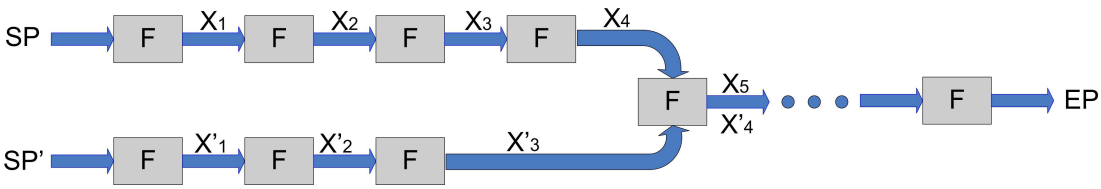

Figure 3.4: Merge of two chains

If intermediate point X_4 and X'_3 have the same value then a merge occurs. Both chains now produce exactly the same intermediate points with $X_{4+1} = X'_{3+i} \; \forall \; i$. This affects the coverage of keys since it is expected that two different chains cover $2 \times (t)$ keys. But because these two chains merge at step 4 in chain 1 (respectively step 3 in chain 2) the coverage drops considerably. Only 3 keys from chain 1 and t keys from chain 2, resulting in $t+3$ different keys, are covered. All other keys from chain 1 are already covered within chain 2.

The other event which lowers the coverage of keys is an loop, but contrary to merges loops are possible regardless of the difference between key and ciphertext

length. A loop happens if an already calculated intermediate point is calcualted again within a single chain. If this happens the coverage of keys is reduced since no new keys are covered from this point on as shown in figure 3.5.

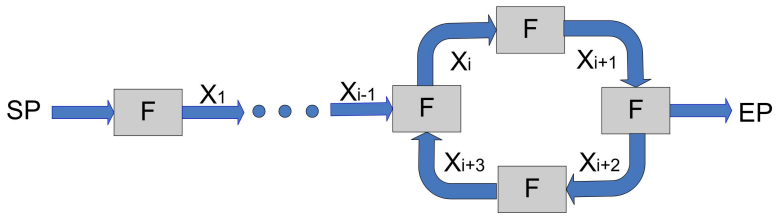

Figure 3.5: Loop within a chain

Until step $i+3$ the generation of the chain is within normal parameters and $i+3$ different keys are covered. During the next step X_{i+4} is calculated which is identical to X_i. Every intermediate point from now is already covered within the chain. Despite the fact that new intermediate points are calculated no new keys are covered, but because the algorithm can't detect the loop it continues until the maximum chainlength $t+1$ is reached.

Multiple Tables

At some point adding a new chain provides little gain to the coverage because the event of a merge or loop is likely. Hellman calculated that this point is somewhere near $N^{\frac{2}{3}}$ for a single table. This is why m and t have their upper bounds set to $N^{\frac{1}{3}}$ each. Because of this s different tables are generated. This is done by a modification of the reduction function R. Up till now the task of R was to simply reduce the resulting ciphertext to the bitlength of k if the one-way function uses different lengths for them. Now, different reduction functions are neccesary for each table. Given the same key and plaintext, R from table 1 needs to generate a different intermediate point than R from table 2. A possible way to achieve this is to simply give each table an integer value from 0 to s. This table index is then added to the ciphertext during the application of R. Now even if two intermediate points from different tables have the same value, the next F produces two different intermediate points since the reduction function R is different. Setting s to $N^{\frac{1}{3}}$ results in the coverage of $m \times t \times s = N$ excluding the loss of coverage through merges and loops and the fact that some keys exist in more than just one table.

3.1.2 Online Phase

In the online phase a ciphertext C' is given that is assumed to be the result of the encryption of P using some key k'. The search for k' is similar to the

table generation. Instead of the chosen key the ciphertext C' is used. First, C' is reduced to the bitlength of the key with the reduction function R. Then this bitstring or X' is compared to all end points stored in the table. If a match occurs a key candidate is already found (with an exceptions which will be adressed later). If X' is not found in the table then the fixed plaintext P is encrypted with X' just like during the precomputation phase. The next X' is again being compared with the table. This is done as often as it takes to find a match or until the number of encryptions reaches the length of the stored chains. If this happens the key was never used during the table generation (excluding key SP). The scheme of the online phase is shown in figure 3.6.

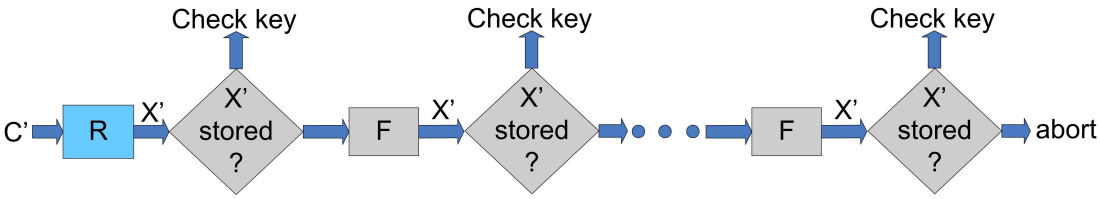

Figure 3.6: Scheme of Hellman's online phase

Now if a match is found after i encryptions a candidate for the key k' can be calculated. This can be done because the start point is stored and the chain in which the match was found can be reproduced until step $t - i$. This step is the point at which the intermediate point, or in this special case the key, was used to produce our current X'. All that is left to do is to check if $E_{k'} = C'$, thus if the key k' is valid. If it is valid the online phase ends, otherwise the procedure continues until a valid key is found or until F was applied as often as $t - 1$ times without finding a valid key. In the latter case the procedure starts with the next table.

False Alarms

A false alarm is an event where a match was found between a stored end point and the ciphertext during the online phase which did not provide a valid key. This can happen if our ciphertext C is reduced to the bitlength of the key. Lets assume that R reduces C by 1 bit. Both $R(E_{k'})$ and $R(E_{k''})$ can produce the same intermediate point even if the resulting ciphertexts of both encryptions are different and different keys were used for the encryption. If both intermediate points are covered then the chance that a false alarm occurs is 50%. Another possibility is that an end point has more than one possible start points, for example in the event of a merge or that a possible start point exists which was not used during the precomputation phase. In either case the stored, therefore wrong, start point is chosen for the chain recomputation resulting in a non-valid key candidate.

3.1.3 Analysis

This section will address the different aspects of Hellman's method like the success rate of the tables as well as the memory usage and the time needed during both the online and precomputation phase. The calculation of the success rate is given by [Hel80].

Success Rate

The success rate of a table is basically the probability that a particular key is covered within all stored chains. Increasing the number of covered keys therefore increases the success rate as well. The number of generated keys depends on the number of different start points, the length of the chains as well as the number of generated tables. The success rate given by Hellman for a single table is denoted as:

$$Success(Hellman) = \frac{1}{N} \cdot \sum_{i=1}^{m} \left(\sum_{j=0}^{t-1} \left(\frac{N - (i \cdot t)}{N} \right)^{j+1} \right)$$

As mentioned before the coverage of a single table is not sufficient enough therefore multiple tables need to be generated for a sufficient success rate. The total calculation for the success rate of multiple Hellman tables is denoted with

$$Success_H = 1 - \left(\Big(1 - Success(Hellman) \Big)^s \right)$$

Memory Usage

Storing the tables of a table lookup needs a huge amount of space. One aspect of a time-memory tradeoff is to reduce the memory usage into a more managable size. Hellman's TMTO generates a table by using m different start points. Storing these equates into $m \cdot storage$, with mem_H denoting the stored bits for each (SP, EP)-tuple. Since s different tables are generated the formula for the memory usage is

$$Memory_H = m \cdot s \cdot mem_H$$

The difference between the table lookup and Hellman's time memory tradeoff is the number of stored tuples. While a table lookup stores N tuples of (SP, EP), Hellman's TMTO only stores $N^{\frac{2}{3}}$ given by $s = m = N^{\frac{1}{3}}$. If our one-way function uses a 56 bit key then the lookup table is ≈ 416.000 times larger than the TMTO table.

Precomputation Time

Generating the tables is done during our leisure time. But nevertheless it takes a lot of time to generate the tables and sometimes time is an important factor in our attack scenario. The precomputation for a single table depends on the number of different start points and the number of encryptions in every chain, therefore $m \cdot t$ encryptions per table. Since s different tables are generated the precomputation complexity for Hellman's TMTO equals

$$Time_H = m \cdot t \cdot s$$

$Time_H$ denotes the number of encryptions which take place during the precomputation phase. To obtain the actual time in seconds the value $Time_H$ is divided by the number of encryptions per second which can be calculated by our device, be it a retail PC or a specialized hardware platform like COPACOBANA.

Online Time

The online time is a synonym for the time our online phase needs to find a key in a worst case scenario. It takes t applications of F to search through a single table since the comparisons are done simultaneous for all chains. Since Hellman's TMTO consists of s different tables the time to search for a key is denoted as

$$Time'_H = t \cdot s$$

Like with the precomputation time, $Time'_H$ is divided by the number of encryptions per second. A problem which arises here is that we still have

$$TAccess_H = t \cdot s$$

table accesses as well. If we can compute one million encryptions per second then we also need the capability to access our table one million times per second. If our table is stored on a hard disk this many accesses seem impossible. A possible solution to this will be presented by the next method: distinguished points.

3.2 Distinguished Points

In practice, the time required to complete the online phase of Hellman's TMTO is dominated by the high number of table accesses. This is due to the fact that these queries result in time consuming random accesses to disk assuming that the precomputed tables are to large to fit into RAM.

The distinguished point (DP) method, introduced by Rivest [D. 82] in 1982 and further analysed by [SRQL02], is a variation of Hellman's approach that addresses this problem. Regarding the considered key chains a distinguished point is a key that fulfills a certain but simple criterion (e.g., the first 20 bits are 0). We call this criterion the *DP-property*. Rivest's idea was to admit only distinguished points as end points of a chain. In this way, during the online phase not every intermediate point can be an end point and must be compared to the table entries but only the points satisfying the DP-property. The DP-property is usually given as a mask which is compared to each intermediate point. The length of this mask, denoted by dp_l, is an important TMTO parameter since it has an effect on several parameters.

The DP method also has the following advantages which will be addressed later that allow to increase the coverage of a table as noted in [BPV98]: Loops and merges can easily be detected.

3.2.1 Precomputation Phase

The basic scheme of the precomputation is identical to Hellman's method. Prior to the table generation the following parameters are chosen: SP and P similar to Hellman's method, the maximum chain length t_{max}, as well as the DP-property given by a bitmask with length dp_l.

The algorithm starts by computing the first $F(SP) = X_1$ of the chain. X_1 is then compared with the DP-property and if they match an end point EP is found. This end point is stored in the table together with the start point SP and the chain length t_{DP}. The chain length is important since it is needed to reconstruct the chain. In Hellman's method every chain has the same length $t + 1$ which is needed during the online phase to search for the key. But for distinguished points each chain has a different length and if this information is not stored then a simple reconstruction is not possible. A possible solution to not storing the chainlength will be presented later but for now we assume that the chain length needs to be stored. If X_1 does not match the DP-property, F is applied until a match is found. To prevent a chain to grow indefinately without ever reaching an end point a maximum chain length of t_{max} is used. If the number of applications of F reaches the chosen upper bound of t_{max} the algorithm stops, discards the

chain and restarts with another SP. The following figure shows the generation of a single chain.

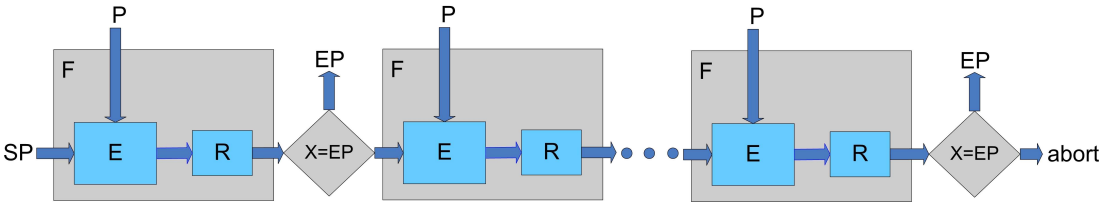

Figure 3.7: Precomputation scheme of a DP table generation

The generated distinguished points tables have a certain characteristic. A single table does not contain any key more than once since double keys are sorted out by loop and merge detection/prevention. Like Hellman's method s different tables, each with m different start points are computed but before a chain is stored it is compared to each chain in the table. If the active chain and a stored chain have the exact same end point then only the longer chain is stored while the shorter one is dicarded. In addition to this a certain characteristic holds true for each stored chain. Each end point is identical in at least dp_l bits, more precisely they all contain the DP-property. These two modifications of Hellman's method allow distinguished points to detected merges and prevent loops.

Merges and Loops

Lets assume that two chains, chain A and chain B both have an identical intermediate point with $X_A = X_B$ but with X_A being reached at step i while X_B is reached in step j and $i < j$. If none of the chains reach an end point, then they both are not stored thus the merge isn't stored either. If they do reach an end point, which will be the same for both chains, then only chain B will be stored since it is longer than chain A. Therefore a merge can't be stored but the number of covered keys is reduced as well since chain A did provide i keys which are not covered by the stored chain B.

Similar to the merges, loops are not stored in the table because the chain either reaches no end point and thus is not stored or the chain reaches an end point and is stored. In this case a loop can't exist because if an end point is reached the first time the chain is stored right away, the generation of the chain stops and no loop can occur.

3.2.2 Online Phase

The online phase of distinguished points is similar to Hellman's method. One starts by computing a chain starting with $R(C)$. The resulting X' is not neces-

sarily compared with the stored table because all stored end points match the DP-property and a comparison is only neccessary if X' matches the DP-property as well. If X' does not match the mask F is applied as often as it takes to either find an X' which contains the mask or until step $t_{max}-1$ is reached. In the latter case a key candidate was not covered within this table and the algorithm resumes with the next table. If an X' is found which matches the DP-property, then the according chain is regenerated. If a match is found at step i the algorithm starts recomputing the chain until step $t_{DP} - i - 1$ is reached just like in Hellman's method. But for distinguished points t_{DP} is not fixed since every chain has an individual chainlength between the boundaries of t_{min} and t_{max}. This is why the chain length of each chain is stored together with the tuple of (SP, EP). Without this information the algorithm could not identify the exact step in which the key candidate was used. If the key candidate is valid the algorithm stops and returns the key, otherwise the algorithm starts searching through the next table. The current table does not contain another key candidate because of the following: The reduced ciphertext $R(C) = X'$ is not present in any other chain throughout the whole table because the distinguished points method generates perfect tables, hence any intermediate point (or possible key candidate) is present only once. Another chain may contain the same X' but either did it merge with the chain which was already checked, looped or the chain was never generated during the precomputation phase. In either case, a further search through the table is not needed.

3.2.3 Analysis

In the analysis of distinguished points a new parameter is used: t_{min} denoted as minimum chain length. This little modification of the original method plays a vital role in the comparison chapter and the need for this parameter will be described throughly in the COPACOBANA chapter. The calculations for the success rate, memory usage and precomputation/online time are given by [SRQL02].

Mask

The mask is the main difference between Hellman's original method and distinguished points. The influence of the mask covers the success rate, memory usage as well as the precomputation and online time. The bitvalue or DP-property of the mask is chosen arbitrarily.
The length of the mask, howerver, should not be chosen arbitrarily because it has an influence on how many chains are stored in each table. A long mask reduces the probability of a match between an intermediate point and the mask, therefore storing fewer but longer chains. A short mask on the other hand will produce many stored chains but each chain provides fewer covered keys because

an end point is found earlier than for a long chain. This directly influences the precomputation time because the computation of a long chain needs more time than for a short chain. The effect on the success rate is limited if the parameters aren't chosen in an extreme manner. If for example a mask length is chosen with $dp_l + 1 =$ key length then only two distinct chains remain. Despite how many different start points are chosen, the maximum coverage will be presented by the maximum coverage of these two chains, more precisely the maximum coverage will be $2 \cdot t_{max}$ at most. This leads to a direct dependancy between the mask length and the memory usage with long masks decreasing and short mask increasing the used storage, based on the same number of start points and chain length for a single table.

Since the mask length influences the number of reached end points it as well has an effect on the coverage, hence the success rate of the corresponding table. A mask which is chosen too long or short reduces the success rate. Short chains do not provide a high enough coverage and too long chains are stored rarely which results in a low coverage as well.

The third influence of the mask is on the number of table accesses. The number of table accesses for Hellman's method is given by t for a single table because each computed X' needs to be compared with the stored table. Distinguished points only need to access the table if X' matches the DP-property. The probability of this match is directly related to the length of the mask. A mask with $dp_l = 1$ results in a probability of 0.5 to reach an end point with any random X', reducing the number of table accesses to $\frac{t_{DP}}{2^1}$. An increase of the mask length therefore reduces the number of table accesses by a factor of 2^{dp_l} which reduces the online time.

A good choice for the mask length always depends on the other parameters as well. A dp_l of 10 may be good for chain lengths between 2^8 and 2^{14} regarding a one-way function with a key length of 40 bit, but it may prove improper for a 56 bit key with chain lengths between 2^{16} and 2^{20}. An approximation of a good mask length always depends on the set objective. A fast online and precomputation phase favors a short chain, while a low memory usage requires a longer chain.

Chainlength

The distinguished points method uses two types of chain lengths. The average chain length L and the chain length after merges L', hence, the situation after the precomputation phase where all chains are stored. At first one could guess

that the chain length after merges is longer than before because only the longer chains are stored with the short ones being discarded. Consider the situation that all generated chains are stored and that merging chains are sorted out after the precomputation. During this "merging phase" many chains will merge with already present long chains because the longest chain covers the most keys and the probability that a chain has an intermediate point which is already within a long chain is higher for a long than for a short chain. After the "merging phase" many long chains will be merged into each other while more short chains remain unmerged. This is due to the fact that their probability to cover an intermediate point which is present in another chain is limited because a short chain only covers few keys which could merge. The stored table will contain few long chains and many short chains, resulting in a shorter average chain length after merge than before.

Success Rate

Incorporating many more parameters than Hellman's methods as well influences the calculations for the success rate. The calculations are more complex since each chain has no fixed length which makes it hard to approximate the coverage by simply using m, t, s. This is increased even further since chains which merge are sorted out, further reducing the coverage. The success rate is denoted as:

$$Pr(DP) = \frac{DP_c}{N}$$

for a single distinguished points table, therefore resulting in the total success rate of:

$$Success_{DP} = 1 - \big(1 - Pr(DP)\big)^s$$

The used parameter DP_c denotes the number of all covered keys in a single table which depends on approximations of the average chain length L of all computed chains, number of start points and the probability that a chain reaches an end point in up to t_{max} steps. The exact calculations can be found in [SRQL02]

Memory Usage

The memory usage of distinguished points depends on the number of generated tables, stored chains and the bits which are needed to store the triple of (SP, EP, t_{DP}). The number of generated tables is already given as s while mem_{DP} denotes the bits which need to be stored for each chain. Calculating the number of stored chains in a single table poses a little problem because m can't be used in this case. Some chains merge or loop and are not stored. These missing chains reduce the memory usage and since $m \cdot s$ is only a upper bound for the number

of stored chains, L' instead of L is used in the following calculation. The number of chains after merges can be approximated by comparing the total number of covered keys with the average chain length after merges resulting in a memory usage of:

$$Memory_{DP} = \frac{DP_c}{L'} \cdot s \times mem_{DP}\ bit$$

with mem_{DP} denoting the stored bits for each triple of (SP, EP, t_{DP}).

Precomputation Time

The precomputation time for distinguished points is slightly different than for Hellman's method. This is due to the fact that a distinguished points chain has no fixed length. Each chain takes an individual amount of time being constructed and it is not always certain that the chain is finally stored in the table. In the latter case $t_{max} - 1$ applications of F were done for nothing. The precomputation time therefore depends on the number of calculations which lead to a stored chain and the ones which don't. For a single chain the average number of calculations is denoted as:

$$\delta = t_{max} \cdot \big(1 - Prob(t_{max})\big) + L \cdot Prob(t_{max})$$

with $Prob(t_{max})$ denoting the probability to reach an end point in up to t_{max} steps. This value is multiplied by the number of generated chains in all tables, hence:

$$Time_{DP} = m \cdot \delta \cdot s$$

defines the total number of calculations during the precomputation phase.

Online Time

The online time for distinguished points is based on the same parameters as Hellman's method. The number of calculations depends on the average chain length after merges are sorted out as well as the total number of generated tables which is defined as:

$$Time'_{DP} = L' \cdot s$$

The number of table accesses depends on the probability that a match between X' and the mask is found. In a worst case scenario each table requires an access while only one access is needed in a best case scenario throughout the whole online phase. The probability to reach an end point in up to t_{max} steps is defined as $Prob(t_{max})$ and the number of table accesses is calculated with:

$$TAccess_{DP} = Prob(t_{max}) \cdot s$$

3.3 Rainbow Table

Rainbow tables were introduced by Oechslin [Oec03] in 2003. He suggested not to use the same R when generating a chain for a single table but a (fixed) sequence of different reduction functions. More precisely, a different R should be applied in each step of the chain computation with $X_i = R_i\big(E_{X_{i-1}}(P)\big)$. The advantage of this modification are fewer merges and a prevention of loops while the number of table accesses decreases as well.

3.3.1 Precomputation Phase

The generation of the rainbow table is nearly identical to Hellman's method with only two minor differences. The first one being the different reduction functions. These can be constructed in the same way as the different reduction functions for each table in Hellman's method or distinguished points. The second difference is the structure of the generated tables. Hellman's method or distinguished points each use m start points and s tables while rainbow tables use only 1 table and m start points but with the difference that $m \cdot s$ for Hellman and DP equals m for rainbow tables resulting in a huge single rainbow table. This can be done because the other methods do not prevent merges, with DP only detecting and not storing them. Adding more chains in a single table therefore increases the chance of merges and loops. Hellman's table would grow linear while the success rate would increase only marginally. The DP table would grow in the same fashion like the success rate but most computed chains wouldn't be stored. In both cases the computation complexity would rise identical to the number of start points. Rainbow tables are different. Loops are prevented and do not reduce the coverage and the event of a merge is unlikely even in a large table. This is why a single rainbow table can contain more chains than a Hellman or DP table. The advantage of this huge single table will be addressed during the online phase since it has no effect on the precomputation. The coverage of the rainbow table is similar to s tables from Hellman's method since they are constructed by using a total of $m \times s$ start points. The construction scheme of a rainbow table is shown in figure 3.8.

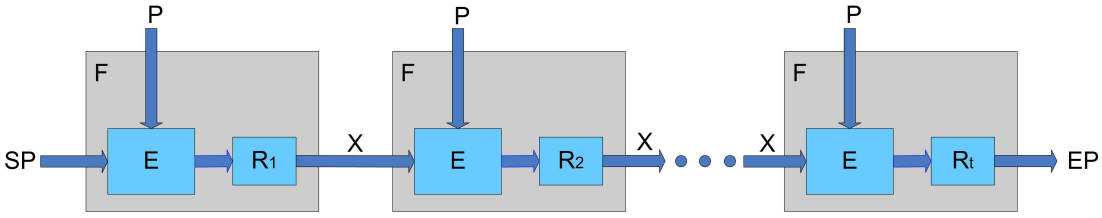

Figure 3.8: Precomputation scheme of a rainbow table

Merges and Loops

The following example will demonstrate that loops can't occur in a rainbow table: The event of a loop occurs if two intermediate points X_i and X_j within a single chain contain the exact same value for both Hellman's method and distinguished points, albeit DP has mechanisms to prevent them from being stored in the table. This is because the next calculated intermediate point X_{j+1} is constructed by the same F function as it was done for X_{i+1}. In rainbow tables X_{j+1} is not contructed in the same way as X_{i+1} because every step uses a different reduction function. Therefore these two intermediate points are different and even if two constructed intermediate points are equal, the predecessors are not and this is why loops can't occur within rainbow tables.
The different reduction functions as well prevent most merges. Let's assume that X_i and X_j are two intermediate points of two different chains. Both of them have the same value therefore a merge would occur in Hellman's method and distinguished points. In rainbow tables a merge only occurs if the two intermediate points were constructed in the exact same step. Only then a merge occurs because only then the used reduction functions are the same. If the two intermediate points are computed in different steps then the two predecessors are different and no merge occurs. If the chance of two chains merging in Hellman's table is defined as g then the chance of a merge in rainbow tables is $\frac{g}{t+1}$, with $t+1$ being the chain length. That is, if two chains would merge in Hellman's table the chance of the same merge in a rainbow table is $\frac{1}{t+1}$ reducing the occurance of megres considerably.

3.3.2 Online Phase

The online phase of rainbow tables slightly differs from the other methods due to the use of different reduction functions. To lookup a key in the table one first computes $R_t(C)$ and compares it to the end points in the table. If no match is found one computes $F_t(R_{t-1}(C))$ and checks for a match, then $F_t(F_{t-1}(R_{t-2}(C)))$ and so on. Thus, this procedure requires in the worst case about $\frac{t^2}{2}$ applications of a step-function and t table accesses. If some EP matches one can retrieve the key candidate similarly to Hellman's method by reconstruction a part of the corresponding chain starting from SP.

3.3.3 Analysis

The analysis of rainbow tables regarding the success rate, precomputation and online time as well as the memory usage is shown below with the calculations for the success rate and online time given by [Oec03].

Success Rate

Like the basic method the probability of success for a rainbow table depends on the number of start points m, tables s and chain length t. The formula to calculate the probability of success is defined as:

$$Success_{RT} = 1 - \Big(1 - \prod_{i=1}^{t} \big(1 - \frac{m_i}{N}\big)\Big)^s$$

with $m_1 = m$ and $m_i = N\Big(1 - e^{-\frac{m_{i-1}}{N}}\Big)$

Memory

The memory usage of rainbow tables is calculated in the same way as for Hellman's method. Rainbow tables therefore have a memory usage of:

$$Memory_{RT} = m \cdot s \cdot mem_{RT}$$

with mem_{RT} denoting the number of stored bits for each (SP, EP) tuple

Precomputation Time

The time needed for the precomputation is identical to Hellman's method as well. Each chain is calculated with t encryptions while each table s contains m different start points. This results in a precomputation time of:

$$Time_{RT} = m \cdot t \cdot s$$

.

Online Time

Rainbow tables need a different approach for the calculation of the online time. This is due to the fact that each step uses a different reduction function R. For Hellman and DP the online time is calculated by multiplying the chain length t with the different tables s. But as mentioned before during the online phase the number of calculations differ for from the other two methods. In the first step only a single application of F is needed while the second step needs 2 applications, the third step 3 and so on. This results in a total number of calculations of

$$Time'_{RT} = t \cdot \frac{t-1}{2}$$

The number of table comparisons during the online phase however are limited to only

$$TAccess_{RT} = t \cdot s$$

with the difference that s for Hellman and DP is a lot larger than for rainbow tables.

4 Special Purpose Hardware

Breaking a (symmetric or asymmetric) cipher is a computationally complex task, even (or especially) if using the aformentioned methods. Considering a security parameter of 2^{56} in case of DES, a normal computer is completely overstrained with this kind of task and this is where special purpose hardware can take over. While a current retail computer can be seen as a "'jack of all trades"' a special purpose hardware is, as it's name implies, a piece of hardware which is designed for a specific task only. In our case this is the computation of DES encryptions. This specialization of just one task provides the possibility to reduce the once too complex and time consuming task of breaking the DES into a reasonable timeframe. Designing these types of special purpose hardware can be done in two different ways. The first is designing the hardware for just one task without the possibility to change or enhance the design without rebuilding it. This approach is done using *application specific integrated circuits* (ASIC) which can be designed only once. The other approach is to use a hardware architecture which can be redesigned over and over again. This gives the possibility to enhance or modify the design to the current task which can be done using field programmable gate arrays (FPGA).

4.1 Field Programmable Gate Array

An FPGA is an integrated circuit (IC) which consists of so called *logic blocks*. These logic blocks can be configured to do certain tasks which range from a simple counter up to a complex task like an encryption function. A single logic block usually consists of a 4 bit lookup table (LUT) and a delay flipflop (DFF). A complex design uses up many of these logic blocks which are placed and routed (connected) during the compilation of the design. Additionally to these logic blocks an FPGA needs some input/output blocks to establish communication between a single FPGA and the bus it is connected to as well as a possibility to temporarily store data. This is done by using block RAM's which vary in size and clock rate depending on the used FPGA architecture. Finally, the FPGA needs to be clocked which is done by a built in digital clock manager (DCM).
The language to design these FPGA's is called a *hardware description language* and one of these is *VHDL*. It is used to design the layout and functionality of an FPGA, but since a introduction to VHDL would go beyond the scope of this

thesis some simple code snipets are presented in appendix xxx
Im Anhang einfügen !

4.2 COPACOBANA

One device using FPGAs is the *Cost-Optimized Parallel Codebreaker* (COPACOBANA) designed by the University of Kiel and Bochum in 2006. It is an FPGA array which uses Xilinx Spartan XC3S1000 FPGAs with the following specifications.

Feature	XC3S1000
System Gates	1 million
Logic Blocks	1920
Distributed RAM	120k bits
Block RAM	432k bits
I/O Bits	391

Instead of placing them on the backplane of the COPACOBANA, 6 FPGAs are placed on a module (DIMM format) as seen in figure 4.1 which can be placed on one of the 20 slots present on the backplane.

Figure 4.1: A single FPGA

This results in a total number of 120 FPGAs which can execute a task simultaneously. These FPGA modules are connected via a 64 bit bus which is controlled by a host module, realised by another FPGA. With the option of placing only some FPGA modules in the COPACOBANA, this design can be adapted to best fit the current task. For example computing only a single chain during the online phase can be done by a single FPGA while multiple FPGAs won't decrease the computation time at all.

The backplane of the COPACOBANA is placed in a box, fit with several fans (they are definitely "'not"' living room capable) which carry out the air heated by the FPGAs depicted in figure 4.2. It is connected via an *universal serial bus* (USB) port to the computer which manages and controls the COPACOBANA.

Figure 4.2: COPACOBANA

Considering the name *"special purpose" hardware*, it means that this piece of hardware is specialized for a specific task. This task was to execute a brute force attack on DES as fast as possible. A successful brute force attack was already done by the *Electronic Frontier Foundation* (EFF) in 1998 with their special purpose hardware called *Deep Crack*. The costs for the Deep Crack were around 250.000 US dollars and it completed the brute force attack in about 56 hours. Compared to this, 2006 the COPACOBANA cost around 10.000 dollars and completed a brute force attack in about 7 days.
But since the COPACOBANA is specialized for a high computation speed some other fields which are important for a time memory tradeoff were neglected.

4.2.1 Hardware Specifications

The internal communication, managed by the controlling FPGA, can handle up to 24 MHz on a 64 bit bus while the external bus is currently connected via an USB 1.0 port. Currently, an upgrade to an ethernet port with up to 100 mbit is in the final stage of testing, of which approximately only 24 mbit can be used since the limits of the controller. Therefore all following calculations are made with the assumed bandwidth of 24 mbit. The next upgrade is planned to be a full 1 gbit ethernet port but the completion of this upgrade lies far beyond the time schedule of this thesis which is why it won't be considered in the upcoming

chapters.

The current design of the COPACOBANA is the brute force scenario and each of the 120 FPGAs are designed with 4 pipelined DES implementations. This translates in a total of 480 DES encryptions which are made simultaneous. Considering the implementation of the time memory tradeoff, the chances are that because of the higher logic requirements, hence a higher space requirement, this number needs to be reduced to 3 DES implementations per FPGA. But since the current DES implementation uses 21 pipeline stages to increase the speed, only 16 stages are really needed. This could reduce the required space to the extend of being able to once again implement 4 DES on one FPGA.
Currently the brute force design allows a clock rate of 136 MHz. The 136 MHz of an FPGA, with a total of 480 FPGAs situated in the COPACOBANA, translates into $\frac{(136\times10^6)\times480}{sec} \approx 2^{35.9}$ encryptions per second.

The high number of calculations per second and the small bandwidth provided results in the following problem. The maximum external bitrate is 24×10^6 bit per second with $2^{35.9} \approx 64 \times 10^9$ DES encryptions per second. If 112 bit for every pair of (SP, EP) is assumed, we get a difference of $\frac{43.2\times10^9\times112bit}{24\times10^6bit} \approx 2^{17.6}$.
This value represents the number of encryptions which need to be computed before the result, the (SP, EP) pair, is being transmitted through the external bus. This is the first, and most important, boundary given by the current design of the COPACOBANA for a time memory tradeoff. It translates in the minimum chainlength which needs to be computed for each of the aformentioned methods. This value does not take the neccesary communications into account which are needed to control the COPACOBANA, therefore the minimum chainlength is set to 2^{18}. Further testing during the implementation will show if this value needs to be increased or maybe can even be decreased.

4.2.2 Scenarios

A time memory tradeoff consists of two phases, the precomputation and the online phase which results in only three possible scenarios. Using the COPACOBANA only for the precomputation, only for the online phase or for both. The online phase only approach is not feasible at all because of the high data throughput we need during this phase which leaves us with two basic scenarios in which the COPACOBANA can be used.

Precomputation Only

In this scenario all computations during the precomputation are done by the COPACOBANA while the computer only manages sending new keys while receiving already processed chains to store them into the table. The online phase in this scenario then needs to be executed by a computer or maybe even another specialized hardware machine with a higher data throughput than the COPACOBANA. The following list summarizes the different pros and cons of this approach.

- The online phase can be done from any computer. This means that literally anyone who has access to the table can execute an attack on DES.
- The computation power of a single computer is adequate to execute the online phase, even if it is not as fast as with the COPACOBANA.
- Splitting the table in several parts and spreading them on different computers decreases the time for the online phase.
- After the precomputation phase the COPACOBANA can be used to generate additional tables to increase the success rate independant from whether a online attack is executed or not. Alternatively a new time memory tradeoff precomputation with a new fixed plaintext can be started.
- A fast implementation of DES is needed for the computer. A slow implementation could easily increase the time for the online phase beyond an hour, even if multiple computers are used.

The scheme shown in figure 4.3 illustrates the basic task of the COPACOBANA in this scenario. The inputs of *Configuration* and *Start Parameters* are received from the COPACOBANA with the *Data Stream to Storage* output being sent to the controlling computer.
The *Configuration* handles the long term parameters like the chainlength, plaintext, number of start points and generated tables as well as neccesary parameters for the Distinguished Points or Rainbow Tables method. These values could all be hard coded into the design of the COPACOBANA but for each new time memory tradeoff a modification of the design would be neccesary. Therefore, these parameters are sent during the precomputation which not only takes up a portion of the bandwidth but as well increases the design, since storing mechanisms for these parameters need to be implemented as well. But this leaves most of the control with the user like changeing several start points or computing another table without modifying the source code as well as saving the hassle to recompile the whole design for each little modification.
The *Control Parameters* cover the currently processed start points as well as the control mechanism for the COPACOBANA like start, stop or reset commands during the precomputation. The output of our precomputation are the (SP, EP) tuples which are sent via the *Data Stream to Storage* to our time memory tradeoff

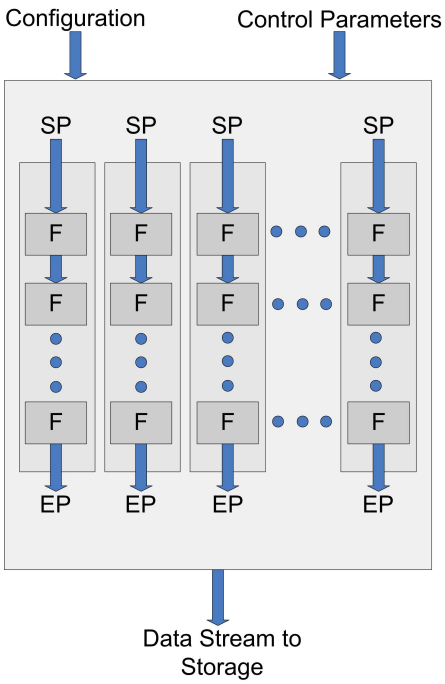

Figure 4.3: COPACOBANA precomputation scheme

table.
The limiting factor in this scenario is solely presented by the bandwidth of the COPACOBANA which affects the speed with which the controlling computer can communicate with the COPACOBANA, or more precisely, how many (SP, EP) tuples can be sent to the table each a second.

Precomputation and Online Phase

The second scenario uses the COPACOBANA for both the precomputation and online phase. This implementation method would be similar to a black box system using the COPACOBANA for all neccesary computations with but only one exception. The table searches need to be done by a computer because the tables are stored in an external storage. The different aspects of this type of implementation are listed below.

- Using the COPACOBANA for both the precomputation and online phase creates one big problem which was already addressed earlier, the bandwidth bottleneck. During the precomputation this bottleneck limits the COPACOBANA to a minimum chainlength. Now this bottleneck limits the number of table comparisons as well. In Hellman's case a single encryption equals one table comparisons, while for Rainbow Tables and Distinguished Points multiple encryptions equal one table access as explained in chapter

3. This fact concludes that Hellman's method is not usable in this scenario until the bandwidth of the COPACOBANA can trasfer at least $2^{35.9}$ endpoints each second which translates into $2^{35.9} \times 7$ byte ≈ 418 gigabyte per second. Even if the other two methods are used, the probability of fully utilizing the COPACOBANA's computation power in this scenario depends on the chosen parameters during the precomputation.

- Implementing both the precomputation and online phase on the COPACOBANA increases the design, hence using up more logic blocks. This will reduce the performance of the design since the compiler has less room for placing the different logic blocks. A possible solution is to implement two distinct designs, each for one phase and load it into the COPACOBANA when needed. The drawback of this solution is that when a improvement of the design is made, it most certainly needs to be done in both implementations, precomputation and online.
- The most obvious drawback in this scenario is the fact that the COPACOBANA needs to be used for the online phase. Therefore the possibility to simply distribute the attack is not given, as is the possibility to simply distribute the precomputed tables to institutions which don't own a COPACOBANA. Additionally, the precomputation and online phase need to be executed sequentially unless a second COPACOBANA is used. Hence, further increasing the tables by additional precomputations is not possible while the COPACOBANA is used for online attacks.

The tasks for the COPACOBANA in this scenario are shown in figure 4.4.

Additionaly to the precomputation tasks the COPACOBANA as well needs to execute the computations needed for the table search. That is, the reduced ciphertext is processed until the requirement for the used method is met and then compared with the table. For Distinguished Points this is the match with the DP-property, while for Rainbow Tables it is the variable number of step functions F which are needed for recomputing the current position within the chain with Hellman accessing the table after each computation.

As mentioned above the limiting factor is the bandwidth which leaves us with only two options. Increasing the bandwidth of the COPACOBANA, thus increasing the costs or using either Rainbow Tables or Distinguished Points to reduce the number of table accesses. Since the modification of the COPACOBANA is too large a task for this thesis this leaves us with only one option. If a COPACOBANA only implementation is to be done then the only viable option is to use either Rainbow Tables or Distinguished Points.

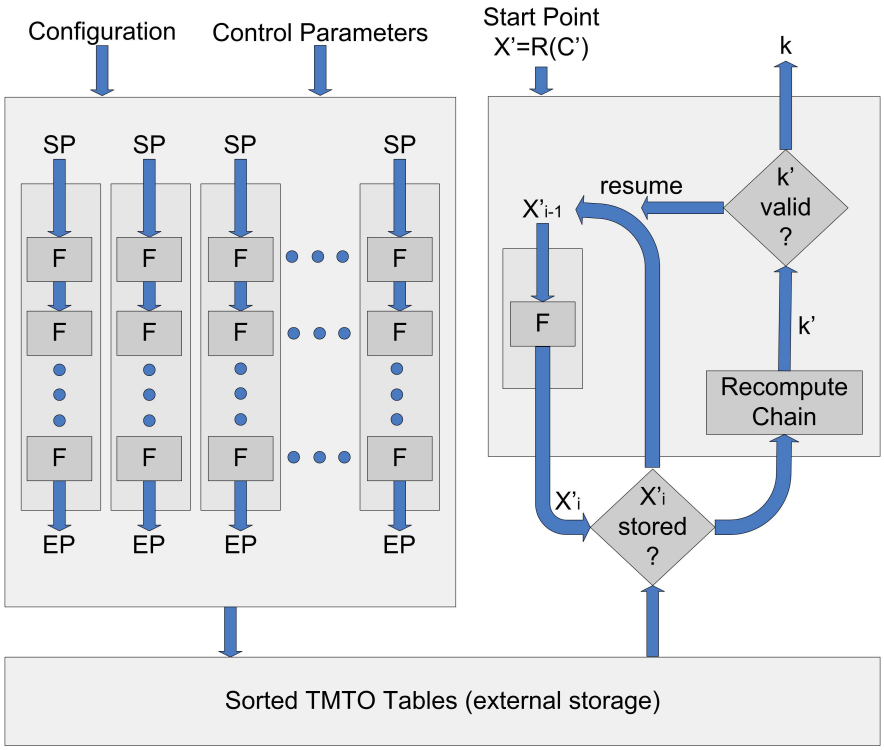

Figure 4.4: COPACOBANA only implementation scheme

5 TMTO Selection

The last chapters introduced the three time memory tradeoff methods, the technical informations of the COPACOBANA and different aspects of the two possible implementation scenarios. The task of this chapter is to compare the three methods with each other in regard to the boundaries which are given by the hardware structure of the COPACOBANA.

Important note: The brute force implementation of the COPACOBANA can compute up to $2^{35.9}$ DES encryptions per second while running with 120 MHz. Reaching the same number of encryptions for the TMTO implementation is unlikely because control mechanisms for the time memory tradeoff computations are needed as well as an increased logic effort compared to a brute force approach. This will slow down the implementation and therefore reducing the maximum number of encryptions per second. Despite this fact the comparisons in this chapter are done with $2^{35.9}$ encryptions per second. The online and precomputation times will not be 100 % accurate for "our" TMTO implementation therefore we need to keep in mind that the computation times might be a bit higher than presented in the following comparisons. Since all comparisons are done with the value of $2^{35.9}$ encryptions for all methods, the results should be significant enough for determining a suited method. Accurate values will be present after the implementation of the chosen time memory tradeoff.

5.1 TMTO Objective

The main objective of a time memory tradeoff is to provide a table with a high probability of covering a certain key. A perfect TMTO table would provide a success rate of 100% similar to a table lookup table. Due to the problems of TMTO tables covering a single key multiple times, the effort to generate such a table is unreasonably high compared to the gain of a perfect table. A reasonable success rate depends on the environment it is used in so in this thesis we want to achive a success rate of at least 80 %. Hence, 4 out of 5 keys are being found which is a decent probability of success.
Another boundary which is neccessary to pick is the allocation of memory for the generated tables. Generating a table which takes up hundrets of terabyte is difficult to manage by a normal computer, let alone the fact that this huge amount of

data needs to be sorted after the precomputation. Considering the retail prices for hard disks, as well as the number of hard disks current motherboards can manage, an upper bound of about 2 terabyte for the TMTO tables should be feasible. As mentioned in chapter 2, the computer controlling the COPACOBANA has a hard disk capacity of 2.5 terabyte which is split into 5 hard disks. The remaining $\approx$ 500 gigabyte provide enough space for the operating system and additional programs.
To sum it up, our time memory tradeoff needs to achive a success rate of at least 80% while taking up at most 2 terabyte of memory. The next section introduces ways to further reduce the memory requirements of time memory tradeoffs, which results in the possibility of storing even more TMTO data (chains) within 2 terabyte.

5.2 Parameter Optimization

Memory capacity is not a big problem as it was a couple of years ago because harddrives are already reaching or surpassing the 1 terabyte barrier. But it still is preferable to have a smaller table than a large one since access and search times in a large database take more time than in a small one. This section introduces ideas to reduce the memory usage of the three discussed methods

5.2.1 Start Points

One design criterion of DES is that the output of a single encryption is pseudorandom, hence after applying F once, the resulting intermediate point is pseudorandom as well. This results in a pseudorandom generation of our chains, regardless of how the startpoints are chosen. Because of this we are free to choose our startpoints in any possible way.
During the precomputation, m different startpoints are used to generate a single table. One way to choose them is by using a simple bitcounter which starts at the bitvalue of 0, and ends at $m-1$. The neccessary bits to store the m different startpoints are bounded by $\lceil log_2(m) \rceil$. This reduces our memory usage from $m \cdot 56$ to $m \cdot \lceil log_2(m) \rceil$ with $m \cdot 56$ bits for storing the endpoints. In addition only $\lceil log_2(m) \rceil$ bits for each startpoint need to be sent from the controlling computer to the COPACOBANA which has an positive effect on the bandwidth usage. This memory reduction can be used for any of the three discussed methods because each of them needs to store startpoints if the table is to be sorted by the endpoints.

The next idea to reduce the memory usage is only viable if either Hellman's method or Rainbow Tables are used. If the table is sorted by the startpoints

while using the binary counter starting at 0 and ending at $m - 1$ storing the startpoints is no longer neccessary. This reduces the memory usage to 0 regarding startpoints with only $m \cdot 56$ bits for storing the endpoints. The negative side effect of this solution is a table which can no longer be sorted by the endpoints which increases the search times during the online phase considerably.
This solution is only partially suited for Distinguished Points because of the merge and loop prevention mechanisms. If two chains merge during the precomputation, only the longer chain is kept. Therefore, the corresponding startpoint is not stored either which results in "holes" in the list of used startpoints. These need to be filled by endpoint placeholders which increase the memory usage. Using this solution for small Distinguished Points tables might be viable, but the more startpoints are used the more chains are discarded resulting in a reduced efficiency of this solution for Distinguished Points.

5.2.2 End Points

During the online phase, the calculated X' are compared with the stored endpoints in each generated table. An optimisation for the memory usage of endpoints can be done considering the Distinguished Points method. Each computed endpoint holds true for the DP-property. The dp_l bits of the mask stay the same for each endpoint in a single table. Therefore these bits don't need to be stored for all endpoints, but only once for each table. This reduces the memory usage from $stored.endpoints \cdot 56$ to $stored.endpoints \cdot (56 - dp_l)$.

This solution can be used for the remaining two methods as well, but at a cost. If the endpoints are reduced by discarding a single bit reducing them to a length of 55 bits, the complexity during the online phase rises. Now each computed X' results in two possible chains which both need to be checked. Therefore, each discarded bit increases the computation complexity during the online phase by a factor of 2 .

Another idea for Distinguished Points is to disregard the length of the chain which is stored for every entry in the table. By saving the bitlength of this entry the computation complexity rises by one.
Normally, during the online phase the computation of the candidate chain ends at $t_{DP} - i - 1$, with the computed intermediate point being the key candidate. Additionally, $t_{DP} - i$ is the position of the intermediate point which matches the reduced ciphertext C. During the modified online phase the chain needs to be recomputed until the current computed intermediate point matches the reduced ciphertext while storing the predecessing intermediate point as well. This means

that if a match is found, the stored intermediate point is the key candidate we're searching for which increases the computation complexity by just one additional application of F.

5.3 Comparison

The comparison of the three methods is based on the already known constraints mentioned throughout the last chapters. First we do have a minimum chainlength of 2^{18}, resulting from the hardware specifications of the COPACOBANA. Second is the memory usage which is not allowed to exceed 2 terabyte of data and third, the success rate of the generated tables which needs to be at least 80%. These three constraints affect all parameters of the time memory tradeoff methods. There is one additional constraint which is not given by the hardware itself but the implemented design. The brute force design incorporates 4 DES engines in each fpga, with a total of 120 fpgas resulting in 480 DES engines which run simultaneously. The aim for the TMTO design is to use the same number of DES engines as well. This affects the minimum number of processed startpoints resulting in the lower bound of $480 \approx 2^9$ startpoints in each table. Next is a short reminder of what each parameter influences.

- Chainlength: The range of the chainlength is lower bounded to 2^{18}, with a maximum of 2^{56}. It affects the success rate, precomputation and online phase time.
- Tables: Interdependancy with the number of Startpoints. Effects memory usage, success rate, precomputation and online phase time.
- Startpoints: There is an interdependency with the number of generated tables which is upper bounded by the allowed memory usage of 2^{43} bit and lower bounded by 2^9. Further including the chainlength results in the lower bound for the success rate of 80%. Effects the memory usage, success rate and precomputation time.

For each of the three methods two figures are shown. The first figure shows all possible choices for the number of startpoints and tables regarding the memory constraint of 2 terabyte. This curve (red) can only be affected (increasing or decreasing the number of possible choices for m and s) if the memory constraint is changed as well. The second figure additionally shows the possible choices in regard to the success rate of 80 %. Modifying this curve (green) can be done in two ways. Either change the desired success rate or the chainlength. The following figure 5.1 illustrates this exemplarily.

The area which lies below the red curve and above the green curve provides eligible parameter pairs which all hold true for our constraints. This method allows us to find a set of possible parameters if a memory and success rate constraint

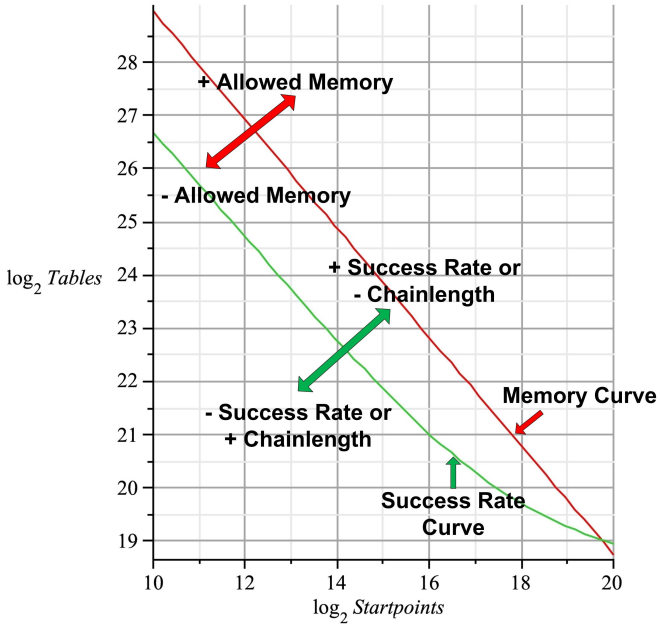

Figure 5.1: Example for Memory and Success Rate Parameter Choices

is given, with the option to widen or shorten the gap between the two curves by either increasing or decreasing the chainlength. This as well gives us the option to directly influence both the precomputation and online time since both values partially depend on the chainlength. The determination of possible parameters for all three methods is done in the following.

5.3.1 Hellman

Hellman suggests in *zitieren* a choice of $2^{\frac{56}{3}}$ for each the number of startpoints, chainlength and tables which only provides a success rate of 55% with a memory usage of 1.5 terabyte. This result is not sufficient enough for our needs. Considering the available formulas given for the Hellman TMTO the first, and easier, task is to find parameters which meet the memory constraint of 2 terabyte. The formula for the memory is as follows:

$$Memory_H = m \cdot s \cdot mem_H$$

with $mem_H = 56 + log_2(m)$ due to the possible parameter optimization on the startpoints presented in the last chapter. Given the 2 terabyte of memory this translates in the following equation.

$$2^{44} = m \cdot s \cdot 56 + log_2(m)$$

which resolves into

$$s = \frac{2^{44}}{m \cdot \big(56 + log_2(m)\big)}$$

Applying this equation with $m = 2^9 \dots 2^{36}$ provides us with the corresponding maximum number of tables allowed without transgressing our memory constraint. Figure 5.2 shows all possible combinations of m, s.

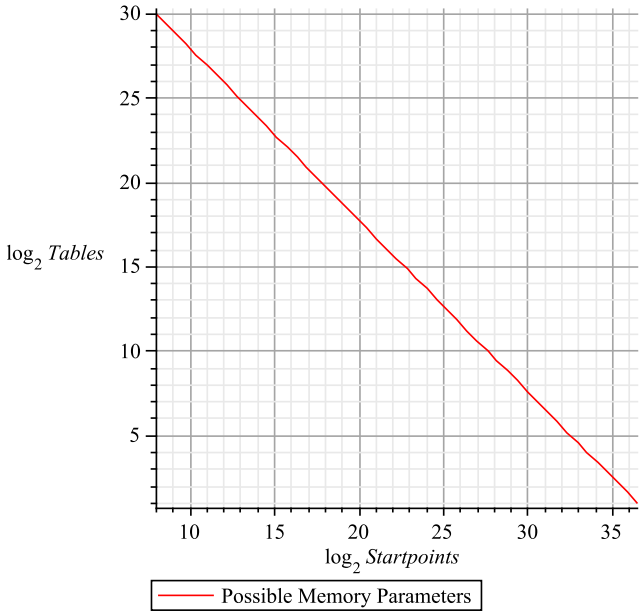

Figure 5.2: Hellman Parameters Part 1

All parameter choices for s and m which lie below the red line are eligible but only few of them actually satisfy our second constraint, the success rate of at least 80% which is given as:

$$Success_H = 1 - \left(\Big(1 - Success(Hellman)\Big)^s\right)$$

with

$$Success(Hellman) = \frac{1}{N} \cdot \sum_{i=1}^{m} \left(\sum_{j=0}^{t-1} \left(\frac{N - (i \cdot t)}{N} \right)^{j+1} \right)$$

Considering the success rate constraint, this translates into

$$s = log_{\left(1 - \left(\frac{1}{N} \cdot \sum_{i=1}^{m} \left(\sum_{j=0}^{t-1} \left(\frac{N-(i \cdot t)}{N}\right)^{j+1}\right)\right)\right)}(0.2)$$

and the area above the green line in figure 5.3 represents possible choices for s and m regarding the constraint of the success rate.

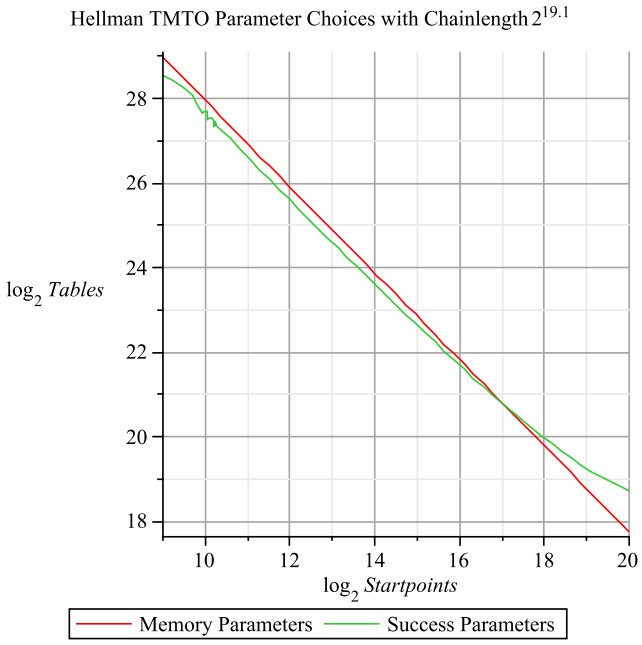

Figure 5.3: Hellman Parameters Part 2

The area which is upper bounded by the red line and lower bounded by the green line represents our possible choices for Hellmans method. Due to the fact that the formula for the success rate given by Hellman is fastidious to calculate for all m, the equivalent form given by $zitierenDPpaper, standaert$ is used. The chainlength of $2^{19.1}$ was not chosen randomly but empirically after trying out different lower values. The short result of this experiment was that a chainlength in the range of $\leq 2^{18.8}$ provides us with no valid parameters to satisfy both the memory and success rate constraint. Only after transgressing the length of 2^{19} possible choices for the number of start points were given.

5.3.2 Distinguished Points

Distinguished Points were described in chapter 3 and considering their memory usage it is hard to predict how many chains are being discarded during the pre-computation process. Therefore the calculations are done while assuming that all computed chains are actually stored. Like with Hellman the bits needed to store the startpoints can be reduced, but as described in the last chapter an additional option to reduce the endpoints is available as well. The memory usage for DP is denoted as:

$$Memory_{DP} = \frac{DP_c}{L'} \cdot s \times mem_{DP}\ bit$$

with $mem_{DP} = 56 - dp_l + log_2(m)$. Similar to Hellman this translates into

$$s = \frac{2^{44}}{m \cdot \left(56 - dp_l + log_2(m)\right)}$$

which results in the curve shown in figure 5.4

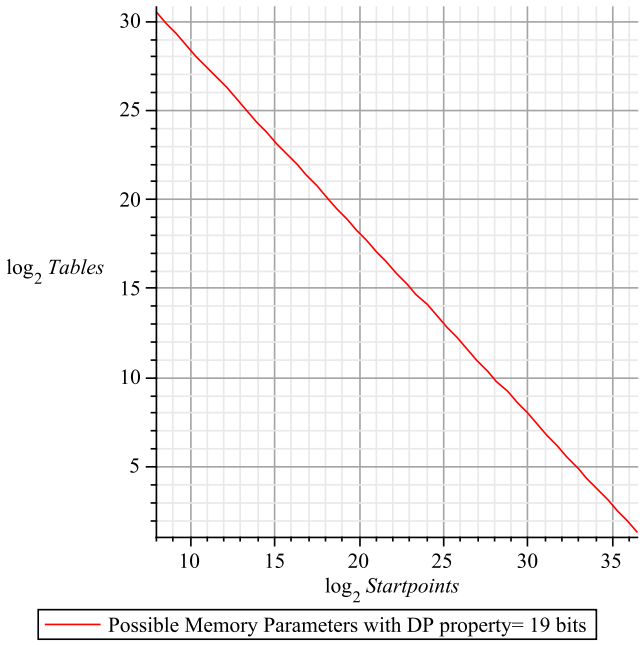

Figure 5.4: Distinguished Points Parameters Part 1

Compared to Hellman, the curve for the possible parameters provides a better range of choices. This is because the memory usage of DP, compared to Hellman, is lower if the same parameters are used since some bits are saved on storing the endpoints. In addition to this, the real, hence not approximated by formulas, curve will even be lying a bit higher due to the fact of discarded chains. The higher curve as well provides DP a higher margin in parameter choices if the curve for the success rate is identical to the one of Hellman.
The success rate for DP is given by:

$$Success_{DP} = 1 - \left(1 - Pr(DP)\right)^s$$

with

$$Pr(DP) = \frac{DP_c}{N}$$

and DP_c representing the number of distinct keys covered in a single table. This translates into the following equation

$$s = log_{\left(1-(\frac{DP_c}{N})\right)}(0.2)$$

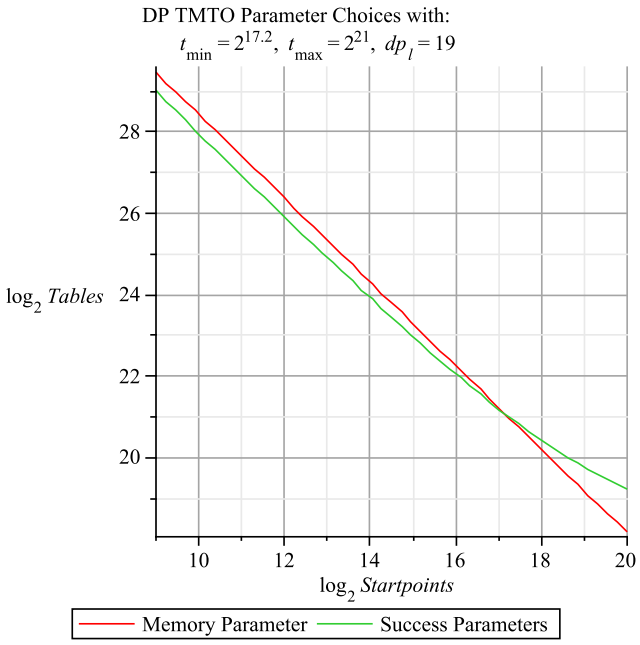

Figure 5.5: Distinguished Points Parameters Part 2

with the results shown in figure 5.5.

The minimum and maximum chainlengths of $t_{min} = 2^{17.4}$ and $t_{max} = 2^{21.2}$ result in an average chainlength of $2^{18.6}$. This lower chainlength will have an positive effect on the online phase compared to Hellman which is explained more detailed in the next chapter.

5.3.3 Rainbow Tables

The last of the methods, Rainbow Tables, provides identical parameter choices regarding the memory constraint as Hellmans method. The memory formula of:

$$Memory_{RT} = m \cdot s \cdot mem_{RT}$$

denoting the memory usage translates into

$$s = \frac{2^{44}}{m \cdot \left(56 + log_2(m)\right)}$$

with the given memory constraint of 2 terabyte. The resulting curve is shown in figure 5.6

The success rate for Rainbow Tables is hard to present in a similar way as for Hellman or Distinguished Points because of the recursion used in the calculation

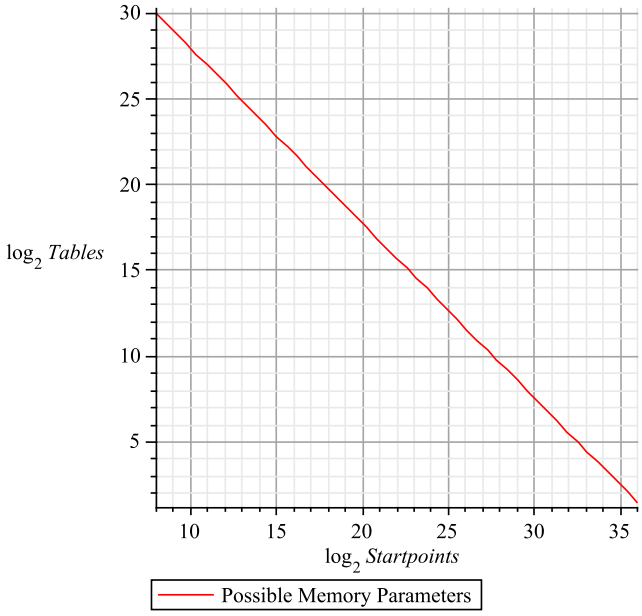

Figure 5.6: Rainbow Parameters Part 1

of the success rate. Calculating only few sets of parameters is still possible, but plotting the curve for values starting at 2^9 and ending at 2^{36} is not possible within a reasonable amount of time. This is why we use a different approach to search for possible parameters satisfying the set constraints.
Rainbow Tables excel quite good even if few tables are used. Therefore we increase the number of tables step by step beginning at $s = 1$ with the corresponding number of startpoints we achieved in 5.6. The chainlength starts at a value of 2^{19} and is increased for each set until the success rate of 80% is reached. The results are shown in the following table.

Tables	Startpoints	Chainlength
1	$2^{37.43}$	$2^{19.89}$
2	$2^{36.45}$	$2^{19.55}$
3	$2^{35.88}$	$2^{19.42}$
4	$2^{35.47}$	$2^{19.36}$
5	$2^{35.16}$	$2^{19.31}$
6	$2^{34.91}$	$2^{19.29}$

Further increasing the tables results in a shorter precomputation phase but with an increased online time. One of our goals is to provide a reasonable precomputation time with a short online phase, hence there is no need to further increase the tables because of the increase of the online time.

5.3.4 Results

The figures 5.3, 5.5 and the above table provide a range of possible parameters which hold true for our constraints. The next task is to choose an eligible pair of m, s for each method. The first limitation is made considering the number of generated tables. Choosing a non-integer value results in the chance that for example 156.34 tables need to be generated which is quite a circumstantial task regarding the source code. This limitation excludes a lot of possible m, s pairs. The remaining pairs are further limited by the following fact. The number of calculations during the online phase of each method depends on the number of generated tables and the chainlength. If a choice can be made between two pairs, with one having a high number of tables and a low number of startpoints and the other one having a low number of tables and a high number of startpoints, the latter one is favored. Hence choosing a m, s pair with a high number of startpoints provides a low online time if the chainlength stays the same.
With this we have a way of choosing a suited pair which is located at the intersection of the memory and the success rate curve for Hellman and Distinguished Points. The choice for Rainbow Tables is made with regard to the resulting precomputation and online time. The next table provides the choices for the three methods.

Method	Startpoints	Tables	Chainlength
Hellman	$2^{17.171}$	$2^{20.635}$	$2^{19.1}$
Distinguished Points	$2^{17.125}$	$2^{21.093}$	$t_{min} = 2^{17.2}$, $t_{max} = 2^{21}$, $L' = 2^{18.7}$
Rainbow Tables	$2^{35.16}$	5	$2^{19.31}$

These parameter choices hold true for our memory and success rate constraint and are used for evaluating the best suited method in the final part of this chapter.

5.4 Final Choice

The chosen parameters in the above table influence both the controlling computer with the attached storage medium as well as the COPACOBANA. The first choice which needs to be made is if the COPACOBANA is used only for the precomputation phase or for both phases.
The next table compares both scenarios regarding their required communication with the storage medium, which equals $t \cdot s$ for Hellman and Rainbow Tables and s for Distinguished Points. In addition the number of computations during the online phase are listed as well, with Distinguished Points using L' for the chainlength since this parameter represents the average number of computations that are executed if no match is found.

Method	Number of Communications	Computations (Online)
Hellman	$2^{39.735}$	$2^{39.735}$
Distinguished Points	$2^{21.093}$	$2^{39.793}$
Rainbow Tables	$2^{21.632}$	$2^{39.941}$

The results of this table are used in the following sections, choosing one of the two possible COPACOBANA scenarios.

5.4.1 Online Phase with COPACOBANA

Using the COPACOBANA for the online phase provides a lot of computation power, but as well results in a lot of communication effort. The number of communications with the storage medium equals the number of communications between the COPACOBANA and the controlling computer since each compared X' needs to be sent.
A single communication equals 1 polling and 1 transfer to the controlling computer with each communication taking up 64 clock cycles. Given the $2^{24.5}$ clock cycles each second, provided by the USB port, this translates into the following communication times between the COPACOBANA and the controlling computer calculated with the formula of $\frac{Number.of.Communications \cdot 2 \cdot 64}{2^{24.5}}$ seconds:

Method	Communication Time
Hellman	$2^{22.235} \approx 56$ days
Distinguished Points	$2^{3.593} \approx 12$ seconds
Rainbow Tables	$2^{4.132} \approx 16$ seconds

These communication delays are related to the COPACOBANA side only, with the controlling computer adding delay times for the USB interface and internal communications as well. These times vary from computer to computer and as well depend on the program which is written for the communication. Since they are too hard to calculate or even approximate they are left out in the calculations. The COPACOBANA brute force design provides us with $2^{35.9}$ possible encryptions per second and the next table shows the time neccessary for the computations during the online phase.

Method	Computation Time
Hellman	$2^{3.835} \approx 15$ seconds
Distinguished Points	$2^{3.893} \approx 15$ seconds
Rainbow Tables	$2^{4.041} \approx 16$ seconds

Because the communications and computations are not done in parallel both the communication and computation times are added resulting in an online phase

using up:
57 days for Hellman, 26 seconds for Distinguished Points and 32 seconds for Rainbow Tables.
Considering the above times, Hellman's method is not suited for an online phase which uses the COPACOBANA. This is due to the fact that both Rainbow Tables and Distinguished Points provide mechanism to reduce the number of table accesses, hence the number of communications. Hellman on the other hand has no mechanism which reduces the number of table accesses, which is why the the communication time is multiple times higher than for the other two methods.

5.4.2 Online Phase without COPACOBANA

If the online phase is handled by the controlling computer all computations need to be done by it as well. Considering an efficient DES implementation on our computer, about 2^{22} encryptions can be done per second. The following table shows the time neccessary for the online phase if the controlling computer is used instead of the COPACOBANA.

Method	Computation Time
Hellman	$2^{17.735} \approx 61$ hours
Distinguished Points	$2^{17.793} \approx 63$ hours
Rainbow Tables	$2^{17.941} \approx 69$ hours

The above times are in a glaring contrast to the ones provided if the COPACOBANA is used, regarding Rainbow Tables and Distinguished Points. Considering only the recent findings a design which uses the COPACOBANA for the online phase seems more promising.

5.4.3 Table Access & Search Times

In addition to the communication and computation time, the time for accessing the storage medium needs to be considered as well. Our storage medium consists of serial ATA hard disks which provide an average access time of ≈ 8 ms. The average access times consider the fact that a data is placed randomly on the disk. This will not happen if the following facts hold true:

- The stored tables are sorted. Accessing data consecutively is done faster which provides an access time near to the best case access time of ≈ 0.8 ms for track to track.
- The stored data is not fragmented. Accessing consecutive data results in reading tracks consecutively as well.

If the above holds true the access time can be reduced considerably, but the actual access times depend on additional factors which can't be evaluated in the course of this work. Since they should be ranging somewhere between 8 and 0.8 ms an approximated access time of 4 ms is used in the following computations. Furthermore a single search needs $log_2(N) = 56$ steps searching the currently processed X' with a table. The resulting search times are shown in the table below which are calculated in the following way: $\frac{Number.of.Table.Accesses \cdot 4 \cdot 56}{2^{10}}$ seconds.

Method	Access & Search Time
Hellman	$2^{37.542} \approx 6328$ years
Distinguished Points	$2^{18.9} \approx 136$ hours
Rainbow Tables	$2^{19.439} \approx 197$ hours

The naive approach for the access and search takes too long, therefore an optimisation is neccessary.Our tables have a combined size of 2 terabyte and the storage medium can transfer about 60 MB per second in average considering data being stored between the rim and the inside of a disk.
A RAM module of our aforementioned controlling computer has an access time of 12 ns. If a part of the table, which is to be searched, is loaded into RAM a single search wouldn't take $4 \cdot 10^{-3} \cdot 56 = 224$ ms but $12 \cdot 10^{-9} \cdot 56 = 0.000672$ ms. Search times for Hellman would be reduced to 170 hours, Distinguished Points to 1.5 seconds and Rainbow Tables to 2.1 seconds. In addition, loading the 2 terabyte into RAM takes about $\frac{2 \cdot 10^{12}}{60 \cdot 10^6} \approx 10$ hours.
The problems which remain are to know which part of the table is currently needed and that each part is allowed to be loaded into RAM only once. This can be solved with the following modification of our online phase. The problem with our current accesses is that while our tables are sorted, the computed X' are pseudorandom. Instead of computing a single X' and then compare it to the table all X' for a single table are computed and then sorted. Now the list of our sorted X' is processed with the table being searched from bottom to top without back jumps. Splitting up a single table into several parts, with each of them fitting into our RAM memory, allows us to calculate with the following access and search times.

Method	Access & Search Time
Hellman	180 hours
Distinguished Points	10 hours
Rainbow Tables	10 hours

5.4.4 Scenario Choice

Considering the current findings, Hellman's method takes too long to execute in both scenarios which leaves us with the choices of either Distinguished Points or Rainbow Tables and either using the COPACOBANA for the precomputation or for both phases. The following table lists the performance of both methods in each scenario with S1 being the precomputation only scenario and S2 being the precomputation and online phase scenario for the COPACOBANA.

Method	S1: Computation	S2: Comp. + Comm.	Search
Distinguished Points	63 hours	27 seconds	10 hours
Rainbow Tables	69 hours	32 seconds	10 hours

Comparing the search time with the time neccessary for the computations (and communications) shows, that the computation power of the COPACOBANA is not needed for the online phase. As mentioned in chapter 4 multiple computers can be used to sped up the computation process during the online phase. In our case six to seven computers (or a 6 to 7 times faster DES implementation) are needed to match the computation time with the search time. Hence, the need for the COPACOBANA is very limited especially if comparing the costs of a COPACOBANA with seven retail computers. In reference to the options mentioned in chapter 4 regarding the precomputation only scenario and the above mentioned facts, the conclusion is that a precomputation only implementation is prefered for our TMTO design.

5.4.5 Precomputation Phase

The last choice before the implementation is whether Distinguished Points or Rainbow Tables are to be used for our TMTO design. Since the time for the online phase of both methods are alike, a closer look on the precomputation phase might show some differences. The precomputation is calculated with $\frac{m \cdot t \cdot s}{2^{35.9}}$, with $t = L = 2^{19.25}$ used for the first value for Distinguished Points while the second value is calculated with $t = t_{max}$. The results are shown in the following table.

Method	Computation (Precomp.)	Time (Precomp.)
Distinguished Points	$2^{57.47}$ / $2^{59.22}$	36 / 121 days
Rainbow Tables	$2^{56.79}$	22.5 days

The number of computations and the resulting time usage for Distinguished Points is split into two values. The first value represents an implementation where each chain is calculated and transported seperately from the COPACOBANA to the storage medium. But our planned design for the TMTO calculates multiple chains in parallel, as is the current brute force design. Hence, the first chain which reaches an endpoint has to wait until the last chain, processed by a single FPGA, is finished before the transfer to the storage medium can be initiated and this is why the second value uses t_{max} as chainlength. In practise, the expected precomputation time will be somewhere between 36 and 121 days. A possible solution to the prolonged precomputation time would be to send each computed chain as soon as it reaches an endpoint. But this solution as well increases the number of polling and control communications, resulting in an much higher bandwidth usage and leading to an overloaded connection between the controlling computer and the COPACOBANA.
Comparing both remaining methods seems to favor Rainbow Tables regarding the precomputation phase.

5.4.6 Additional Considerations

The current specification of the COPACOBANA is a 24 MBit connection. Doubling the bandwidth will allow us to use the aforementioned solution to reduce the precomputation time to about 36 days for Distinguished Points. Another possible utilisation is to halve the chainlength while simultaneously increasing the product of $m \cdot s$ by two. The effects of the increased bandwidth are shown in the next table with the following parameters:
DP: $m = 2^{17.5}$, $s = 2^{21.5}$, $t_{min} = 2^{16.2}$, $t_{max} = 2^{20}$, $dp_l = 18$, $L = 2^{18.25}$, $L' = 2^{17.7}$
Rainbow: $m = 2^{35.51}$, $s = 8$, $t = 2^{18.2}$
each reaching a success rate of 80% but with an increased memory usage.

Method	Time (Precomp)	Time (Online)	Memory Usage
Distinguished Points	31 / 104 days	42 hours	3500 gigabyte
Rainbow Tables	22 days	24 hours	4100 gigabyte

Both methods benefit from the increased bandwidth, but Rainbow Tables gain even more than Distinguished Points because of the structure of a rainbow table regarding the online phase.
During this phase $\frac{t^2}{2}$ computations are needed, hence reducing t by a factor of 2 decreases the computations by a factor of 4. But the benefit of an decreased online time comes at the cost of an increased memory usage. In this field, Distinguished Points excel since the increased number of tables and startpoints don't use up as much memory as compared to Rainbow Tables. While both methods are similar considering the current specification of the COPACOBANA, Rainbow Tables do profit more if the bandwidth of the COPACOBANA is increased.

This fact has an considerable impact on our decision, especially since a 1 gigabit bandwidth module is currently in development

Another aspect which can be used to compare both methods is their required space on a single FPGA. Since there is no implementation of both methods, the following facts are approximations of how many logic gates are used from either Distinguished Points or Rainbow Tables.
The special feature of Distinguished Points is the check wheter an currently processed intermediate point reaches an endpoint or not which involves dp_l bits. This check needs to compare the dp_l bits of the mask with the current intermediate point which can be done using a simple XOR. After each endpoint check the current chainlength of the processed intermediate point needs to be checked as well. This can be done in the same way as for the endpoint check comparing the current chainlength with t_{min}. Both checks can be done in parallel during a single stage using $dp_l + log_2(t_{max}) = 19 + 21 = 40$ XORs for each implementation of DES on a single FPGA
In addition to sending t_{max} which equals sending t for Rainbow Tables, t_{min} and the DP-property with the length of dp_l needs to be sent as well. This results in additional communications during configuring a single FPGA and using up 40 D-FF's for storing them.
Rainbow Tables' special feature are the different reduction functions. They can be implemented using a counter which is XOR'ed with the currently processed intermediate point. Considering our chainlength, t different reduction functions need to be generated. This results in the usage of 20 XOR for each DES implementation on the FPGA. The value which is XORed with the intermediate point can be derived from the counter which counts the number of F applications already computed, hence taking up no additional logic gates.
The communication overhead is not increased for Rainbow Tables, as is the need for storing additional informations. In addition to the above mentioned usage of gate logic, the table index needs to be stored for both methods. Distinguished Points takes up an additional $log_2(s) = 22$ D-FFs and XOR while Rainbow Tables only needs 3 D-FF and XOR due to the lower number of tables.
Comparing both methods results in a lower logic gate usage for a Rainbow Tables design as for a Distinguished Points design.

5.4.7 Chosen Method

Rainbow Tables provide a shorter precomputation and only slightly longer online time than Distinguished Points. If the development of the bandwidth increase is finished, even the online phase will be shorter than for Distinguished Points. Additionally, Rainbow Tables do take up less logic gates in a design and the precomputation phase takes only about 23 days. In this time, only 4 brute force

attacks can be executed while 6 to 20 brute force attacks can be executed while waiting for the completion of the Distinguished Points precomputation phase. In consideration of these facts, Rainbow Tables will be the method we implement in our TMTO design for the COPACOBANA.

6 Implementation and Results

During the last chapter the best suited method for our implementation was chosen to be the Rainbow Tables method by Oechslin. This method now needs to be implemented in VHDL and loaded onto the COPACOBANA. The first part of this chapter introduces the design and how it will be realised in VHDL while the second part gives a rough overview of the C program which controls the COPACOBANA. The performance results like achieved clock frequency and used logic gates as well as their effects on the precomputation and online phase are presented in the third part.

6.1 Design Overview

Important for the implementation with VHDL is to know exactly what needs to be implemented and which modules are neccesary. A rough scetch of the different aspects of a Rainbow Tables implementation is shown in figure 6.1.

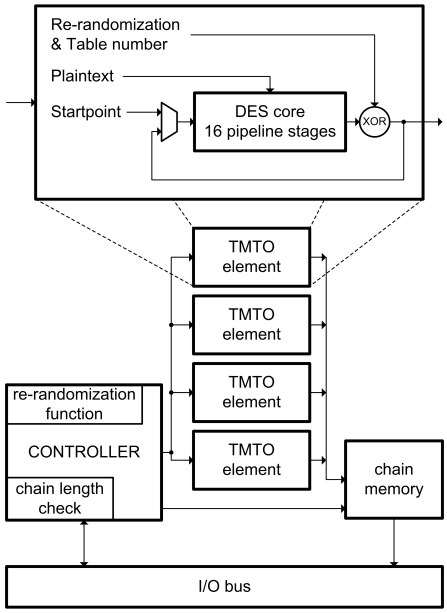

Figure 6.1: Scetch of the Rainbow Tables Design

Our design is split into three major modules (controller, TMTO, storage) which will be introduced more briefly. Basically, the need for modules is not a given. But we need to keep in mind that it should to be possible to optimise or change the source code if a modified Rainbow Tables method or maybe even a change to Distinguished Points or another method is to be implemented later on. Therefore the design is split in different modules which ease the work of the next person who wants to work with it.

6.1.1 TMTO Module

The TMTO module represents the core of our design. It is responsible for all computations during the precomputation phase. It is again split into three different modules with the first being the DES implementation itself. The DES implementation or *DES CORE* is a slightly modified DES implementation derived from the standard *zitieren*. It is realised by a pipelined approach, which means that all 16 rounds are implemented. This takes up more space than an interative approach but with certain advantages which lead to a faster and more efficient design.
Breaking a single DES round down to its basics leaves only three functions left to be implemented in VHDL. First the XOR between the right 32 bits of our processed intermediate point and the roundkey, then the 8 S-boxes and finally the XOR between the S-boxes and the left side of our processed intermediate point. Everything else is subject to wiring. The whole key scheduling can be realised only through wiring since it is possible to calculate which bit of our key is present at which position during each of the 16 rounds. This can only be done if an pipelined design is chosen because the key scheduling is not identical for all rounds, sometimes shifting the processed key once, sometimes twice. In an iterative approach the key scheduling would take up logic gates to choose if it is neccessary to shift once or twice in a given round by using a multiplexer. Figure 6.2 shows the parts which need to be implemented for a single round in an pipelined approach.

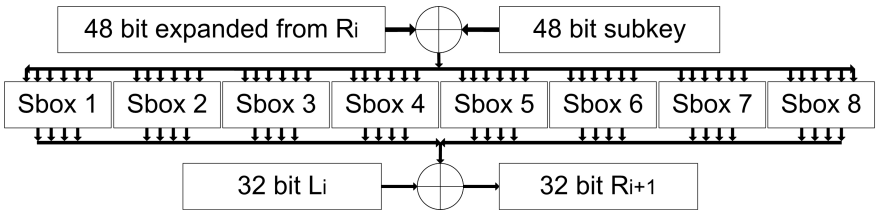

Figure 6.2: Schematic of a single DES round

48 XOR before, and 32 XOR after the S-boxes, while the S-boxes itself are realised through table lookups. For each round, this design takes up 80 XOR, 224

4-bit lookup tables and 120 D-FFs for storing the processed intermediate point and key. A full standard DES implementation takes up a total of 1280 XOR, 3584 4-bit lookup tables and 1920 D-FFs.
The number of XORs and lookup tables is fixed, but the number of D-FFs can be reduced slightly. A standard DES produces a 64 bit ciphertext from which we only need 56 bits for the next application of F. This means that we don't need to store 64 bits of our processed intermediate point in round 16, but only 56. Its a saving of only 8 D-FFs, but everything that saves us logic gates is appreciated.

The second part of our TMTO module is the reduction function. The different reduction functions are done by using a simple counter which is increased after each application of F, hence after 16 clock cycles. This counter is XORed on the highest bits of our computed intermediate point before it is being processed in the next cycle. In addition to this the current table id, which is used to generate different tables, is XORed as well with the intermediate point.

Our TMTO module incorporates 4 DES CORES with the startpoint scheduling being the focus of our last part of the TMTO module. Only 50 bits are sent by the controlling computer, and the last 6 bits of our startpoint are distributed as follows:
A total of 4 DES CORES is used, which means that each one gets a fixed 2 bit value which is attached to the 50 bits of the startpoint. Additionaly 4 bits are used by each core to address the 16 processed startpoints. This is due to the fact that we use a pipelined approach with 16 stages. The counter which is used to generate the different reduction functions as well is used to compare the current number of F-applications with our chainlength. If the chainlength is reached, a signal to the storage (via the controller) is sent which tells it to store the next 16 endpoints sent by the TMTO module. After 16 additional clock cycles another signal is sent to the controller telling it that the computation is finished.

6.1.2 Storage Module

The storage module incorporates 4 dual channel RAM modules. Each of the modules can store 16 entries, with each entry being 62 bits long. These 62 bits are split into the 56 bits used up by the endpoint, with the remaining 6 bits representing the part of the startpoint which is not sent via the controlling computer, but defined by the 4 DES CORES and the pipeline stages as mentioned above.
Two different clock domains are used by the storage module. The first clock domain is as fast as the bus. It is neccesary to establish a reliable communication between the bus and the storage without using additional logic elements which

check if a packet was being sent or not. The second, and faster, clock domain is used to connect the storage module with the TMTO module.
The functionality of the module itself is very limited: It stores the receiving 62 bit values if writing is enabled, and sending the values if reading is enabled. The few logic elements in this module are used to choose which of the 4 RAM modules is allowed to send its entries to the data bus.

6.1.3 Controller

Similar to the TMTO module, the controller is divided into three parts. The first part manages the communication from and to the data bus. Control signals like reset, the sending of the plaintext, chainlength or a new startpoint are processed here. In addition to this the controller is responsible for when to write on the bus and when not. This is neccessary since the COPACOBANA can be damaged if multiple FPGAs are writing simultaneously onto the bus. While being polled, the controller as well signals the controlling computer wheter a computation is finished or not and the controller receives the stored endpoints which are sent to the bus after a finished computation. This first part as well generates the two clock domains which are used for the design.
The second part handles the TMTO module. The plaintext and chainlength are sent to the TMTO module as well as the startpoints received during the precomputation process. Furthermore the controller is responsible for the communication between the TMTO module and the storage module.
The third part controlls the storage module. The communication frequency between those two modules is low since the controller only signals the storage module if it needs to store the computed endpoints presented by the TMTO module or if it needs to send the stored endpoints to the bus.

Regarding the modular design a modification of the current rainbow implementation can be done quite easily. If, for example, rainbow sequences are to be used the only change is neccessary in the TMTO module. In here an additional counter needs to be used which counts the current number of processed sequences and compares them to the value of the chainlength. The chainlength itself can be split into two parts. The highest bits representing the number of sequences and the lower bits the chainlength of each sequence.

6.2 PC Program

This section will give an overview of how things were done instead of listing the 2033 lines of code. Four programs were written for the whole TMTO. First is the

precomputation program which manages the communication between the controlling computer and the COPACOBANA. It is similar to the program used for the brute force attack, with an additional part which stores the startpoint and endpoint tuples into a table. It enables the user to start and stop a precomputation as well as resuming it from an earlier state. The storing itself is done in the following way. Currently a single directory is created for each generated table. In this directory the stored tuples are split into 4096 subtables, with each subtable storing endpoints which end with the same 12 bits. This is done to quicken both the search through the tables as well as the neccessary sorting after the finished precomputation. The sorting itself is made by the second program using the sorting algorithm Quicksort. To further decrease the time spent on sorting, each subtable is loaded into RAM for faster processing. The sorting of a table can be started right after the precomputation of the first table is finished, resulting in a precomputation time which is calculated with the time for the computations plus the time neccessary for sorting a single (the last) table.
The online phase uses the remaining two programs. Normally the online phase works as follows: First an intermediate point X' is calculated, then it is compared to the table. But in doing so the table accesses are quite random, with accessing a specific subtable two times in a row being unlikely. Hence, a single subtable is likely to be loaded more than once into RAM. A possible solution to this was already presented in chapter 5 which reduces the number of accesses to the storage medium speeding up the whole online phase process. So while the third program searches for key candidates (a match between X' and the stored endpoints), the fourth checks these candidates for their validity.

6.3 Achieved Result

Up till now all approximations of time usage were done using the brute force implementation with its $2^{35.9}$ encryptions per second. But since our design is finished, more accurate approximations of the precomputation and online phase times can be done. The comparison of both designs is shown in the following chart.

Type	Time Memory Tradeoff	Brute Force
Slice Flip Flops	6906	
4 bit LUT	14128	
Occupied Slices	7571	
Clock Frequency	96 MHz	136 MHz

A slightly less efficient design was anticipated compared to the brute force design which uses less logic gates and was optimised several times. The current

difference of the stable clock frequency still is a bit disappointing. Maybe an optimised TMTO design can provide a higher clock freqency matching the speed of the brute force design.

In addition to the clock frequency, the brute force design continuously computes encryptions while our TMTO design needs to wait for the computer to collect the computed chains before starting with a new computation. This fact further reduces the number of encryptions per second of our design. Lets consider the ideal scenario for our design. Right after the chain computation is finished the computer polls the FPGA. The FPGA sends its answer and the 50 bits of the startpoint it was processing. The FPGA then starts to send its 64 entries stored in the storage module. The computer receives all entries and sends a new startpoint to the FPGA with the final command being a reset to start the new chain generation. This adds to a total of 67 messages which are sent forth and back and during which the FPGA does not compute any chains. This 67 messages are sent using a 24 mbit connection, hence using a frequency of 24 MHz. And each message uses a 64 cycle window for a safe communication. This translates in a loss of $67 \cdot 64 \cdot \frac{96}{24} = 17152$ clock cycles for each FPGA since our TMTO module loses 4 clock cycles for each clock cycle of the bus. In total $17152 \cdot 480 \approx 2^{23}$ encryptions are lost in a best case scenario. This reduces the number of computations of our design to $2^{35.42}$ encryptions per second.

Finally the anticipated results of our Rainbow Tables TMTO are the following:

Parameter	Value
Success Rate	79.9%
Memory Usage	2055 gigabyte
Precomputation Time	31.4 days
Online Time	69 hours

The time needed for the online phase represents a worst case scenario, with an average Rainbow Tables TMTO taking up about 35 hours. Neglecting the time for the precomputation, the following list compares the different attack methods executing a known/chosen plaintext attack on DES in an average case scenario.

Attack Method	Time	Cost
DEEP Crack Brute Force	56 hours	250.000 dollar
COPACOBANA Brute Force	160 hours	10.000 dollars
COPACOBANA Rainbow TMTO	35 hours	10.500 dollars

The above table shows that out Rainbow TMTO is a cost efficient and very fast attack which is executed sucessfully in about 80 % of all cases.

7 Final Thoughts

This final chapter summarises our findings of this thesis as well as to give an conclusion of how time memory tradeoffs may fare in the future as well as possible optimisations to our current design.

The first part of this thesis was dedicated to analyse the suitability of three basic time memory tradeoff methods executing an attack on the Data Encryption Standard. For this purpose we used the specialised FPGA array COPACOBANA which is already successfully executing brute force attacks on DES. While all three introduced methods are feasible in a theoretical approach, the analysis of their use with the COPACOBANA pointed out their weaknesses in regard to a practical realisation. As of our findings in chapter 5, the Rainbow Table method by Oechslin proved to be the best suited method out of those three.
This method was then designed and implemented for the use of the COPACOBANA, as presented in chapter 6. The performance of our design showed that the TMTO design proves to be superior to a brute force method using either the COPACOBANA or Deep Crack machine if the attack is executed multiple times on the same known plaintext. An attack on DES in with a known plaintext is executable within 35 hours on average using our implemented TMTO design, with both brute force designs being slower. This shows that our Rainbow Tables TMTO design provides a cost and time efficient solution to breaking DES in a known or chosen plaintext scenario, with the online phase being executed by a common retail computer instead of a more expensive hardware solution. This allows virtually everyone with a computer and access to the precomputed tables to break DES within few days instead of the 1000 years which pass while executing a brute force attack.
Our design proved to be slightly less efficient than the already optimised design for the brute force attack, but further refinement could lead to a better performance. This increase in performance will lead to an even faster precomputation time, with the following section introducing some possible optimisations which could not be accounted for in the initial thesis due to limited time and/or money.

7.1 Possible Optimisations

Already mentioned multiple times throughout this thesis, one major limitation of our time memory tradeoff was given by the available bandwidth of the COPACOBANA. The positive effects of a bandwidth increase were already outlined in chapter 5. An increased bandwidth provides us with the option to reduce the chainlength. Since the value of t is used in all but the memory usage formula, reducing the chainlength reduces the success rate, the precomputation and online time alike. While this results in an positive effect for our precomputation and online time, the success rate drops. This can be compensated with an increase of either the number of startpoints, the number of tables, or a combination of both. Hence, reducing the chainlength while keeping our success rate results in an increased memory usage up till a point where the data is no longer manageable or searching the tables takes too much time compared with the computations neccessary during the online phase.
Our current approach has a limit of 2 terabyte of memory. Halving the chainlength, while keeping the 80 % success rate will double our memory usage. The time to load this amount of data into RAM will take up 20 hours. Considering our online computation time of 69 hours, a reduction of the chainlength to at most $\frac{2^{19.31}}{log_2(\frac{69}{10})} = 2^{16.53}$ seems feasible. Reducing the chainlength any furter results in a longer access and search time than computation time. This can be compensated by a faster storage solution. One way to do so is using a RAID system which simply increases the data throughput. The other possibility is to use a storage medium with a much faster access time. In doing so, the need to load the tables into RAM is not a given any more. Instead, the computed X' can be searched directly on the storage which as well eliminates the need to compute and sort all X' before starting the search of a table. The current IDE/SATA/SCSI hard disks do not provide the level of access times neccessary for this approach, but the newly developed *solid state disks* (SSD) do. They provide access times of $\approx 0.2ms$ which for instance reduces our native access & search time for Rainbow Tables of 197 hours to 9.8 hours in a worst case scenario.
At first this seems like a poor improvement since solid state disks are a lot more expensive than SATA drives, but contrary to our earlier solution, the tables can now be searched in parallel. Each Rainbow Table can be stored on a SSD and searched independently while this is not possible with our current solution of the online phase. And considering a reduced chainlength this results in fewer accesses for a single table, but being compensated with more startpoints and tables. The increased number of tables results in more accesses in overall, but they can be searched in parallel if each table is stored on a seperate SSD. So using solid state disks for the storage medium provides an even faster online time if it is combined with the increase of the COPACOBANA's bandwidth. The only downside of this modification is the highly increased cost for the storage solution since the price

for a single gigabyte rises from $\approx$ 0.2 dollar to $\approx$ 20 dollar, but with all new developements, the prices of SSDs will drop over time.

The following table shows the effects on our TMTO if the bandwidth is increased to 1 gigabit and solid state disks with an access time of 0.2 ms are used while fulfilling the success rate constraint of 80 %. The parameters used are: $t = 2^{15}$, $m = 2^{36}$, $s = 52$.

Parameter	Value
Success Rate	80%
Memory Usage	38272 gigabyte
Precomputation Time	29 days
Online Time	6 minutes

The low online time is due to the fact that only $\frac{2^{15} \cdot 2^{15}}{2}$ applications of the step function F are needed for a single table, so the above online time holds true if the controlling computer can compute at least $2^{20.5}$ DES encryptions each second while being able to access the 52 SSD disks simultaneously. The huge downside is the cost for the storage medium of about 720.000 dollars.

Another, less expensive, solution is to split the tables between multiple computers. In this way, both the access & search time and the computation time during the online phase is divided by the number of used computers. Five tables are generated during the precomputation phase which can easily be distributed to 5 computers. Each of them needs to compute only a single set of X' and compare it to their Rainbow Table. This results in a reduced online time averaging at 7 hours.
Splitting up the table onto more computers is possible as well since our implementation divides each generated table into 4096 subtables resulting in a total of 20480 distributable subtables. Each of these subtables has a size of $\approx$ 97 megabytes which, for instance can easily be loaded into RAM on a permanent basis. Computing a set of X' for a single table is distributed on the 4096 computers, with each of them computing 158 X'.
This is possible, because of the structure of Rainbow Tables. A single computer would need to calculate X'_1 by applying F once, then X'_2 by applying F twice and so on. Two computers could split the computations, with the first one computing all even X'_i while the second computes all odd X'_i with both of them computing $\frac{2^{19.31} \cdot 2^{19.31}}{2 \cdot 2}$ steps. Using 4096 computers gives each computer the task to execute $\frac{2^{19.31} \cdot 2^{19.31}}{2 \cdot 4096} = 2^{25.6}$ steps. The computed X' subsets then are stored on a single computer. There, the set is sorted and divided into new subsets and distributed to the computers with the corresponding subtable. Without considering the time to send and sort the computed subsets of X' the online time in

this approach can be executed in about 12 seconds. The data which needs to be sent to the central computer, sorting and redistributing the subtables, has a size of $\frac{2^{19.31} \cdot 56}{8} = 4444$ kilobyte for each table. This data can easily be sent within a second if the computers are connected via a 100 Mbit ethernet network connection, with the redistribution taking up another second. The quicksort algorithm has to sort all $2^{19.31}$ X' which results in $\approx 2^{23}$ steps in average. Our quick-and-dirty quicksort implementation was executed in 65 seconds, resulting in an online phase which can be executed in about 79 seconds if 20480 computers are utilised in a cluster-like structure.

7.2 Future Time Memory Tradeoffs

The result of the thesis showed that breaking the DES, which uses a security parameter of 56 bit, is a possible task. It is common knowledge that a 56 bit cyper is no longer secure, partially due to the mentioned brute force attacks which take only few days to achieve the key. This final section gives a rough approximation of how secure an encrypted plaintext is, if an 80 bit block cipher is used, with the attack being executed in 15 years. For this we assume that *Moore's Law* holds true for the next 15 years, resulting in an increased computation power of 2^{10}. In addition, we assume that a COPACOBANA-like machine is using FPGAs which as well are 2^{10} times faster than the ones used in the current COPACOBANA. We will call this machine *COPA2* with a computation power of $2^{45.4}$ encryptions each second. Our final assumtion is, that the design of the 80 bit cipher is similar to the DES, so that our inital TMTO design can be used.
The parameters used for this Rainbow Tables TMTO are the following: $t = 2^{30}$, $m = 2^{40}$, $s = 1650$. The table below shows our estimations of the TMTO performance using a single COPA2.

Parameter	Value
Success Rate	80%
Memory Usage	25344 terabyte
Precomputation Time	1330 years
Online Time	221 days

The online phase is executed with the COPA2 as well, because the computations are a lot more extensive with a chainlength of 2^{30}. This demonstrates that an 80 bit cipher is resistant to our current TMTO design using a single COPA2. But encrypting vital information is not recommendable, because the purchase of 1000 COPA2 reduces the precomputation time to 486 days and the online phase could be executed in about 5 hours if all COPA2 can be fully utilised for the computations.

Bibliography

[BPV98] J. Borst, B. Preneel, and J. Vandewalle. On time-memory tradeoff between exhaustive key search and table precomputation. In P. H. N. de With and M. van der Schaar-Mitrea, editors, *19th Symp. on Information Theory in the Benelux*, pages 111–118, Veldhoven (NL), 28-29 1998. Werkgemeenschap Informatie- en Communicatietheorie, Enschede (NL).

[D. 82] D. Denning. Cryptography and Data Security, p.100. Addison-Wesley, 1982.

[Hel80] M. E. Hellman. A Cryptanalytic Time-Memory Trade-Off. *IEEE transactions on Information Theory*, 26:401–406, 1980.

[Oec03] P. Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Proceedings of CRYPTO 2003*, volume 2729 of *LNCS*, pages 617–630. Springer-Verlag, 2003.

[SRQL02] F. Standaert, G. Rouvroy, J. Quisquater, and J. Legat. A Time-Memory Tradeoff using Distinguished Points: New Analysis & FPGA Results. In *Proceddings of CHES 2002*, volume 2523 of *LNCS*, pages 596–611. Springer-Verlag, 2002.