A Project Report for CSN-300 (Lab Based Project) of Spring Semester 2020-2021

On

TMTO attack on Light-Weight Cipher

Submitted by

Radhika (18114060)

`radhika@cs.iitr.ac.in`

Rishi Ranjan (18114066)

`rranjan@cs.iitr.ac.in`

Shubhang Tripathi (18114074)

`stripathi1@cs.iitr.ac.in`

Supervised by

Prof. Sugata Gangopadhyay

[illegible]
18/5/21

Department of Computer Science and Engineering

Indian Institute of Technology (IIT) Roorkee

May 4, 2021

Abstract

In this work, we present a time-memory trade-off attack on a light-weight cipher. The light-weight cipher used is a substitution permutation network implemented by adding a substitution block to the Feistel System (implemented as discussed with our supervisor) as presented in [1]. We study the time-memory trade-off attack on DES as discussed by Hellman [2].

In this paper, we present a novel way to execute a TMTO attack on a cryptosystem with larger keyspace than the plaintext and ciphertext space. In Hellman's paper and other variants of TMTO presently available the cryptosystems discussed include only those with smaller or equal keyspace than the ciphertext or plaintext space. We also show the implementation of the TMTO variant on the cipher discussed here.

Contents

1 Introduction

1.1 Background

Feistel structure is a popular block cipher design scheme [3]. Many popular modern symmetric key cryptosystems like DES, Triple DES, Blowfish, RC2, RC6 are based on the feistel network design.

Indeed many attacks have been lead against the design scheme, but there has been no complete cryptanalysis of the cipher. Hellman proposes a time-memory trade-off attack against DES [2]. The time-memory trade-off attack ellucidated in the paper restricts the cryptosystems to have equal plaintext space, ciphertext space ad keyspace. However Hellman suggests a few methods to deal with smaller keyspaces, no tweaks regarding attack against cryptosystems with larger keyspace are discussed.

We first discuss the general Feistel system design 1.3.1 and PRESENT Lightweight cipher 1.3.2. We then discuss the prevailing Time-Memory Trade-off attack by Hellman in 1.3.3. We introduce the cryptosystem implemented by using the ideology of PRESENT in Feistel Structures in 4. The attack proposed by us is delineated in 4. Finally, we analyse the trade-off curves in 5.

1.2 Notations

We use the following notation throughout this report.

- C - Ciphertext Space
- P - Plaintext Space
- K - Key Space
- p - A plaintext belonging to P
- c - A ciphertext belonging to C
- k - A key belonging to K
- N - Search Space Size
- P - Time required for pre-processing (offline) phase
- T - Time required for real-time (online) phase
- M - Memory storage space available to attacker
- $| * |$ - The size of set $*$

For the cryptosystem present in this report $|C| = 2^8$, $|P| = 2^8$ and $|K| = 2^{12}$. N, P, T, M are discussed later in 4 and 5.

1.3 Preliminaries

1.3.1 Feistel Cipher

Feistel Ciphers are a popular class of modern round-based block ciphers. Their structure consists of multiple rounds of application on a block, where each round consists of substitution and permutation steps, each round may use a **Key** for encryption which may be same for all rounds or different depending on the cipher.

The block is divided into two equal halves: left (L), right (R). In each round, the right half is unchanged while the left half is encrypted using the right half and the round key. This encryption operation is also referred to as the **Feistel Function**. After this, the two halves are swapped with each other. This process is repeated for the number of rounds that are specified in the cipher definition.

In a feistel structure, the relationship between the input and output of the i'th round is:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$

where L_i, R_i represent the left and right halves of the block after i rounds of processing, F is the aforementioned Feistel Function and K_i is the round key for i'th round.

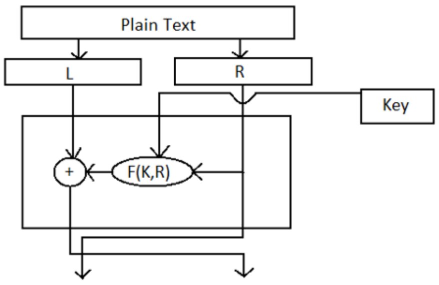


Figure 1: Round of a Feistel Cipher

1.3.2 PRESENT

PRESENT is a very light-weight block cipher meant to be used for small devices with less chip area and power constraints. Although being a very light weight cipher, it was designed to be as cryptographically secure as possible, while providing good hardware performance.

It is an example of a Substitution - Permutation Network consisting of 31 rounds. In each round, a round key is generated which is XORed with the state of the block at that point. Then the block is passed through a substitution box which is basically a mapping from 4-bits to 4-bits. It is this sbox which we use in our cipher. After this it uses a bit-based permutation.

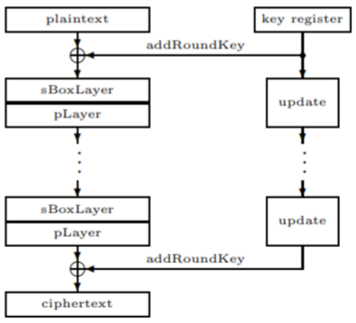


Figure 2: Structure of PRESENT Cipher

1.3.3 Time-Memory-Tradeoff Attack

Time memory tradeoff attack, as the name suggests, is an attack which has a trade-off between computational time and the memory required to carry out an exhaustive key search attack on a cryptographic algorithm.

Typically an exhaustive keyspace search for a cryptographic algorithm requires a large amount of computational power due to (generally) large keyspaces and little heuristic to guide this search. On the other hand, a constant time search requires the storage of all possible 3 tuples of (P, C, K) which again requires very large amount of memory given by

$$M = |P| * |C| * |K|$$

as well as a very resource heavy precomputation (offline) phase. Thus, a tradeoff between two requirements (memory and time) is considered while mounting such key search attacks on a cipher.

The standard TMTO attack (as proposed by Hellman [2]) is basically a chosen/known plaintext attack with the following phases. Let P_0 be a fixed plaintext block and we define a function f such that

$$f(K) = R[S_k(P_0)]$$

In other words, for a given plaintext P_0, f is a function from K to K.

The cryptanalysis begins, with us choosing m keys from the keyspace (SP_1, SP_2 etc). For ($1 \leq i \leq m$) we define

$$X_{i0} = SP_i$$

. And further we define

$$X_{ij} = f(X_{i,j-1})$$

.

And thus for the given plaintext P_0 and ciphertext C we construct our matrix of $X_{i,j}$ of the dimensions $m * t$ as shown in Fig 3. Cryptanalysis is as simple as finding C inside this matrix, say at $X_{i,j}$, then $X_{i,j-1}$ is the key of the cryptosystem.

Although we may get false positives (the key might not be correct for all the further encryptions) but we can ignore these.

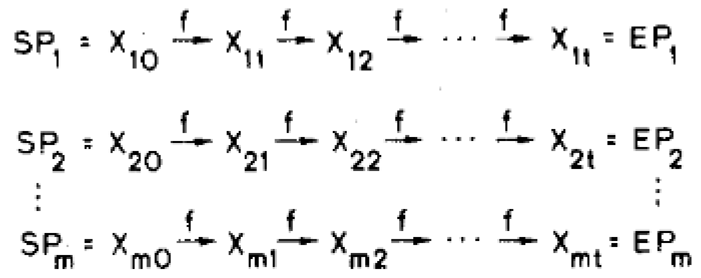

$$\begin{tikzcd}
SP_1 = X_{10} \arrow[r, "f"] & X_{11} \arrow[r, "f"] & X_{12} \arrow[r, "f"] & \cdots \arrow[r, "f"] & X_{1t} = EP_1 \\
SP_2 = X_{20} \arrow[r, "f"] & X_{21} \arrow[r, "f"] & X_{22} \arrow[r, "f"] & \cdots \arrow[r, "f"] & X_{2t} = EP_2 \\
\vdots & & & & \vdots \\
SP_m = X_{m0} \arrow[r, "f"] & X_{m1} \arrow[r, "f"] & X_{m2} \arrow[r, "f"] & \cdots \arrow[r, "f"] & X_{mt} = EP_m
\end{tikzcd}$$

Figure 3: Matrix of Images under f

2 Literature Review

2.1 Symmetric Key Cipher

In a symmetric key cipher, if a message is encrypted using some key, then the same key is needed to decrypt the obtained ciphertext. Some symmetric cryptography algorithms are AES, DES, Blowfish, RC5, etc. The components of symmetric key cipher are:

1. Plaintext: data input to the encryption algorithm.
2. Encryption algorithm: a sequence of operations which convert the plaintext into the ciphertext using the key.
3. Secret Key: the key used to encrypt and decrypt messages.
4. Ciphertext: output of the encryption algorithm. This must be sent to the decryption algorithm along with the key to recover back the plaintext.
5. Decryption: sequence of operations which transforms an encrypted message back to original message using the Key.

A good review of **symmetric key cryptography** can be found in **A Review on Symmetric Key Encryption Techniques in Cryptography** [4].

2.1.1 Block Ciphers

A symmetric key block cipher has 2 algorithms - Encryption (E) and Decryption(D), which take n bits as input and using k bits of secret key, give n bits of output. These can be classified into:

- Feistel Networks
- Substitution Permutation Networks
- Unbalanced Feistel Networks

Several algorithms like DES, AES, triple DES etc are also block ciphers. **Survey: Block cipher Methods** [5] discusses various block cipher algorithms.

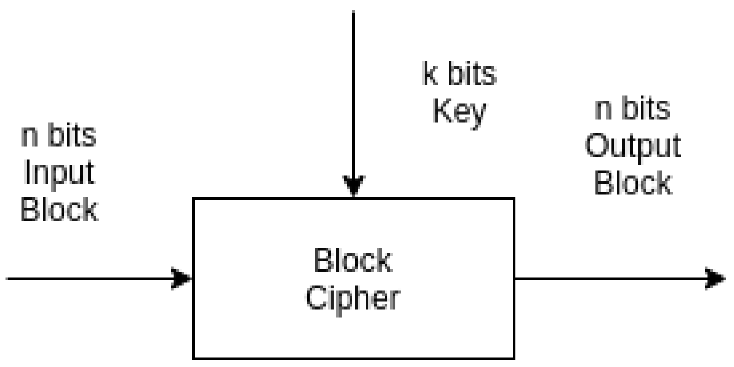


Figure 4: Structure of a Block Cipher

2.1.2 Substitution Permutation Networks

Substitution Permutation networks are linked mathematical operations (to produce substitutions) which are widely used in most of the block ciphers [6]. Every such network has a Substitution box, called an S-Box which is a one-to-one function from one particular block of bits to another block of bits. This S-Box adds an extra layer of "randomness" to the encryption mechanism of the cipher. According to Shannon, a good S-Box should add good amount of "confusion" and thus predicting this substitution network should be difficult [7].

In order to strengthen the encryption property of our block cipher, we use a Substitution Permutation Network, with the S-Box as introduced in [1].

2.2 Cryptanalysis of Cryptosystems

Cryptanalysis is the study and analysis of a cryptosystem in order to effectively decipher the coded message (ciphertext) without the knowledge of key.

Cryptanalysis varies on the basis of information available to the attacker:

- **Passive:**
  - **Ciphertext-only :** The attacker has access to only certain ciphertexts
  - **Known Plaintext :** The attacker has access to certain plaintext, ciphertext pairs for a specific key.
- **Active:**
  - **Chosen Plaintext :** The attacker can obtain ciphertexts for arbitrary set of plaintexts chosen by themselves.
  - **Chosen Ciphertext:** The attacker can obtain plaintexts for arbitrary set of ciphertexts chosen by themselves.

The baseline for any cryptosystem is ciphertext-only. The minimum requirement of security for any cryptosystem can be stated as it must be computationally secure against a brute force attack. A brute force attack is executed when an attacker tries all the possible keys until found correctly.

Another type of attacks are side-channel attacks. After design phase of a cryptographic algorithm the implementation on physical system can generate unintentional information leakage of the secret information. This is called a side-channel attack and the attacker can use it to reduce the entropy or cryptographic strength.

2.3 Pseudorandom Permutation Functions

Random permutation is a permutation which is selected randomly from a set of possible permutations of a function's domain. Thus, a pseudorandom permutation is one which cannot be distinguished from a random permutation.

Mathematically a pseudorandom function is defined as

$$F = \{f_s\}_{s \in \{0,1\}^*}$$

$$f : \{0,1\}^m \to \{0,1\}^m$$

The condition on f is that it should to one-to-one in nature.

Construction methodologies of pseudorandom permutations has been a well researched topic and one of the most prominent methods of pseudorandom permutation generation using pseudorandom functions generator has been nicely studied in [8]

To increase the search space coverage of the time-memory-tradeoff attack proposed in this report, we utilise pseudorandom permutation functions.

2.4 TMTO Attack

The TMTO (time-memory-trade-off) attack was first introduced by Martin E. Hellman in his article titled **A Cryptanalytic Time - Memory Trade-Off** [2] in the IEEE transactions on Information Theory. He introduced it as "A probabilistic method, to cryptanalyze an N key cryptosystem with $N^{2/3}$ operations with $N^{2/3}$ words of memory". This was after a precomption phase of N operations. With this method, he mounted an attack on the DES cryptosystem, which was the defacto standard for encryption during that time. As mentioned before, this was a chosen plaintext attack.

The basic idea was that if one were to precompute the ciphertexts for all possible keys possible in the Keyspace of size N for a chosen plaintext, one could just lookup an unknown ciphertext in this data and find out the key which was responsible for that. This needed a memory of O(N) and a time complexity of $O(\log_2 N)$ for sorted ciphertext or O(1) if using a hashtable. On the other hand, one could also just compute all these ciphertexts whenever needed, this would require a memory of O(1) and time complexity of O(N). These are both very impractical for large keyspaces like those of DES(2^{56} keys). Instead, Hellman proposed a method which lies somewhere in between these two extremes, which requires $O(N^{2/3})$ time complexity and $O(N^{2/3})$ memory requirement.

Unlike our cipher, where the Keyspace(K) is larger than the ciphertext(C) and plaintext(P) spaces, DES had a smaller Keyspace compared to the ciphertext and plaintext spaces. For Hellman's attack, one needs to have:

$$P = C = K$$

.

In his article, in order to mount his attack on DES, Hellman used a simple reduction function to reduce a ciphertext (64 bit) down to 56 bits. This is very different from our case, where we had to append more bits to the ciphertext.

The most important takeaway from this was the construction of a TMTO matrix, which was as mentioned in the previous section[1.3.3]. This approach was given by Hellman, and is a central part of our TMTO attack implementation as well.

In his article, Hellman also proposed the idea of using multiple tables in order to better cover the Keyspace, which could also be processed in parallel. These tables would be generated using different reduction functions. However, he did not go into depth on this topic in that article. We implement this technique, using pseudo-random permutation functions in our model, to get better Keyspace coverage.

Hellman's article proposed an attack on block ciphers. However there also exists another class of ciphers called stream ciphers, which produce a large bit string which is XORed with a plaintext to get the ciphertext. A paper named **Cryptanalytic Time/Memory/Data tradeoffs for Stream Ciphers** [9] was published by Alex Biryukov and Adi Shamir, which focuses on implementing a time-memory-trade-off attack against stream ciphers. It also had two phases just like Hellman's approach. A major difference, however, is that it also takes a parameter D, which is the amount of output data which is available to the attacker.

3 Problem Statement

In this project, we take up the task of mounting a time-memory-tradeoff attack on a newly implemented block cipher.

Our work on this project was thus divided into two major problems.

- Implementation of a 3 round feistel network with an added substitution box from the PRESENT cipher [1]
- Mounting a time-memory-tradeoff attack on this feistel network cipher.

Even though time-memory-tradeoff attacks are quite well studied attack methodologies, this feistel network has a larger keyspace (of 2^{12} integer space) than the ciphertext and plaintext space (of 2^8 integer space each) which introduces quite a few subtleties to be taken care of. One major issue to be accounted for is collision i.e. for two keys k_1 and k_2 in K there can exist one or more common $\{p, c\}$ pairs. Thus we also introduce a novel attack strategy to deal with ciphers having larger keyspace than ciphertext/plaintext space.

4 Methodology

4.1 Cryptosystem Implementation

We begin with the implementation of a lightweight block cipher based on the feistel network. As explained earlier, we use a 3 round feistel network cipher as the basis. The block size of the cryptosystem is 8 (i.e the plaintext, ciphertext statespace is 2^8).

The key length used in our implementation is 12. Let us assume that the key structure is of the form

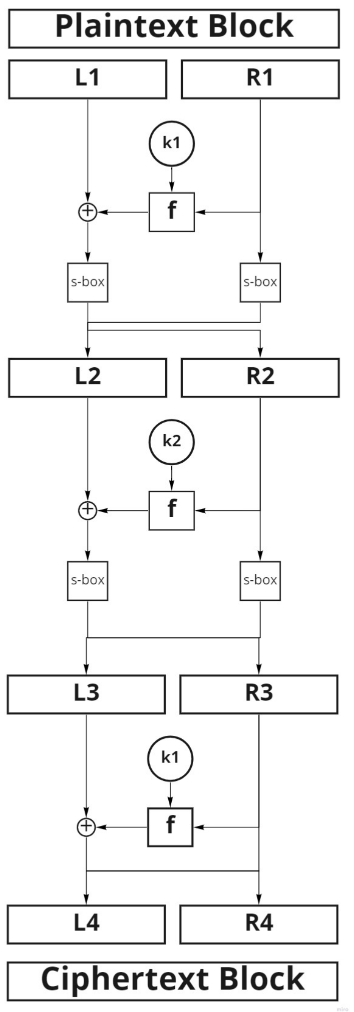
$$k = k_1 k_2 k_3 ... k_{12}$$

The round keys k^1, k^2, k^3 are defined to be the following strings of 4-bits

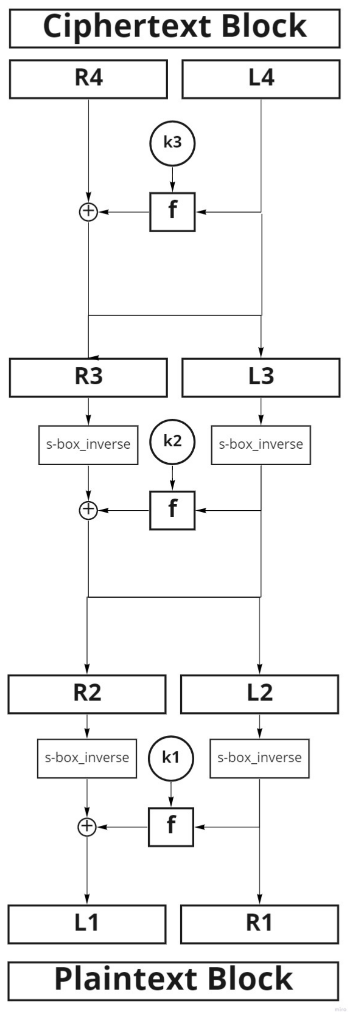
$$k^1 = k_1...k_4 \oplus k_5...k_8$$

$$k^2 = k_5...k_8 \oplus k_9...k_{12}$$

$$k^3 = k_9...k_{12} \oplus k_1...k_4$$



(a) Encryption



(b) Decryption

Figure 5: The figure 5a explains the encryption scheme used in the cryptosystem and 5b explains the decryption scheme of the cryptosystem.

The figures above have certain elements.

- k_i - Round key for the i^{th} round obtained as 4.1 explained.
- f - It is the round key function that is defined as:

$$f(x, k_i) = x \oplus k_i$$

  where k_i is the round key and x is a 4 bit string
- S-Box - A substitution box has been used in order to obscure the relationship between the key and ciphertext.
- S-box-inverse - This block is the inverse of the s-box used. Basically, for example it takes as input $0xC$ and outputs $0x0$ and so on. Refer table 1.

The substitution box that we use in our implementation is taken from the PRESENT cipher [1] and it is as shown in Table 1

X	sbox(X)
0x0	0xC
0x1	0x5
0x2	0x6
0x3	0xB
0x4	0x9
0x5	0x0
0x6	0xA
0x7	0xD
0x8	0x3
0x9	0xE
0xA	0xF
0xB	0x8
0xC	0x4
0xD	0x7
0xE	0x1
0xF	0x2

Table 1: Substitution Box

Below is a code snippet used for the encryption of a plaintext block.

```
    def encrypt_block(self, plaintext: int, **kwargs):

        left = plaintext % (2**4)
        right = (plaintext >> 4) % (2**4)

        round_keys = self.get_round_keys(key=kwargs.get('key', None))

        for i in range(NUM_ROUNDS):
            left_new = right
            right = left ^ round_keys[i] ^ right
            left = left_new
            if i < NUM_ROUNDS - 1:
                left = sbox(left)
                right = sbox(right)
        ciphertext_block = (left << 4) + right
        return ciphertext_block
```

4.2 Attack Implementation

The TMTO attack that we implement has to deal with the keyspace being larger than the ciphertext and plaintext space. This poses a problem that there are collisions in the keyspace for a given plaintext, ciphertext pair (p, k). To deal with this problem, while the construction of our matrix X during the TMTO attack, we append 4 extra bits to the ciphertext generated by our cryptosystem, this approach makes the modified ciphertext, and keyspace equal and allows us to mount to tmto attack on the encryption, by appending 4 extra random bits, we also ensure that our tmto matrix is able to covers maximum keys in the keyspace.

In effect, for a given ciphertext C, we generate the modified cipher text.

$$C_m = r_1r_2r_3r_4C$$

This modified ciphertext C_m acts as the key X_{ij} which is an entry in the tmto matrix and used to generate the next element of the matrix row $X_{i,j+1}$

4.2.1 Offline/ Preprocessing Phase

The offline/preprocessing phase of the attack involves calculating all the matrices required to do a semi-exhaustive key search. We start by decomposing the plaintext P_0 into our 8-bit blocks. For each block the function compute_tmto_lists works, which is a modified (and better) version of the precomputation phase introduced in Hellman's paper. Here, we take into account a subtlety where Hellman says $P = C = K$, but since this condition doesn't hold in our case. The size of each element of the matrix should be equal to the key size but in [2] the algorithm searches the ciphertext in the tmto_lists. The ciphertext length for the aforementioned cryptosystem is smaller than the key size thus this is not possible. To evade this we use constant bits of $length = keysize - ciphertext\_length$ for each tmto_list and the remaining bits i.e. $ciphertext\_length$ are pseudorandomly generated. For each list L_i the constant bits used are i.

Also, the tmto_lists make use of t pseudo-random permutation functions. The pseudo-random permutation functions in our case are used to generate only the remaining bits of $length = ciphertext\_length$. We define the pseudo-random permutation functions as follows:

$$f_p^j(X_{i0}) = X_{i0} \oplus (j * (2^4 + 1))$$

We first generate our basis m pseudorandom keys SP_i. For each starting key, SP_i or X_{i0}, we calculate X_{i0}^j which is the permuted list if initial basis keys,

$$X_{i0}^j = f_p^j(X_{i0})$$

. Then with the modified basis keys, we calculate the list till X_{it}^j and drop the intermediate values, storing only the initial X_{i0}^j and X_{it}^j. Varying the value of j from 1 to t, we generate t such lists, with the permute functions running over our basis pseudorandom keys to give us X_{i0}^j each time. This reduction in list size helps us reduce the storage requirement of lists from $m * t)$ for a single list to $2 * t$ for each of the t lists that we store. But since this is a time-memory-tradeoff attack, it also adds a little overhead in the online/search phase of the attack.

So, after the preprocessing phase, we have $L_1, L_2, ...L_t$ lists each with $2 * m$ elements each.

```python
def compute_tmto_lists(plaintext):
  lists = []
  sp = sample(range(0, 2**8), m)

  for l_num in range(t):
    num_to_append = (l_num << 8)
    matrix = list()
    for i in range(m):
      matrix.append([0 for _ in range(t)])
      for j in range(t):
        if j == 0:
          matrix[i][j] = num_to_append + permute(l_num, sp[i])
        else:
          matrix[i][j] = num_to_append + \
            sbox_feistel_system.encrypt_block(
              plaintext, key=matrix[i][j-1])
    l = {}
    for i in range(m):
      l[matrix[i][t-1]] = matrix[i][0]
    l = {k: v for k, v in sorted(l.items(), key=lambda item: item[1])}
    lists.append(l)
  return lists
```

The noticeable artefact above is the formulation of t lists rather than just having one matrix or list. In general, this approach helps to cover maximum number of keys. But in our case, it also helps us handle the difference between the keylength and ciphertext length.

This can be seen as another subtlety to be taken care of. As we append 4-bits to each element in order to keep in account the difference, there seem to be 2 ways to handle this. Either append 4 random bits or append 4 constant bits. One might find the appending of 4 random bits more intuitive and right. If we refer to the 3, we see that each element $X_{i,j}$ is the key to $X_{i,j+1}$ as ciphertext. This property of tmto lists is used further in 4.2.2 to obtain the key. But if we append 4 random bits and store lists instead of matrices then it is impossible to regenerate the elements as only 8 bits can be regenerated by the relation between plaintext, ciphertext and key. Thus, to evade this problem we append constant 4-bits to each element.

Another question that might arise is why not add the same 4-bits to the beginning of each element in each of the lists. The answer to this is fairly simple. As we know the tmto lists must be formulated so as to cover maximum number of keys. Since there are t Lists, adding t unique 4-bit (difference between keysize and ciphertext_length) samples to each of the t-lists respectively gives us the best place to cover maximum number of keys. Here, if we only had one list or matrix instead of t we would only be able to cover $|C|$ number of keys at maximum out of a total of $|K|$ keys. By having t unique lists, we might cover $t * |C|$ number of keys, which in our case is equal to $|K|$.

4.2.2 Online/ Realtime Phase

The online phase involves searching for the key systematically through the generated tmto lists. The function to search for keys for a block of ciphertext is *get_key_block*[4.2.2]. Thus, we are given our known plaintext block P the ciphertext block C for which the key is to be searched, and from the preprocessing phase, we have our tmto lists $L_1, L_2, ...L_t$

Since, we know that the bits appended at the beginning of any ciphertext to make it equal to the key size is i for each list L_i. Then to search for the key, with the intermediate values of the tmto matrices discarded, we start by encrypting P with C as the key (by appending appropriate bits to C depending on the list we're searching in) to get the value C_1. We then try and find C_1 inside the tmto list. If it is not found, then C_1 is used as the key to encrypt P and the search repeated. If after x such encryptions, we are able to find the resulting ciphertext C_x in the i^{th} row of a tmto list L_j, then to finally get the key for this ciphertext, we take X^j_{i0} and applying

$$X_{ij} = f(X_{i,j-1})$$

$(t - x - 1)$ number of times, get $X^j_{i,t-x}$ which is one of the intermediate values of tmto matrix we discarded, and the key for this (P, C) pair.

Thus, we search for the key in the reduced tmto lists by using repeated encryption to finaly get the ciphertext to the final vlaue of each row of every tmto list. This increases our search phase time a bit but greatly reduces the matrix storage space while also maximising the number of keys that we could search for using this tmto attack.

```python
def get_key_block(plaintext: int, ciphertext: int, block_idx: int):
  with open(f"block_{block_idx}.pkl", "rb") as f:
    lists = pickle.load(f)

  for i in range(t):
    num_to_append = (i << 8)
    c = num_to_append + ciphertext
    for column in range(1, t):
      c_t = c
      for x in range(t - 1 - column):
        c_t = num_to_append + \
          sbox_feistel_system.encrypt_block(plaintext, key=c_t)
      if c_t in lists[i]:
        x_10 = lists[i].get(c_t)
        key = x_10
        for x in range(column - 1):
          key = num_to_append + \
            sbox_feistel_system.encrypt_block(
              plaintext, key=key)
        c_expected = num_to_append + \
          sbox_feistel_system.encrypt_block(plaintext, key=key)
        if c == c_expected:
          return key
        else:
          # If collision occurs but it is not the right key
          continue
  return False
```

There is some overlap in the elements of a matrix. Thus, we check if the key found is a false positive and proceed accordingly (refer line 26-30 in 4.2.2).

```python
def get_key(plaintext: int, ciphertext: int):
  original_plaintext = plaintext
  original_ciphertext = ciphertext
  block_idx = 0
  while plaintext != 0:
    plaintext_block = plaintext % (2**8)
    ciphertext_block = ciphertext % (2**8)
    key_found = get_key_block(plaintext_block, ciphertext_block, block_idx)
    if key_found != False:
      final = sbox_feistel_system.encrypt(
        original_plaintext, key=key_found)
      if original_ciphertext == int(final, 2):
        break
    plaintext = plaintext >> 8
    ciphertext = ciphertext >> 8
    block_idx += 1
    if plaintext == 0:
      return False

  return key_found
```

Given a larger keyspace K, there is high probability to have pairs p_1, c_1, k_1 and p_2, c_2, k_2 where $p_1 = p_2$ and $c_1 = c_2$ but $k_1 \neq k_2$, i.e. collision in the $P \times C$ space corresponding to k_1 and k_2. Taking it into consideration the false positives (refer line 11 in 4.2.2) we use the *get_key_block* 4.2.2 for the next block of the chosen plaintext and corresponding ciphertext block. This problem arises in the cryptosystems considered by Hellman [2] with negligible probability, thus is not discussed there.

4.3 Tools Used

The implementation of this project is done in python. This is because of the rich support of cryptographic libraries as well as great community support for python.

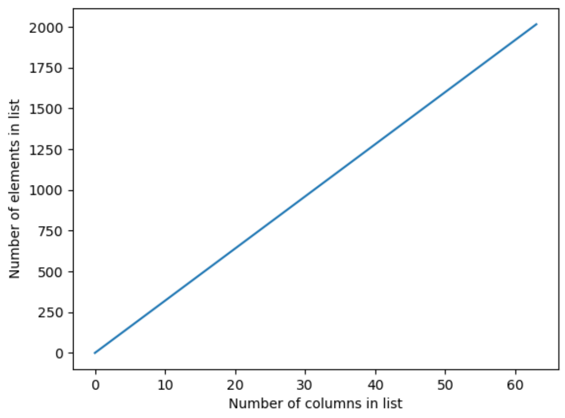
The modules used in the implementation of our project are:

- Python's random module and functions like **getrandbits** (which is used to get an n bit random number), **seed** and **sample**. The seed for our pseudo random number generator is the current time in its epoch format (which gives a float number, further converted into integers). This seed helps us make the numbers generated well-randomized. The **sample** function used, helps us to get a list/sample data of size n from a given iterable list. We use this function in order to get our initial sample of keys SP_i for the generation of tmto matrices.
- After the completion of offline/precomputation phase, when we generate the tmto matrices. In order to store these matrices efficiently and in native pythonic objects (i.e lists and dictionaries), we use the **pickle** module which is used to serialize python objects into

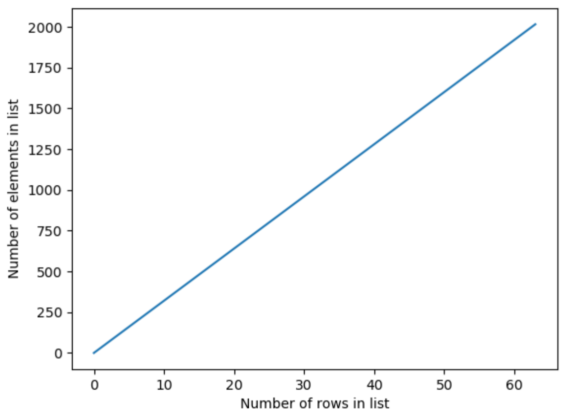
compressed binary data and deserialize back to python objects. Thus, the static storage of our generated tmto matrices is finally handled using the efficient **pickle** module.

5 Simulation Results

Based on the discussion in the previous section, we now present the complexities exploiting the TMTO attack. For the cryptosystem presented above, $N = 2 * m * t$, i.e. the storage search space used. It can be obtained by observing that there exist t lists with $2 * m$ elements each.



(a) N vs t



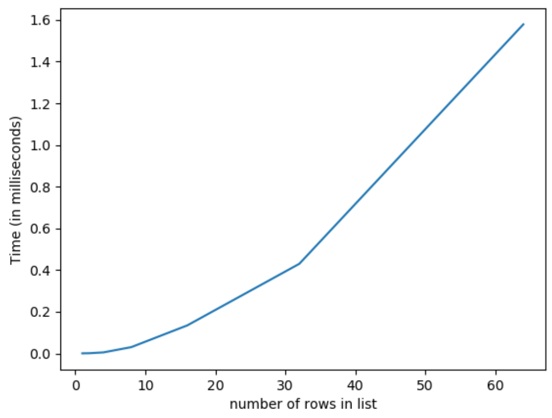
(b) N vs m

Figure 6: The above plots show the relationship of N with m and t respectively.

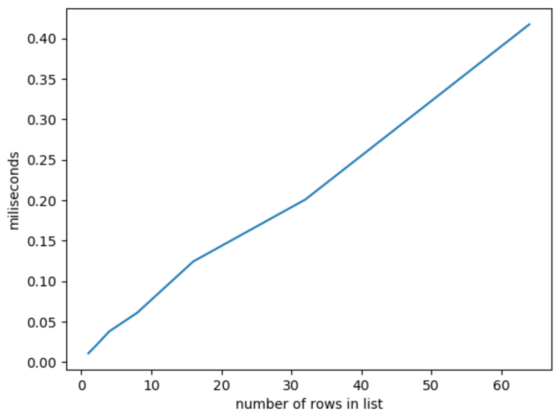
P, the time required in precomputation phase, can be given as

$$P = t^2 * m + m$$

.



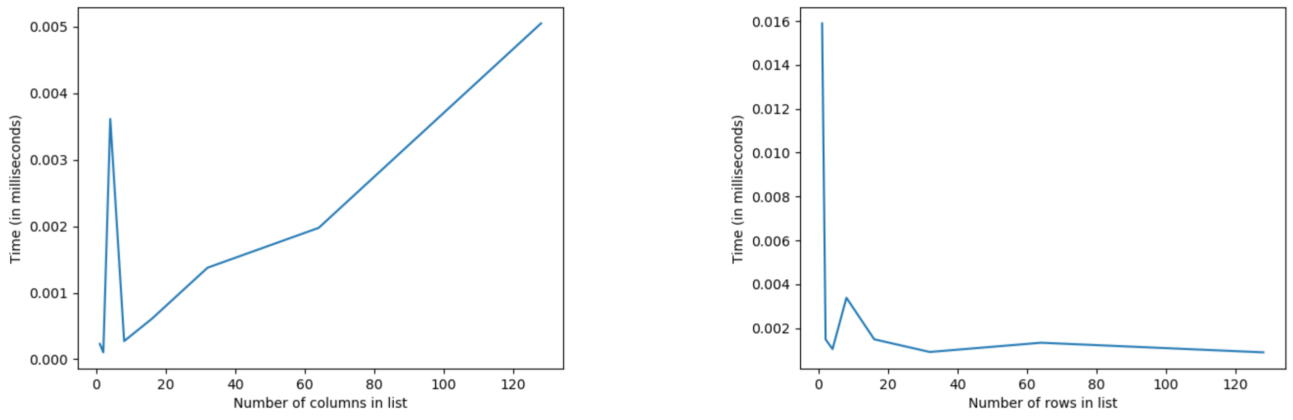
(a) P vs t



(b) P vs m

Figure 7: The above plots show the relationship of P with m and t respectively.

For T, the time required during the online phase we need to consider the collisions between $P \times C_i$ corresponding to k_i and $P \times C_j$ corresponding to k_j where $i \neq j$ as discussed in 4. Thus, here we see the best and worst case complexities.
The best case complexity being $T = t^2$ and the worst case complexity being $T = n * t^2$ where $n =$ number of blocks in known plaintext. A factor of $\log m$ cannot be seen here because python uses hash-tables to implement dictionary and search in hash-tables is $O(1)$.



(a) T vs t

(b) T vs m

Figure 8: The above plots show the relationship of T with m and t respectively.

The optimal m and t as obtained are 2^4, 2^4. Similar to that proposed by Hellman [2] where $m = N^{1/3}$ and $t = N^{1/3}$ here N is equal to $|K|$.

In our case, the search space is $N = 2 * m * t$ but essentially it amounts to $m * t^2$ as we check among all the elements in the t matrices theoretically. Thus, the search space in theory is $m * t^2$. For optimal performance, $|K| = m * t^2$, $P = t^2 * m$ and $T = t^2$. One may note that some factors such as time to sample m random values and sorting the lists on the basis of t^{th} row of corresponding matrix have been neglected as they do not contribute to the order.

6 Conclusions

This project has been a very informative project which has helped us learn about block ciphers and time-memory-tradeoff attacks. We show how tmto attacks are an efficient way of crypt-analysing block ciphers. We implement a novel lightweight block cipher based on the feistel network, by adding an extra layer of substitution box which helps ensure Shanon's property of confusion. Further, we present a novel way of mounting a time-memory-tradeoff attack on a cipher with a keyspace larger than ciphertext space. The implementation that we present for this problem is also quite efficient, both spacially and computationally and is a variant of the base time-memory-tradeoff attack proposed by Hellman [2] that covers a larger class of cryptosystems.

References

[1] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007* (P. Paillier and I. Verbauwhede, eds.), (Berlin, Heidelberg), pp. 450–466, Springer Berlin Heidelberg, 2007.

[2] M. Hellman, "A cryptanalytic time-memory trade-off," *IEEE Transactions on Information Theory*, vol. 26, no. 4, pp. 401–406, 1980.

[3] A. Biryukov, *Feistel Cipher*, pp. 455–455. Boston, MA: Springer US, 2011.

[4] M. Ubaidullah and Q. Makki, "A review on symmetric key encryption techniques in cryptography," *International Journal of Computer Applications*, vol. 147, pp. 43–48, 2016.

[5] S. Albermany and F. Radi, "Survey: Block cipher methods," vol. 5, 11 2016.

[6] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition.* Chapman Hall/CRC, 2nd ed., 2014.

[7] C. E. Shannon, "Communication theory of secrecy systems," *The Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[8] M. Luby and C. Rackoff, "How to construct pseudorandom permutations from pseudorandom functions," *SIAM Journal on Computing*, vol. 17, no. 2, pp. 373–386, 1988.

[9] A. Biryukov and A. Shamir, "Cryptanalytic time/memory/data tradeoffs for stream ciphers," in *Advances in Cryptology — ASIACRYPT 2000* (T. Okamoto, ed.), (Berlin, Heidelberg), pp. 1–13, Springer Berlin Heidelberg, 2000.