Algebraic Cryptography Project Report

Olga Zaghen

The purpose of this report is to discuss about the BunnyTN cipher and the Square Attack, to which the cipher is vulnerable, and their implementation in MAGMA. For both of them, the exposition is so structured: they are firstly introduced explaining the main ideas underlying them and the group implementation is discussed; after that my personal version is explained listing all the changes I made starting from the group project, and the reason why I made them. Then timing comparison between the two versions is performed. In conclusion of the report I inserted a brief digression on the possibility to extend the Attack to more than three rounds.

1 BunnyTN: implementation of the cipher

1.1 General procedure and group version

BunnyTN is a translation based block cipher that operates on plaintexts and master keys with integer values that vary from 0 to 2^{24} (excluded). Through the algorithm the state and the round keys are handled as elements of the vector space $\mathbb{F}_{2^6}^4$, and they can also be seen as vectors of $\mathbb{F}_2^{24}$ thanks to the isomorphism $\mathbb{F}_{2^6} \cong \mathbb{F}_2^6$, given by $\xi : \mathbb{F}_2^6 \longrightarrow \mathbb{F}_{2^6}, \xi(v_0, ..., v_5) = \sum_{i=0}^{5} v_i e^i$, where e is the primitive element of $\mathbb{F}_{2^6}$ considered in the cipher, root of the primitive polynomial $x^6 + x^4 + x^3 + x + 1 \in \mathbb{F}_2[x]$. The field $\mathbb{F}_{2^6}$ is also referred to as the "BunnyTN field"; furthermore, for the sake of simplicity, in the report I will often call "bricks" the four $\mathbb{F}_{2^6}$ elements of the state and of the round keys when they are expressed as vectors of $\mathbb{F}_{2^6}^4$.

The cipher performs fifteen rounds by default, each composed by an S-Box, a Mixing Layer and the sum with the round key. Whitening is also executed at the beginning before the first round, through the sum with the first round key. The Mixing Layer simply consists in the multiplication of the state vector with a particular matrix in $GL(4, \mathbb{F}_{2^6})$. The matrices for the Mixing Layer and its inverse were given explicitly in the BunnyTN paper. The S-Box consists of four parallel functions (that perform exponentiation and sum in $\mathbb{F}_{2^6}$), each one for a specific state vector brick. According to the project requirements, we implemented the procedure `KeySchedule`, which takes as input the master key `k` expressed as an integer, an empty sequence

K that has to be filled with the round keys expressed as vectors over the BunnyTN field, and an integer r which represents the number or rounds, since it could be different from the default. The other function to implement was BunnyTN, which accepts as input the integer values of the plaintext and the master key and gives back as output the computed ciphertext, expressed as integer by default. Additional parameters are accepted that permit to define a different number of rounds, perform decryption instead of encryption (the default), express the result as hexadecimal string and also directly give the sequence of round keys one wants to use, without the need for the function to compute them from the master key through the key schedule algorithm.

Let's talk about the design and implementation choices made in the group version of the project; then, starting from them, I'll explain how I came up with various observations in order to make changes and improvements and eventually develop a final personal version.

First of all, the only global variables we defined were the essential ones: Z is the ring of the integers, useful for various conversions; F64, which is the BunnyTN field with e as primitive element, is created as extension of GF(2) through the primitive polynomial f, which belongs to Q<x>, the defined polynomial ring over GF(2). There are also the matrix MAT and its inverse matrix InvMAT associated respectively to the Mixing Layer and its inverse: we thought it would be not efficient at all to define and create the matrices every time the Mixing Layer is executed and we decided to store them as global variables, since they're often used.

Let's analyze the BunnyTN function now. A check on the sequence of round keys is initially carried out: if it is not transmitted as additional input, then K is filled up invoking the KeySchedule procedure. After this, a control is performed on the relation between the number of round keys and the number of rounds of the cipher. In particular, if the number of rounds is greater than the number of round keys, a simple warning is printed out, since some errors may have occurred inserting the input, but the computation is still possible and hence it is done; besides, if the number of round keys is less than the number of rounds, then the calculation can't be performed: in this case a warning message is delivered, again, and the number of rounds is redefined as the number of round keys before proceeding.

After this step the plaintext, expressed as an integer in the input, is converted to a sequence of bits and then to a vector of four elements over the BunnyTN field. The vector structure is optimal for the state because it allows to efficiently perform two operations in each round: the sum with the round key vector and the Mixing Layer, which consists in the multiplication of the state with a particular matrix, and this can be done only if the state is a vector. Obviously the state could also be stored

with the structure of a sequence and the vector may be created only when needed, but the operation of converting from sequence to vector and vice-versa is quite expensive if executed frequently, and defining the plaintext as a vector from the start allows to perform the least number of conversions as possible (the only one needed in each round is the definition of the vector as output of the `Sbox` function). Furthermore, the plaintext is converted to a vector in $\mathbb{F}_{2^6}^4$ and not in $\mathbb{F}_2^{24}$ for quite obvious reasons: the round key is also expressed in $\mathbb{F}_{2^6}^4$ and additionally we found out, as I'll explain better later, that the most efficient way to perform both the S-box and Mixing Layer is through direct computation on vectors over $\mathbb{F}_{2^6}$. In general, our goal was to avoid performing too much conversions either between sequences and vectors, either between $\mathbb{F}_{2^6}$ and $\mathbb{F}_2^6$, in order to save time and computational power. Talking about conversions, in the entire project the one from $\mathbb{F}_{2^6}$ to $\mathbb{F}_2^6$ and vice-versa is carried out through the `"Eltseq"` command in one case and with the cast `"F64!"` in the other, while the passage from integer to sequence of bits and the inverse are performed respectively through `Intseq` and `Seqint`, hence the *little-endiannes* convention is adopted. It's important to notice that `Seqint` only works with sequences of integers, and this is why the cast `"Z!"` always needs to be performed on the bits of the sequence.

At this point an *if* clause is used to determine whether to perform encryption or decryption, where the encryption is composed by an initial whitening with a round key and a certain number of rounds in which the S-Box function, the Mixing Layer and the xor with the round key are performed; for what concerns the decryption, it carries out an initial whitening and then the rounds composed by the inverse Mixing Layer, the inverse S-Box function, and the xor with a round key. Obviously the whitening and the xor with the round keys in the decryption are executed taking the round keys contained in `K` from the last one to the first one.

Let's see now how the encryption works: the bitwise xor with the round key is easily performed through the sum of the vectors of elements in the BunnyTN field, thanks to the isomorphism $\mathbb{F}_{2^6} \cong \mathbb{F}_2^6$. Furthermore, in every round `Sbox` and `MixingLayer` are invoked: they are functions that take as input and return as output vectors over $\mathbb{F}_{2^6}$ and only execute the basic operations described in the specification of the cipher. `MixingLayer` only performs the product of the vector with the matrix `MAT`; the same reasoning goes for the decryption. For what concerns the S-Box, the BunnyTN paper only describes the encryption parallel functions, $\gamma(x_1, x_2, x_3, x_4) = (x_1^{62}, x_2^5, x_3^{17}, x_4^{62} + e^2)$ with $x_i \in \mathbb{F}_{2^6}$, while we had to compute the decryption ones. Here is the idea underlying our procedure: the first three parallel encryption functions are of the form $s(x) = x^n$ for a certain $x \in \mathbb{F}_{2^6}$ and $n \in \{1...63\}$, $gcd(n, 63) = 1$. The corresponding decryption ones are their inverses, and they can be expressed as $s^{-1}(x) = x^m$, with

m multiplicative inverse of n modulo 63 (it exists because $gcd(n, 63) = 1$), i.e. such that $n \cdot m = 1 \bmod 63$, and m is easily computed through the Extended Euclidean Algorithm. Since $a^{63} = 1$ for every $a \in (\mathbb{F}_{2^6})^*$ and $0^{63} = 0$, we have that $s^{-1} \circ s(x) = x^{n \cdot m \ mod \ 63} = x$ for every $x \in \mathbb{F}_{2^6}$. Observing with the Extended Euclidean Algorithm that 62 is the multiplicative inverse of 62 modulo 63, 38 is the multiplicative inverse of 5 modulo 63 and 26 is the multiplicative inverse of 17 modulo 63, the first three parallel decryption functions are so determined. The procedure followed to compute the inverse of the last parallel function is similar, indeed the encryption function is $s(x) = x^{62} + e^2$ and the corresponding decryption parallel function will be $s^{-1}(x) = (x + e^2)^{62}$ since $e^2 + e^2 = 0$ in $\mathbb{F}_{2^6}$.

One may think there could be more efficient alternatives to direct computation for the S-Box and the Mixing Layer: we reasoned about this a lot and I also did when developing my own personal version of the project. As a group, we came up with two alternative versions for the `Sbox` function, but they both were too slow compared to direct computation, and they can be found in the group project as comments. The "second version", as it is called in the code, makes use of a look-up table composed by four sequences, corresponding to the four parallel functions performed on the bricks of the vector state, and in the i-th position of every sequence is stored the result of the function executed on e^{i-1}, except for the last position where it is performed on the value 0. I also noticed that we made a mistake, since obviously $e^0 = e^{63} = 1$ and our look-up sequences contain both the result computed on e^0 and on e^{63} (indeed they are 65 positions long), while they should only contain one. However, the `Sbox` function works in this way: every element of the input vector is checked; if it is 0, the result is in the last position of the corresponding look-up sequence, otherwise the `Log` command is used in order to determine the right index where to find the result exploiting the fact that for every $i \in \{0..62\}$, $Log(e, e^i)$ returns i. For what concerns the "third version", it is the slowest one: it makes use of a look-up table made up of four sequences corresponding to the four parallel functions and in the i-th position they store the result of the operation performed on the integer `i-1` converted to an element of $\mathbb{F}_{2^6}$. A detailed description of this procedure will be given later while discussing my "alternative" version of the project.

After the execution of the rounds, the current state, which is the ciphertext, is converted to a sequence of bits (`B`) and then the corresponding integer is computed. At this point, if the condition `hex:=true` is an additional input of the function, then the integer is converted to a hexadecimal representation, otherwise nothing more is done.

Let's now analyze `KeySchedule`. In this procedure the computation is performed with the help of so called *words*, that are elements of the

BunnyTN field; the initial ones are obtained from the master key, and starting from them many others are computed, their number depending on the number of rounds of the cipher. They are eventually used to create the round keys. All the *words* are computed starting from the initial ones through simple operations in $\mathbb{F}_{2^6}$ and in $\mathbb{F}_2^6$ (the rotate left), hence a *word* is seen at the same time as an element of the BunnyTN field and a 6-bit sequence. This is not so immediate in the implementation and a choice should be made on the definition of the *words*: we decided to define them as $\mathbb{F}_{2^6}$ elements, because most of the operations can be easily performed in this way and just one conversion to a 6-bit sequence is needed in order to execute the rotation to the left.

`KeySchedule` is so structured: firstly the input integer which represents the master key is converted to a sequence of bits and then to a sequence of four elements in the BunnyTN field. Those four elements (`k2`) together with the four elements in the vector obtained applying `Sbox` to them (`k3`) are manipulated in order to generate the first eight *words*. The rest of the *words* is computed exactly in the way described in the BunnyTN paper. The operations performed are sum of already computed words, powers, sum with a given constant called `cost` and rotation to the left, for which a conversion to bit representation is needed, as stated before. `k1` and `k2` are sequences while `k3` has the structure of a vector just because it's an output of the `Sbox` function. `W` is also a sequence: its only purpose is to store elements (the *words*) that are required for the computation of round keys. When new *words* are defined they are stored in the same sequence `W`, which gets bigger as more elements are added. At the end of the procedure these *words* are considered four by four, as described in the specification paper, and a vector of length four over $\mathbb{F}_{2^6}$ (a new round key) is created out of them and it is added to `K`, that is filled up by the procedure with all the round keys.

1.2 Improvements given by my version

The changes put forward by my personal version (contained in `Olga_Zaghen.mag`) on the group one do not significantly change its structure and algorithms; they mostly concern the usage of variables, structures and functions that may be done slightly differently in order to gain in terms of computational power, time and memory. I made several attempts to develop more efficient algorithms than the ones used in the group project, but the point is that the round functions (the Mixing Layer, the S-Box and the xor with the round key) are so simple and fast that every try to perform them through look-up tables only slows down the execution and in some cases even makes it more complex. This is what I deduced from the outcomes of my attempts.

In the group version we made two tries (written as comments) to use look-up tables instead of direct computation for the S-Box and unfortunately they were both too slow. This may be caused by the operations and conversions made in order to come up with indexes needed for those look-up tables, since the elements of $\mathbb{F}_{2^6}$ can't be directly used as indexes. This observation made me initially think of a completely different implementation of the cipher, dealing with sequences of bits instead of elements of $\mathbb{F}_{2^6}$. In this way, the look-up table addressed to as "third version of the S-Box" in the group project could be more efficiently used, since the elements of the field are already expressed as 6-bit sequences. This alternative version I firstly developed is in the file `Olga_Zaghen_Alternative.mag` and I won't describe it in detail, since I then found out that its performance in time terms is much worse than both the group version and my final personal version; the only point I want to stress with it is that using look-up tables for the S-Box and Mixing Layer doesn't speed up the computation neither if we deal with vectors over the BunnyTN field nor if we express the state as a sequence of bits.

I will talk now about the changes performed on the group version of the project in order to obtain my final personal version `Olga_Zaghen.mag`. After that, I'll give a brief description of `Olga_Zaghen_Alternative.mag`.

First of all, I decided to delete the functions `MixingLayer` and `Sbox` (and their inverses) and to perform the computation directly in the rounds of `BunnyTN`. I made this change because every time a function is invoked some memory space needs to be occupied and additional computation is performed in order for the function to execute. Taking this into account, it seemed to me not convenient at all to define and invoke a function to perform just one simple operation. Obviously this implies that every time `Sbox` and `MixingLayer` (and their inverses) were invoked in the group project now they have to be computed directly, for example in the `KeySchedule` function and in the implementation of the Square Attack.

Let's list now all the changes performed in the `KeySchedule` function, which is the part I modified the most:

- First of all, the line `k3 := Sbox(k2)` has been removed, and the main reason is that `Sbox` is not used anymore. Also, `k3` represented a waste of memory space and computation, since only its elements are necessary for the procedure and there is no need to create a vector in order to store them. This is why in my version the direct computation of the parallel S-Box functions is performed in the definition of `W` with the elements of `k2`.

- The constant `cost` is defined differently and I made this choice because its value is known and fixed, and never changes. This implies

that we can precompute the value of `cost` in $\mathbb{F}_{2^6}$ and always define it like that, because we need it expressed in the BunnyTN field for the computation. This avoids the necessity to perform a cast from sequence of bits to element of $\mathbb{F}_{2^6}$, which may be quite expensive if performed all the times `KeySchedule` is invoked.

- The last change performed in this function is the elimination of the variable `a`: even though it is just one variable, is it completely useless and it only contributes to wasting memory and computational power.

For what concerns the `BunnyTN` function, no substantial changes were made. First of all, I decided to delete both of the warning alerts: in case the number of rounds is greater than the number of round keys, there must necessarily be an error in the inputs, and in this case it is better to stop the execution of the program and alert the user with an error message displayed by MAGMA itself instead of modifying the number of rounds and performing the computation, which may be useless because of the wrong inputs. I also deleted the one that displayed a warning message in case the number of rounds was less than the number of round keys. Indeed this set of inputs may be intentional: `BunnyTN` with less rounds than the number of round keys may be invoked for some purpose. That's why the warning message might be disturbing, and, in any case, it is in general not necessary because the computation can still be performed. Another change I made is something I've already talked about: the S-box and Mixing Layer operations are directly performed in the rounds, invoking no additional external functions. The last thing which differs from the group project is that I decided to delete, at the end of the function, the sequence `B`: its values are used only for the computation of `c`, which can be accomplished without storing them in `B`.

Now that my personal final version has been described, let's discuss `Olga_Zaghen_Alternative`, that I developed in order to verify if the time performance can get better if certain look-up tables are used and state and round keys are expressed as sequences of bits (to be more "compatible" with such look up tables). The differences with respect to my final version `Olga_Zaghen` are the following: first of all, `KeySchedule` is the same, except for the fact that when computing the output, it does not fill `K` with vectors of $\mathbb{F}_{2^6}^4$, but it expresses them as vectors of 24 bits. This implies that also `BunnyTN` receives as input a sequence `K` with round keys of that form. What also changes in `BunnyTN` is that `P` is expressed as a vector of 24 bits (as the round keys are), so that the bitwise xor can be performed between keys and plaintext, and also the `Sbox` and `MixingLayer` functions (and their inverses) accept as input and give back as output 24-bit vectors. Nevertheless, the general structure of the function remains the same. The most substantial

changes can be seen in the `Sbox` and `MixingLayer` functions, implemented with look-up tables.

For what concerns `Sbox`, so as its inverse, the adopted mechanism is the same as in the "third version of S-Box" of the group project: the look-up table is composed by the four sequences `f[1], f[2], f[3], f[4]` of 64 elements each, and in the i-th position of the j-th sequence it is stored the result of the j-th S-Box parallel function performed on the element of $\mathbb{F}_{2^6}$ obtained by converting the integer `i-1` to a 6-bit sequence and then to an element of $\mathbb{F}_{2^6}$. The result is stored as a 6-bit sequence, and the same is done for the inverse look-up table. When the `Sbox` function is invoked, the 24-bit input is considered as the union of four 6-bit sequences; each of these sequences is converted to integer, and this value incremented by 1 is the index in the proper sequence `f[j]` where the result of the j-th parallel function applied to the $\mathbb{F}_{2^6}$ element corresponding to the 6-bit input sequence is stored. After the four results expressed as sequences of bits are collected, they're glued together and a 24-bit vector is created. The same procedure is adopted by `InvSbox`, but of course with the inverse look-up table.

This version of the `Sbox` can be efficiently used only if the state is expressed as a vector of bits because otherwise, if the S-Box function is invoked on vectors of four $\mathbb{F}_{2^6}$ elements, an additional initial conversion from $\mathbb{F}_{2^6}$ to sequence of bits is to be performed, as it was done in the "third version of S-Box" of our group project, and this cast slows down the function.

Let's now talk about the `MixingLayer` function. The main underlying idea is the following: the result of a linear function applied to a certain linear combination of initial vectors is equal to the same linear combination of the results of the linear function applied to the initial vectors. This fact can be exploited in the following way: we could compute and store in a look-up table the results of the Mixing Layer function (which is linear) applied to a basis of $\mathbb{F}_2^{24}$, and the basis I chose is the set of 24 24-bit vectors of Hamming Weight equal to 1. The results are so computed: the 24-bit vector of the basis is divided in four 6-bit sequences, each one is converted to an $\mathbb{F}_{2^6}$ element and the obtained vector is then multiplied for the matrix. These results are also converted and stored as 24-bit vectors, and the look up table is a sequence of length 24 in which in the i-th position the result of the Mixing Layer applied to the element of the basis with the bit 1 in the i-th position is stored. At this point, when the `MixingLayer` function is invoked on a 24-bit vector, the vectors of the basis whose sum is equal to the input vector are considered and the results of the Mixing Layer applied to them are collected from the look up table; these results are then bitwise xored. In this way, according to the observation on linear functions made above, the result obtained from the sum will correspond to the Mixing Layer

applied to the input vector. The particular basis is chosen in order to make this process as fast and efficient as possible: indeed, if the i-th element of the input vector is 1 then the i-th element of the look-up table is to be considered for the sum. The same procedure is adopted for constructing the inverse function.

In the context of states expressed as vectors of bits, this algorithm is certainly more efficient than converting every time a 24-bit vector to a vector of four $\mathbb{F}_{2^6}$ elements and then performing the product with the given Mixing Layer matrix. Obviously, if the state is already expressed as a vector of length four over the BunnyTN field, it is much faster to compute the product with the matrix directly instead of converting it to a vector of bits and then using such look-up table. The conversion to bits slows down the computation.

A solution may be to implement look-up tables directly usable with BunnyTN field elements without the need to convert them to sequences of bits. The second version of the S-Box proposed in the group project tries to do it, but it's still slower than direct computation. I also made some attempts for my personal project, and I came up with an idea for the S-Box and one for the Mixing Layer. I tested their performances and they resulted to be again too slow compared to direct computation, hence I didn't write them down in my personal project but I'll give now a brief explanation of their structure.

The S-Box's look-up table is made up of four sequences of length 64. Each one contains in the i-th position, with i ranging from 1 to 63, the result of one of the parallel S-Box functions performed on e^i, and in the 64th position the result of the operation performed on 0. The `Sbox` function, when invoked, can make use of four *case* statements, each for a different brick of the input vector: if we consider the first brick, referred to as `a`, and the first sequence of the look-up table named `f`, the first *case* statement will be `" case a: when 0: return f[64]; when e^ 1: return f[1]; when e^ 2: return f[2] ..."` until `e^ 63`. Instead of using *case* statements, the control can be performed with a *for* cycle through all the sequence `f` of the look-up table corresponding to a particular brick `a` of the vector: `"(for i in [1..63]...) if e^ i eq a then result := f[i]"`. A check on whether the element is equal to 0 is also needed.

For what concerns the Mixing Layer function, it is even more difficult to find suitable end efficient look-up tables for vectors over $\mathbb{F}_{2^6}$, because a look-up table should be constructed according to the whole vector and not the single elements. I thought of adapting to $\mathbb{F}_{2^6}^4$ the method followed in the "alternative" version of the project, exploiting the idea of linearity underlying it. The considered look-up table is made up of four sequences of length 64. In the first sequence the result of the Mixing Layer applied to all

vectors of the form `Vector([`e^i`, 0, 0, 0])`, with i varying from 1 to 63, is stored, while in the last position is it applied to `Vector([0, 0, 0, 0])`. In the second sequence the same is done for vectors of the form `Vector([0,` e^i`, 0, 0])`, and so on. The computation of `MixingLayer` is performed in this way: let's suppose `a` is the first brick of the input vector. From the first sequence of the look-up table, the resulting vector corresponding to the Mixing Layer executed on `Vector([a, 0, 0, 0])` is taken with a *case* statement. After this is done for every brick of the input vector and four new vectors have been collected from the look-up sequences, the sum of these four is computed and the result is obtained. Although these two mentioned methods may seem quite good, they are still slower than direct calculation. The long discussion made about the low efficiency of look-up tables for S-Box and Mixing Layer aims to justify the choice of not using them in my personal project.

1.3 Time comparison

Let's now compare the timing of the three versions, in order to show the improvement of my final personal version with respect to the group one, and the fact that the alternative version is not efficient. The processor of the computer used is AMD Quad-Core Processor A8-6410, while the memory is 3,3 GiB. The code used to get the time values is the following:

```
1: time
2: for i in [1..2^n] do
3:     k := BunnyTN(i, i);
4: end for;
```

Pseudocode 1: CPU TIME COMPUTATION

The results printed out by MAGMA, with n ranging from 9 to 15, are the following and they show an improvement of my personal version with respect to the group one, making an average of all the results, of about 9.1%.

n	Group version	Personal version	Alternative version
9	0.410	0.360	0.990
10	0.780	0.710	1.940
11	1.570	1.420	3.810
12	3.150	2.870	7.700
13	6.190	5.690	15.200
14	12.670	11.440	30.950
15	24.600	22.960	61.060

Table 1: Time comparison for the BunnyTN cipher

2 The square attack on BunnyTN

2.1 General procedure and group version

We performed the Square Attack on the cipher, which is a chosen plaintext type of attack aiming to recover the third round key used in three-round version of the cipher. The underlying idea exploited by the attack is that the translation based block ciphers structured as BunnyTN grant a particular property, called *sum zero property*: I'll explain it in the context of the BunnyTN cipher. If we choose a set of 2^6 plaintexts of which the first brick assumes all the possible values in $\mathbb{F}_{2^6}$ while the other bricks have all the same constant value $c \in \mathbb{F}_{2^6}$ and we compute the first two rounds of the cipher starting from them, we obtain that at this exact point if we sum the corresponding bricks of every partial ciphertext then the sum is zero for every brick. This is the *sum zero property*, which can be exploited by an attacker in this way: he may choose a set of plaintexts as stated above, compute their ciphertexts with the three-round cipher (which is done in our case with the `BunnyTN_BlackBox` function) and, starting from them, perform a partial decryption until the end of the second round, where the *sum zero property* should hold if the correct third round key has been used. The purpose is, indeed, to guess the third round key: during the partial decryption all the possible values for the bricks of the round key are tested and if they allow the *sum zero property* to be satisfied, they'll be considered valid candidates for the bricks. In order to select the correct round key among all the candidates, another attempt is performed with an alternative set of chosen plaintexts (where the constant bricks are assigned a different $\mathbb{F}_{2^6}$ value): a new set of candidates is obtained, and the intersection of this set and the previous one is taken. This is done until there is just one candidate for the round key.

A clarification should be made on how the decryption of the third round is executed: performing the sum with the round key first and then the inverse Mixing Layer and the inverse S-Box is not the best choice in terms of time and computation cost. Indeed, in this case we should compute the inverse Mixing Layer of every ciphertext xored with all the possible candidate round keys, which is expensive. What we did instead was to perform the inverse Mixing Layer on the ciphertexts first, and only then try to guess the bricks of the round key and verify if they are the right ones, i. e. if the *sum zero property* is satisfied at the beginning of the third round. Obviously, what is guessed in this case is not directly the third round key but the third round key to which the inverse Mixing Layer has been applied. This method is more efficient than standard round decryption since the inverse Mixing Layer is executed less times.

After this computation is done with various sets of chosen plaintexts,

until one single candidate is left, the third round key is easily found applying the Mixing Layer to that candidate. From now on in this section I will refer with the word "pseudokey" to the third round key on which the inverse of the Mixing Layer is performed, since this is the element we try to uniquely determine. In the comments of the project code it is often referred to as `"K3'"`, while the third round key is `"K3"`. Once the pseudokey is found, if Mixing Layer is applied to it then the third round key is obtained.

Let's now discuss the group implementation of the attack. In the project the two functions `Square_BunnyTN` and `BunnyTN_BlackBox` are defined and used, as it was required by the assignment. Since they strictly depend on the master key one wants to attack, the key `K` is firstly defined as a global variable (this was required, too). `BunnyTN_Blackbox` processes the plaintext received as input with the three-round BunnyTN cipher, invoking the `BunnyTN` function and using `K` as the master key. Since the key is a global variable and the definition of `BunnyTN_Blackbox` depends on it, also the various round keys don't change, and that's why we decided to precompute the sequence of round keys `K_session` and always give it as an additional input to the cipher function. In this way the sequence of round keys does not need to be computed afresh every time `BunnyTN_Blackbox` is invoked. Let's now describe `Square_BunnyTN`: in order to understand how it works, it's necessary to introduce first the structures `EVE` and `K_prob` and the function `SingleSquareAttack`, since they are essential elements.

`K_prob` is a sequence of four sets such that in the i-th set the candidate i-th bricks for the pseudokey are contained. The performance of the Square Attack with a single set of chosen plaintexts determines a `K_prob` structure which may have more than one possibility for every brick. The idea is, for each brick, to perform the intersection between the sets of candidates obtained through computation from different sets of chosen plaintexts, until each brick is uniquely determined. At this point the wanted pseudokey is found, and from that the third round key is obtained. The candidate bricks are contained in sets because we need a malleable structure that allows to add elements and remove elements easily. These four sets are collected in a sequence: this permits to quickly determine which brick of the pseudokey the various candidates refer to, and they are easily handled.

`EVE` is a sequence of at most four elements, which may be 1, 2, 3 and 4, aiming to keep track of which bricks of the pseudokey have already been uniquely determined: at the beginning `EVE = [1, 2, 3, 4]` and when the i-th brick is fixed then number `i` is excluded from the sequence.

About `SingleSquareAttack`, it is a function that accepts as input a constant `c` and the sequence `EVE`. The purpose of this function is to compute a sequence of sets of possible bricks for the pseudokey through the Square Attack performed on a single set of chosen plaintexts (from this fact comes

the "Single" in the name of the function). With the help of `EVE`, this is done only for the bricks which have not been determined yet: the sets corresponding to the already computed bricks are left empty. This sequence of sets, which is again called `K_prob`, is then returned as output. We decided to define this function because it contains a set of operations that are performed multiple times, and invoking this external function to execute them makes the structure of `Square_BunnyTN` more elegant and clear to understand.

`SingleSquareAttack` works in this way: firstly the empty `K_prob` sequence is created, and after that the set of ciphertexts `C` is computed through the `BunnyTN_BlackBox` function, starting from the chosen plaintexts which have the first brick that varies in $\mathbb{F}_{2^6}$ and the other three equal to the variable `c` received as input. The plaintexts are created choosing the four bricks in $\mathbb{F}_{2^6}$ first, converting them to sequences of bits and then to integers, as they are required by `BunnyTN_BlackBox`. After this is done, the inverse Mixing Layer of every obtained ciphertext is computed and stored in a sequence called `Ever` (to do this, a conversion is carried out to express the ciphertexts as vectors over the BunnyTN field and they are stored in the sequence `Ev`). At this point `EVE` establishes the pseudokey bricks which are not known yet and their possible values are computed: for each brick every element of $\mathbb{F}_{2^6}$ (that is, every value the pseudokey brick could assume) is summed with the corresponding brick of all the 64 elements contained in `Ever` and, for every value tested as pseudokey brick, the 64 obtained results are stored in the sequence `w`. After that, a parallel function of the inverse S-Box is performed on the elements of `w`. `InvSbox` is not invoked because the parallel functions are executed independently on single bricks. The result is then stored in the sequence `v`. For each value tested for a particular pseudokey brick, it is now checked if the *sum zero property* is satisfied at the beginning of the third round, and this is done by verifying whether the sum of all the elements contained in `v` (the i-th brick of all partial ciphertexts for some i) is zero; if the equality holds, then the tested value for the pseudokey brick is inserted in the appropriate set of the `K_prob` sequence. At the end of this procedure, the resulting sequence contains empty sets corresponding to the indexes not in `EVE` and the computed candidate bricks in the others.

All the elements introduced so far are essential for the `Square_BunnyTN` function, which accepts no inputs and gives back as output the attacked third round key. The idea of the function is the following: `EVE` is firstly created, and the first sequence `K_prob` is computed through `SingleSquareAttack` with the constant 0 and `EVE` as input. At this point a sequence `cost` is defined containing all the $\mathbb{F}_{2^6}$ values: it is used to keep track of the constants which have already been used as inputs for the `SingleSquareAttack` func-

tion. The reason why it is useful to store those values in a data structure is that otherwise we would not know which constants have already been utilized. In this way it suffices to remove an element from the sequence once it has been used, as it is immediately done with the zero value. Afterwards `EVE` is updated according to whether some bricks of the pseudokey have been determined already, and then if `EVE` is not empty another sequence of sets of possible bricks, `K_prob1`, is created and computed using `SingleSquareAttack` with the next available constant in `cost`, which is deleted from the sequence immediately after. `K_prob` is then redefined as the intersection between `K_prob1` and `K_prob` itself. After this, `EVE` is updated again, and the previously described operations, i.e. update and check of `EVE`, execution of `SingleSquareAttack` and intersection of `K_prob1` and `K_prob`, are performed cyclically until the pseudokey is univocally determined (i.e. when `EVE` is empty). This is done through a *while* cycle which keeps track of the number of elements in `EVE`. Eventually the Mixing Layer function is applied to the pseudokey and the third round key is obtained. This happens with the help of the sequence `e`, which contains the four elements of the pseudokey, obtained from the `K_prob` sets by casting them to sequences and extracting their first elements (this would not be possible with sets, because they are not indexed). After this is converted to a vector, the `MixingLayer` function is applied and the result is stored in `key_long`, to be converted again to a sequence of bits in `key_long1` and then processed to get the integer representation of the result. All these sequences used to store partial results of the computation are not necessary and indeed they have been removed in my personal version of the project.

2.2 Improvements given by my version

The first obvious change is the fact that, since the `InvMixingLayer` and `MixingLayer` functions do not exist anymore, their previous use is substituted by direct computation of the results.

The main change I performed is the distinction between the function `FirstSingleSquareAttack`, which is executed first, and the procedure `SingleSquareAttack`, used just in case the "first" one is not sufficient to determine the candidate pseudokey. In the group project a single function named `SingleSquareAttack` is invoked in both cases. Another important change is that the `EVE` sequence is not used anymore, because I think it is not strictly necessary. In fact, its only purpose is to allow keeping track of which bricks of the pseudokey have not been determined yet, but this can be easily achieved just by checking the cardinalities of the sets of `K_prob`. Also, `EVE` constantly needs to be updated and modified, and this takes to pointless waste of time and computational power.

What led me to the first change is the observation that the `SingleSquare-`

`Attack` function of the group project acts in this fashion: it accepts as input a constant for the set of plaintexts and the sequence `EVE`, and it gives back as output a sequence of sets of all the candidate pseudokey bricks, and these are selected testing all the possible $\mathbb{F}_{2^6}$ values (64 elements for each brick). The so-obtained sequences of candidates are then compared and their intersection is computed. The point is that, even though the procedure of this function makes sense for the first set of chosen plaintexts, it does not for the next ones, since we know that the right bricks of the pseudokey are already contained in the `K_prob` given as output by the first invocation of the function. Hence it is not necessary to test all the 64 possible values for each pseudokey brick suggested by `EVE` every time `SingleSquareAttack` is performed: it is sufficient to do it during the first execution, and after that only the candidate bricks of the already computed `K_prob` should be tested.

I had the idea to distinguish between the computation done with the first set of plaintexts and the the rest of them. The easiest way to do this was to implement two different algorithms:

- `FirstSingleSquareAttack` is a function: it accepts as input `c` which is the constant used to construct the set of chosen plaintexts, and its output is a sequence of sets of the candidate bricks for the pseudokey. This function is only used for the computation with the first set of plaintexts, and there is no check on which brick to work on (as it was done in the group project with `EVE`): it is obvious that none of the bricks has been determined yet and so the computation is to be done for all of them. The possible values for every brick are all the elements in $\mathbb{F}_{2^6}$ with no constraints. This function is very similar to the group project function, except for some details that will be listed afterwards, and for the fact that, as I already said, `EVE` is not used anymore.

- `SingleSquareAttack` is a procedure: it has no output and accepts as input a constant `c`, used to determine the set of chosen plaintexts, and the sequence `K_prob` which is initially computed by `FirstSingleSquareAttack` and then modified every time this procedure is executed. The procedure is similar to the function described above, except for the fact that only the essential computation is carried out: for every brick, a check on `K_prob` is performed to know if it still has to be determined (that is, if the set corresponding to that brick contains more than one element). If a brick needs to be determined, the only key bricks tested during the computation are the ones already contained in `K_prob`: in practice, the condition `for key in` $\mathbb{F}_{2^6}$ in the group version is now substituted by `for key in K_prob[i]`. Otherwise, if the set of the corresponding brick contains

one single element, nothing is done. In this way no useless computation is performed. The result of testing the elements in `K_prob` on whether they grant the *sum zero property* allows to modify `K_prob` itself, deleting a candidate brick in case the property is not satisfied, and this is different from what `FirstSingleSquareAttack` does.

The way `SingleSquareAttack` is structured not only avoids unnecessary computation in the procedure itself, but also allows to determine the pseudokey working directly and only on `K_prob`. This means that there is no need to create a new sequence of possible bricks every time a single Square Attack is executed (`K_prob1` is not used anymore) and hence the intersection of sets of those sequences has not to be computed anymore. This change leads to save time, computational power and also memory space. An observation should be made about `FirstSingleSquareAttack` and `SingleSquareAttack`: they are auxiliary functions, not required by the assignment and not strictly necessary, since their operations could be directly carried out by `Square_BunnyTN`; nevertheless, they are very useful because they collect a set of tasks that can be seen as a unique and independent block, and they help in making the structure of the attack clearer.

Let's go on analyzing these two functions and pointing out the performed changes with respect to the group version of `SingleSquareAttack`. First of all, I defined a global variable `W`, used in both of the functions: it's a sequence in which the bit representation (with 0 and 1 seen as elements of $\mathbb{Z}$) of all the elements in $\mathbb{F}_{2^6}$ is determined and stored. These bit sequences are used in order to generate the sets of chosen plaintexts for the attack, and in particular they correspond to the first brick of the plaintexts. The computation of these elements was originally performed directly in the group function `SingleSquareAttack` every time it was invoked, in order to generate the set of plaintexts and express them as bit sequences. I thought it may be better to create the sequences once and then set `W` as a global variable, instead of performing the cast of all those elements to sequence of bits and then to integers every time. All these conversions may be quite expensive in terms of time if executed frequently. Although this choice requires to occupy part of the memory permanently, it allows to save time and computation.

Going ahead in the description of the two functions, the sequence `Ev` has been deleted because not strictly necessary; indeed, it was only used for the creation of `Ever`, which is now directly computed without storing partial results of the process. This leads to less waste of memory. Furthermore, I performed some changes in the *for* cycle that aims to create or modify `K_prob`. First of all, I substituted the *for* clause "`for i in EVE do `" of the group version with a combination of a *for* clause "`for i in [1..4]`" and an *if* clause "`if #K_prob[i] gt 1 then`", because `EVE` is not used

anymore. In addition to this, I deleted both the sequences `w` and `v`, because useless: the calculation for which they were used can be performed also without them. Indeed, `w` stored the brick of every element in `Ever` summed with the candidate brick of the pseudokey, and `v` consisted in the result of a particular parallel function of the inverse S-Box applied to the elements of `w`. `w` was only used for that purpose, which clearly makes it unnecessary because `v` could be directly computed; besides, `v` is only needed to verify if the sum of all its elements is equal to 0. That's why I defined only a variable `v`, which is not a sequence anymore but an element of $\mathbb{F}_{2^6}$: it initially has the value 0, and every time the inverse S-Box parallel function is computed on an element of `Ever` summed with the candidate pseudokey brick, the result is added to the current value of `v`. The final value of this variable `v` corresponds to the sum of the elements of the old vector `v`, and hence only this final value is needed to check the *sum zero property* and determine what to do with the candidate brick of the pseudokey: in the case of `FirstSingleSquareAttack` it is added to `K_prob` if the property is satisfied, while in `SingleSquareAttack` it is removed from the sequence if the *sum zero property* is not granted.

The last change I made is in the way the inverse S-Box is performed: in the group project a time and computation consuming method was adopted, which consisted in the fact that every time an element of `w` was considered in the *for* cycle on the index `l`, a check on the variable `i` being 1, 2, 3 or 4 was performed. This was pointless: once a certain `w` is considered, we know that all of its elements consist in bricks of the same index, hence with the same `i` (in my version the vector `w` is not used, but the reasoning is the same). This means that it makes more sense to carry out a check on the `i` value first, that will be the same for all the elements of `Ever` to be considered, and only then perform a *for* cycle with all the needed computation. I did this with a *case* statement that, according to the `i` value, determines which *for* cycle corresponds to the calculation that needs to be executed.

Let's now talk about `Square_BunnyTN` and the main structures and procedures I decided to remove or redefine and that caused a waste of time, memory and/or computation:

- `EVE`, as explained above: its presence is not necessary and its purpose can be easily performed through checks on the sets in `K_prob`. Indeed, a certain brick of the pseudokey has not been determined yet if the cardinality of the corresponding set in `K_prob` is greater than 1. This kind of check substitutes the use of `EVE`: in `Square_BunnyTN` the condition `"#EVE gt 0"` is replaced by `"(#K_prob[1] gt 1) or (#K_prob[2] gt 1) or (#K_prob[3] gt 1) or (#K_prob[4] gt 1)"`, while in `SingleSquareAttack`, I substituted the for clause `"for i in EVE do "` of the group version with a combination of a for

clause "`for i in [1..4]`" and an if clause "`if #K_prob[i] gt 1 then`". The removal of `EVE` leads to less waste of memory, but most importantly to less waste of time and computation, since it constantly has to be updated and modified and there's no point in doing it since its job can be carried out simply with `K_prob`.

- At the end of `Square_BunnyTN` I removed the sequence `e` because unnecessary, since it is only used for the creation of `key_long` that comes next, which can be directly computed. The sequence `key_long1` has been deleted too, because it is unnecessary and it leads to a waste of computation, since its construction makes use of the `cat` command. Only its elements are needed to perform the calculation and they don't have to be stored in order to obtain the output `n`: this can be easily accomplished using `key_long` only. In this part of the code I made a further change in the definition of `key_long`. The conversion from set to sequence performed in the group version to be able to get the elements contained in `K_prob` easily through the indexes of the sequences is not truly necessary and also quite expensive. An alternative way to extract the four bricks from their sets may be to use for example the `Max` command: we can't refer to elements in sets with indexes as it is done in sequences, but since every set contains just one single element (the brick of the guessed pseudokey) it also coincides with the maximal element of the set.

- Since it is usually sufficient to execute the Single Square Attack on different sets of chosen plaintexts for less than three times in order to uniquely determine the pseudokey, it's not necessary to store in the sequence `cost` all the values of $\mathbb{F}_{2^6}$: only few are needed. I decided to store in it only six different constants, and 0 is not included: this avoids the necessity to exclude it from the sequence after its creation, as it was done in the group version. This allows to save computational power, while the fact that `cost` is shorter grants less memory waste.

Additionally to the last point, I made sure that the `cost` sequence is created only if after the execution of `FirstSingleSquareAttack` the possible values for the four bricks of the pseudokey are not uniquely determined yet. In order to do this the cardinality of the sets in `K_prob` is checked. The same holds for the index `i`, which is defined only if `cost` is, for the same reason above. The conditioned creation of `cost` and `i` is put in place through an *if* clause. This avoids waste of memory and computation in case the pseudokey has been determined already.

2.3 Time comparison

For the evidence on time improvement given by my version, I executed in MAGMA the following algorithm:

```
1: t := Cputime();
2: for K in [1..2^n] do
3:     K_session := [];
4:     (part of project code from the definition of K_session to the end of
       the definition of Square_BunnyTN: for the personal project W is
       not defined every time, but just once before executing this for cycle)
5: end for;
6: Cputime(t);
```

Pseudocode 2: CPU TIME COMPUTATION

For n ranging from 12 to 18, the values assumed by `Cputime(t)` are the following and they show an improvement in time of about 7.70%:

n	Group version	Personal version
12	0.700	0.620
13	1.340	1.210
14	2.630	2.400
15	5.180	4.780
16	10.240	9.430
17	20.630	18.920
18	41.190	38.080

Table 2: Timing comparison for the Square Attack

3 Digression about attacks on BunnyTN

The last part of our project required us to think about whether an extension of the Square Attack on BunnyTN could be executed on more than three rounds. We verified that an extension of the classical Square Attack on four rounds can't be performed (as it is in AES), and this simply derives from the *sum zero property* not being satisfied by the set of partial ciphertexts at the end of the third round. In the final part of the project file we wrote down two algorithms to prove that *sum zero property* is granted at the end of the second round but not of the third.

What is more interesting is verifying whether a Partial Sum Attack could be performed on BunnyTN, as it is done on 6-round AES. From various observations we came to the conclusion that with high probability the attack doesn't work. The point is that what makes AES vulnerable

to the attack is the weakness of its Mixing Layer or, to be more precise, the fact that if we consider a certain active byte in the state matrix before applying Shift Rows and Mix Columns we can see that, after applying them, only the bytes of its same column will depend on the active byte, while it will have no influence on the other columns. This is the property that allows the attacker to force only four bytes at a time of the fifth round key (plus one of the fourth round key) instead of having to guess the bytes of the fifth round key all together. This fact also makes it possible to add the so-called round zero and extend the attack to six rounds (from the zeroth to the fifth) instead of just five.

The described property of the Mixing Layer is not satisfied by the BunnyTN cipher: indeed, in this case the plaintext is split into four bricks and if we focus on a particular active brick of the state vector we see that after multiplying it with the Mixing Layer matrix it influences all the four elements of the resulting vector. We deduced it from two facts: firstly by observing the matrix, since it doesn't seem to have a particular structure that may assure the property above or a similar symmetry (the only interesting thing we observed is the fact that the sum of the elements of every column is equal to zero); secondly, we tried to show in practice that the matrix grants high diffusion. We considered the matrix used for the BunnyTN encryption and we multiplied it with several vectors of a particular form (e.g. with just one brick different from zero and the other three bricks equal to zero or to some constant), in order to understand if the result of the operation had a particularly relevant structure, too (for example if one of the four elements remained unchanged or changed in a predictable way according to the input). Every test showed that any brick of the resulting vector changed with the performed operation in an apparently random way, and hence this proofs that starting with just one active element, all the elements result to be influenced by it after applying the Mixing Layer function. I would say it may imply that the BunnyTN's Mixing Layer is strongly proper. Another important fact to take into account is that the considered matrix is *Maximum Distance Separable*, which assures high diffusion between the bricks of the message.

Let's suppose we decide to consider an active brick at the beginning of the third round where the *sum zero property* is satisfied, and we want to exploit this fact by guessing the active (influenced by the chosen active brick of the state) brick of the third round key and the active ones of the fourth round key. It turns out that at the beginning of the fourth round all the bricks of the state are influenced, and so the entire fourth round key needs to be forced, and also at least one brick of the third round key: this is more expensive than just trying to guess the master key with a brute force attack.