Università degli Studi di Milano

Doctor of Philosophy in Computer Science
Cycle XXXV

Department of Computer Science "Giovanni degli Antoni"

Strength evaluation of cryptographic primitives to linear, differential and algebraic attacks

R18

Candidate: Sergio Polese

Supervisor: Prof. Andrea Visconti

Course Coordinator: Prof. Roberto Sassi

A. A. 2021-2022

Abstract

Cryptanalysis is an effective method for ensuring the security of cryptographic primitives by attacking them with the most advanced techniques.

This thesis provides a deep investigation of three different kinds of cryptanalysis for symmetric ciphers, differential, linear and algebraic, applying them to several symmetric ciphers, from the older ones to the most modern.

The algebraic approach consists in solving a polynomial system of equations representing a cryptographic primitive and involves a careful choice of the set of key variables to be fixed. The main instruments used to solve the system are Sat solvers and Gröbner basis of which a comparison is offered in some cases. Particular focus has been paid to SHA1 hash function and on the stream cipher E0, used in the Bluetooth protocol.

Samely, differential and linear cryptanalysis are applied to several symmetric ciphers. In particular, it is shown how to develop in Python an automatic tool for searching differential and linear trails with the constraint programming language Minizinc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the digital era [30], internet security is necessarily becoming one of the main concerns, especially because an exponential amount of private information is stored online. Caring about protecting personal data is the equivalent of worrying about not having a physical wallet stolen and cyber attackers are the new pickpockets.

A big slice of modern internet security relies on symmetric cryptography and it is likely that it will also be, at least in the near future. Indeed, even the threats posed by quantum computing, that compromise irremediably the security of the most common public-key ciphers, represent only a minor issue for symmetric cryptography: it is enough to double the key sizes to stay safe from the Grover quantistic algorithm [46]. In particular, cryptanalysis aims at analysing new and old information systems to find possible security flaws and promptly report them.

An additional concern is brought by the fact that we are often moving from the use of desktop computers to small devices, such as smart cards, industrial controllers, and others. This is the reason why in august 2018, NIST published a call for algorithms [87] to be considered for lightweight cryptographic standards.

The subject of this thesis is cryptanalysis on symmetric ciphers which will be explored in three of the most commonly used techniques.

The first part addresses algebraic cryptanalysis, a cryptanalytic method consisting in representing a cipher as a system of polynomial equations to be used for attacking the key. Algebraic cryptanalysis is commonly adopted and many examples can be found in the literature [4, 5, 20, 32, 48].

The first chapter provides a brief introduction to block and stream ciphers and to hash functions. In the second, one can find a detailed description of algebraic attacks, comprehending a tutorial on how to algebraically represent common operations, an overview of some techniques one can apply, and an introduction of two common methods used to solve the system: SAT solvers and Gröbner Bases. In the third and fourth chapters, the results obtained on the common hash function SHA-1 and the Bluetooth cipher E0 are shown extensively.

The first attack [9] tries to find a preimage of round-reduced versions of the compression function of SHA-1 with two different approaches: leaving more or less than 160 message bits free and fixing all the others. The resulting polynomial system is solved with SAT solvers.

In the short keystream-based attack on E0 [55], we provide a comparison of the solving performances by SAT solvers, Gröbner bases, and BDDs. We also explain how to fix variables from the key in a smart way which permits us to obtain very good results compared to the ones that can be found in the literature.

The second part of the thesis focuses on linear and differential cryptanalysis.

The latter, introduced in the first chapter, was first proposed by Biham and Shamir in 1990 [13] to attack, among others, the Data Encryption Standard (DES). It studies the propagation of an input difference throughout the cipher and exploits input/output difference couples holding with a high probability to mount chosen-plaintext partial key-recovery attacks. In the dissertation, the reader can find also a detailed analysis of how input differences propagate through the most common components of a cipher.

Linear cryptanalysis, firstly applied by Matsui to the FEAL cipher in 1992 [68], tries to find a linear relation involving only plaintext, key, and ciphertext bits that is or isn't satisfied particularly often. Similarly to the differential case, an explanation of how to deal with common components and an outline of the known plaintext partial key-recovery attack are provided.

In the last chapter of the thesis we present an automatic tool developed in Python which let a user obtain automatically differential or linear trails for any kind of cryptographic primitive. In an active cryptographic world that is continuously developing and changing itself, having automatic cryptanalysis procedures is of major importance.

1.1 Notation

Here we have a summary of the notations that will be used throughout the paper.

Indexes representing the bits of a word will be put in square brackets. Moreover, the indexes of the bits will always start from 0, with 0 representing the most significant bit.

In table 1.1 one can find some operations or symbols that are used throughout the dissertation.

Symbol	Operation
$\oplus$	Exclusive OR
$\wedge$	AND
$\vee$	OR
$\lll$	Left rotation
$\ggg$	Right rotation
$\ll$	Left shift
$\gg$	Right shift
$\boxplus$	Addition mod 2^{32}
$\boxminus$	Subtraction mod 2^{32}
$\mathbb{F}_2^n$	The binary field $GF(2)$

TABLE 1.1: Operations and symbols

Part I

Algebraic cryptanalysis

Chapter 1

Symmetric ciphers

In this chapter we will give a brief overview of the main symmetric cryptographic literature used in modern days: block ciphers, stream ciphers, and hash functions.

1.1 Block ciphers

A block cipher is a symmetric cipher in which the plaintext is considered as a number of concatenated blocks that are used, together with the key (and some other inputs, at times), to produce ciphertext blocks of equal length.

Block ciphers find applications in network-based cryptography such as windows drives, emails, and passwords. As a result, cryptanalysis on block ciphers has been widely explored and is still a very active field of research.

As already stated, a block cipher operates on n-bit blocks of plaintext to produce blocks of ciphertext of the same length. A "block" is a set of bits of fixed length.

Technically, we have a bijective function

$$E(P) : \mathbb{F}_2^n \to \mathbb{F}_2^n,$$

where $\mathbb{F}_2$ is the binary field, that is, a permutation. Note that this function has to be invertible so that a given ciphertext can be decrypted uniquely into the original message. It is trivial to prove that $2^n!$ such transformations are possible.

A block cipher also needs some data, shared by the users, to select which of these functions they are using to exchange messages. This is usually the key. An ideal block cipher is a big substitution box represented by a table containing the full transformation. In this case, the key would be the full table and one can choose between all the possible transformations with the desired length. In such a way, if the block size is big enough, one can avoid any possible brute force or cryptanalytic attack. However, using such a cipher is not practical.

Hence, the precise permutation is usually defined by the key, a word of k bits known only by the users. Namely, a block cipher is

$$E_k(P) : \mathbb{F}_2^n \times \mathbb{F}_2^k \to \mathbb{F}_2^n.$$

It is straightforward that the number of possible transformations a block cipher can represent is 2^k. k should be selected considering a security/efficiency trade-off. The Advanced Encryption Standard (AES), which is widely used nowadays, has three different versions with a block size of 128 bits and three different key lengths: 128, 192, and 256 bits.

A block cipher have some basic security requirements:

- It must be difficult to recover the key, even knowing a number of plaintext/ciphertext couples. A brute force attack simply tries all the possible keys on some plaintext/ciphertext couples until the right one is found. The attack is easily avoidable by taking a suitable key bit length;
- It must be difficult to recover the plaintext from a ciphertext without the knowledge of the key;
- Two different ciphertexts must be indistinguishable: having two different ciphertexts and plaintexts, it shouldn't be possible to associate them with probability higher than 50%.

1.1.1 Subtitution permutation networks

A common strategy to design block ciphers is to iterate a simple encryption function for a suitable number of rounds. Such a block cipher is called *iterated block cipher*. The round function should be applied every time with a different key: usually the subkeys used in every round are derived using a key schedule.

Substitution-Permutation Networks (SPNs) are iterated block ciphers with a round function that is constituted by two kind of components:

- **Substitution boxes** (S-Boxes): they are one-to-one correspondences between blocks of bits of fixed length. Common lengths are 4 or 8 for both input and output. For security purposes, an S-Box should not only be a permutation but should satisfy some properties. For instance, in a good S-Box changing a bit in the input should change at least half of the output. Moreover, every output bit ideally has to depend on every input bit. Finally, it should have good resistence to linear and differential cryptanalysis (more in section 1 and 2);
- **Permutation boxes** (P-Boxes): they are permutations of all the bits of the state. It is usually applied after the application of the S-Box.

The idea behind SPNs is essentially that the S-Boxes spread even a simple change in their inputs to many output bits guaranteeing the **diffusion** property by Shannon [81], and the P-Boxes spread the changes to many of the other S-Boxes guaranteeing the **confusion** property. In a few rounds, every output bit is linked to every input bit.

1.2 Stream ciphers

One of the main alternatives to block ciphers in symmetric cryptography is stream ciphers. Both kinds of ciphers have many real-world applications.

The main features of stream ciphers are:

- **Speed**: they are usually faster than other forms of encryption, including block ciphers;
- **Low complexity**: they are usually quite easy to study and implement;
- **Progressive encryption**: they encrypt a message one byte, or bit, at a time making it the best kind of ciphers for certain applications. This can be a big advantage even when decrypting: one can decide to decrypt only the first 5 bits of the message instead of decrypting the full ciphertext.

Stream ciphers are widely applied in the real world: web browsers, mobile communications, and Bluetooth.

To encrypt the message, the plaintext is simply XORed with the keystream, a pseudorandom stream of bits. The ideal case would be to have a completely random keystream whose bit length is equal to the one of the message, known to both users in advance. This can be achieved by exchanging this secret through a secure and independent channel but it's highly impractical to do so when the data traffic is very large. Hence, the keystream is generated with a key-dependent algorithm that should be cryptographically strong and efficient. In this way, the sender and the receiver have to share a much smaller key, used as a seed to an algorithmic procedure to generate a keystream of the desired length.

Formally, we could write again the stream cipher as a function:

$$\begin{aligned} E : \mathbb{F}_2^n \times \mathbb{F}_2^k &\to \mathbb{F}_2^n \\ (P,K) &\to E_k(P) = P \oplus K_s, \end{aligned}$$

where K_s is the keystream.

But actually, the strength of the cipher comes from the pseudorandom generator of the keystream that is a function:

$$\begin{aligned} E : \mathbb{F}_2^k &\to \mathbb{F}_2^* \\ K &\to K_s, \end{aligned}$$

where k is the bit length of the key.

Similarly to block ciphers, also the stream ciphers have to be secure against a key recovery attack. Moreover, the pseudorandom generator should be unpredictable, so that, for instance, an attacker cannot predict a certain keystream bit even when knowing all the previous ones.

1.3 Hash functions

A hash function is a function that accepts as input a block of data M of any size and produces a hash value $H(M)$ of fixed length, often called the "message digest". More formally:

$$\begin{aligned} H : \mathbb{F}_2^* &\to \mathbb{F}_2^n \\ M &\to H(M) \end{aligned}$$

is a hash function with a n-bits output.

Definition 1. *A hash function $H()$ satisfies the following points:*

- *it maps an arbitrary length message m to a fixed n-bit message digest MD;*
- *given a message m, it is easy to compute $MD = H(m)$;*
- *$H()$ does not require secret data to be computed;*

Additionally, for a hash function to be "good", a) images should be evenly distributed and apparently random and b) the output should change with high probability if a bit or some bits of the input are changed. Hash functions that are used in cryptography are referred to as cryptographic hash functions.

Definition 2. *A cryptographic hash function $H()$ satisfies the following points:*

- *(**Collision-resistance**) A hash function $H()$ is collision-resistant if it is difficult to find messages m_1 and m_2 such that $H(m_1) = H(m_2)$.*
- *(**Preimage-resistance**) Given a digest MD, it is difficult to find a message m such that $H(m) = MD$.*
- *(**Second preimage-resistance**) Given a message m_1, it is difficult to find m_2 such that $H(m_1) = H(m_2)$.*

It should be noted that collisions exist and that preimages of a given hash value are many. Actually, being the input of any length we have an infinite number of collisions and preimages to a given digest. However, finding them should be computationally impossible.

Hash functions have many applications. Here we have some of them:

- **Storing passwords**: users' passwords in a database should not be kept in plain text. To protect them, after being concatenated with a string called "salt", they are usually hashed. When a user wanting to access a website inserts his password, to check if it matches the stored one, it suffices to hash the given value and the right salt and check if the digest matches the one saved in the database;
- **Data integrity**: files can be accidentally damaged or modified when downloading from the internet or sending them on a channel. Attaching to them their hash let the receiver check if the file was in some way modified. Collision rarity is what guarantees this method to work properly;

- **Message autehentication**: it is quite similar to data integrity but this time we want to defend the data from a malicious third person who wants to modify it. Since in this case it is not enough to just hash the file because the adversary could do the same, a message authentication code (MAC), also known as keyed hash function, is used, namely a cryptographic hash function where the digest is dependent also from a key.
- **Proof-of-Work** in a blockchain: they are used as a way of ensuring computational work has been done to add a block to the chain.

1.3.1 Birthday attack

The brute-force complexity for finding a collision in a hash function with a n-bit hash value is $2^{n/2}$ thanks to the so-called "birthday attack". This name comes from the birthday paradox which states that taking at random 23 people the probability that at least two of them have the same birthday is about 50%. If we take 75 the probability becomes 99.9%. This might sound strange and counter-intuitive and thus is considered a "paradox".

The generic probabilistic problem is:

Problem 1. *Given k different objects, we draw m samples at random with replacement. What is the probability to take the same object twice?*

We can compute that the solution to 1 is

$$p(m,k) \approx 1 - e^{-\frac{m(m-1)}{2k}}. \tag{1.1}$$

Proof. We will work on the complementary probability, i.e. no object is drawn twice:

$$\overline{p(m,k)} = 1 \times \left(1 - \frac{1}{k}\right) \times \left(1 - \frac{2}{k}\right) \times \cdots \times \left(1 - \frac{m-1}{k}\right).$$

Using Taylor expansion we have the approximation $e^x \approx 1 + \frac{1}{x}$, getting

$$\overline{p(m,k)} \approx 1 \cdot e^{-\frac{1}{k}} \cdot e^{-\frac{2}{k}} \cdot \cdots \cdot e^{-\frac{m-1}{k}} = e^{-\frac{1+2+\cdots+m-1}{k}} = e^{-\frac{m(m-1)}{2k}}.$$

□

In order to find a collision for a hash function we may apply the following procedure:

1. Choose $2^{n/2}$ random messages: $M_1, \ldots, M_{n/2}$;
2. For $i = 1, \ldots, 2^{n/2}$ compute $t_i = H(M_i) \in \{0,1\}^n$;
3. Check if, for any $i \neq j$, $t_i = t_j$. If not so, go back to the first step;

In this specific case, we have that the objects are all the possible hash values, therefore $k = 2^n$.

In order to get probability $p(m,k)$ in 1.1 higher than 0.5, if k is not too small, we have to set $m \approx \sqrt{k}$. In our case, $m = 2^{n/2}$.

Although collisions exist, they are not easily findable by regular or malicious users, because the process of computing $2^{n/2}$ message digests is a very time-consuming operation. Usually, n is chosen in such a way that the probability to find a collision by brute force is negligible.

1.3.2 Brute-forcing a preimage

Similarly, we can brute-force against preimage and second preimage resistance. Given a digest D and, in the case of a second preimage, a first preimage M, the following algorithm can find a (another) message hashing to that digest:

1. Choose 2^n random messages: $M_1, \ldots, M_n$;
2. For $i = 1, \ldots, 2^n$ compute $t_i = H(M_i) \in \{0,1\}^n$;
3. Check if, for any i, we have $t_i = D$. If not so, go back to the first step;

The probability to get a specific digest D is $\frac{1}{2^n}$ and the number of messages expected to be analysed is 2^n.

1.3.3 Merkle-Damgärd construction

The most common approaches to build cryptographic hash functions are the Merkle-Damgård construction [69], sponge construction [86] and Haifa construction [12]. In addition, researchers adopt block ciphers in different ways to design a hash function [6].

We will explain in detail only the first one (for more, see [73]) since it is one of the most commonly used (see e.g. MD5, SHA-1, SHA-2) and one of the results of my research was an algebraic attack on SHA-1 which relies on it. *Merkle-Damgård construction* is based on a one-way compression function (i.e. a function, difficult to invert, whose input bit length is bigger than the output bit length) and a padding scheme (see figure 1.1).

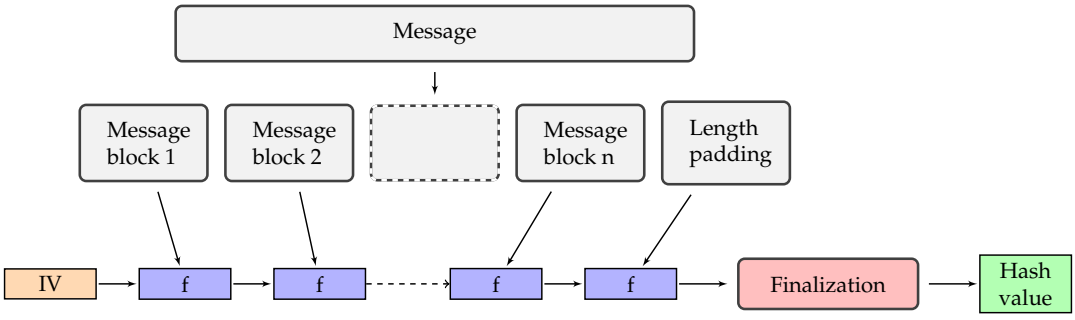

FIGURE 1.1: The Merkle-Damgärd construction

The padding scheme is needed to have a fixed-length binary string that is then split into blocks of fixed size which are processed by the compression one at a time. However, they are not hashed separately since the output of a block is combined with the following block in the compression function.

Chapter 2

Algebraic attacks

2.1 The algebraic representation

The algebraic representation of a primitive is its representation as a system of boolean equations. More specifically, every bit of the inputs and outputs of the function (e.g. plaintext, key, ciphertext, hash digest) is represented by a variable, and these variables are related by equations that describe exactly how the ciphertext (or hash digest) bits are obtained by combining the bits of the inputs. Note that the variables can take values in the binary field $\mathbb{F}_2$ and that the $*$ and $+$ operation correspond to the Boolean operations AND and XOR.

Definition 3. *Given a primitive with inputs $I_0, \ldots, I_n$ with bit length $m_0, \ldots, m_n$, output C with length s and function E, that is $E(I_0, \ldots, I_n) = C$, its algebraic representation is given by:*

$$\begin{cases} C[0] & = P_0(I_0[0], \ldots, I_0[m_0 - 1], I_1[0], \ldots, I_n[0], \ldots, I_n[m_n - 1]) \\ C[1] & = P_1(I_0[0], \ldots, I_0[m_0 - 1], I_1[0], \ldots, I_n[0], \ldots, I_n[m_n - 1]) \\ \vdots & \\ C[s-1] & = P_{s-1}(I_1[0], \ldots, I_1[m_0 - 1], I_1[0], \ldots, I_n[0], \ldots, I_n[m_n - 1]) \end{cases}$$

where $P_0, \ldots, P_{s-1}$ are some polynomials in $m_0 + m_1 + \cdots + m_n$ variables.

If looking at the kind of system as a set of logical formulas (the values 0 and 1 can be seen as truth values) we can say it is represented in its algebraic normal form (ANF), a way of writing logical formulas we will discuss in more detail in section 2.3.1.

We remark that, when dealing with stream ciphers, we don't consider the plaintext and the ciphertext in the algebraic representation but we just represent the underlying pseudorandom generator of the keystream. Hence, the keystream will be the output C and the key, or seed, will be the only input I to the function.

Example 2.1.1. *Let's consider the toy block cipher in figure 2.1 with a 32-bit plaintext P and a 16-bit key K. Consider P_1, P_2 are respectively the left and right half of the plaintext and C_1, C_2 the left and the right half of the ciphertext.*

Then the algebraic representation of the cipher is the following:

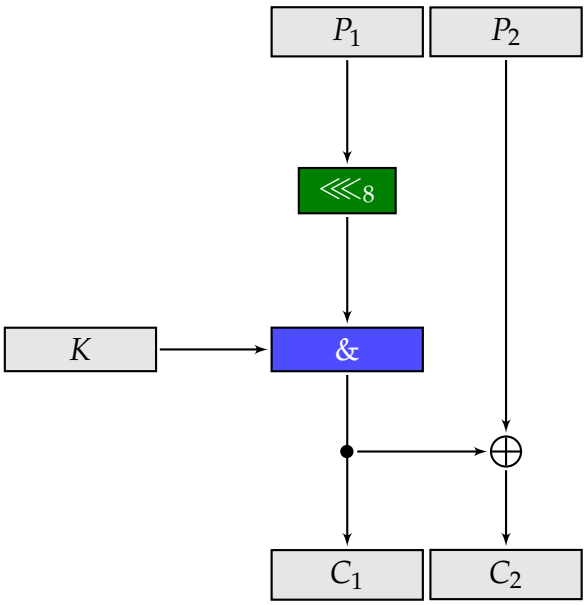

FIGURE 2.1: Toy block cipher

$$\begin{cases} C[0] &= P[8] * K[0] \\ C[1] &= P[9] * K[1] \\ \vdots & \\ C[15] &= P[7] * K[15] \\ C[16] &= P[8] * K[0] + P[16] \\ \vdots & \\ C[31] &= P[7] * K[15] + P[31] \end{cases}.$$

2.1.1 Auxiliary variables

As the example shows, this representation results in a system of s equations where s is the bit length of the ciphertext and the variables involved are only the ones from the inputs and outputs of the cipher. Nonetheless, we usually make use of some auxiliary variables representing that help us solve the system more efficiently. This can be useful or necessary in some cases:

- Firstly, we have to consider a common practice when designing a primitive (e.g. a block cipher) is to use some kind of "round function" that is applied several times so that the polynomials $P_1, \ldots, P_s$ may be very long. Adding some auxiliary variables entails a higher number of equations and variables. However, the equations can become much smaller. One might ask if this always improves the efficiency of the solver or when is appropriate to group some variables and operations in a new auxiliary variable and we usually get the answer from testing activities on the systems of equations and their resolutions.

- Nonlinear operations are usually the most difficult part to deal with when trying to solve a polynomial system of equations such as the ones above. Sometimes it can be useful to linearize the system by adding auxiliary variables whenever a multiplication between two variables comes up. This would result in a system where the linear equations and the nonlinear ones are clearly split and eventually produce an improvement in the solving times.

2.1.2 Main operations and their algebraic representation

In this subsection, we give a detailed description of how the main components appearing in modern primitives are represented in the system of equations. We will use I (I_i) for the input (inputs) of the operations and O for the output. If not specified, n is the bit length of the words.

- **AND** and **XOR**: As already stated, they simply correspond to the $+$ and $*$ of the binary field;
- **OR**: an or operation between two inputs is represented as:

$$O_i = I_1[i] \oplus I_2[i] \oplus I_1[i] * I_2[i] \text{ for } i = 0, \ldots, n-1;$$

- **NOT**: Not operation simply flips every bit of the word it is applied to

$$O[i] = I[i] \oplus 1 \text{ for } i = 0, \ldots, n-1;$$

- **LINEAR LAYER**: A linear layer is a multiplication of the input with a bit matrix. It usually represents the P-Box or part of the P-Box in a SPN cipher, introduced in subsection 1.1.1. Let us suppose to have a 4×4 matrix:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix},$$

then the system of equations results in:

$$\begin{cases} O[0] = I[0] \oplus I[1] \oplus I[3] \\ O[1] = I[2] \\ O[2] = I[1] \oplus I[3] \\ O[3] = I[1] \end{cases};$$

- **MIX COLUMN**: The mix column is again a multiplication with a matrix but this time we multiply bytes and then reduce modulo a certain integer. The matrix can be still transformed into its bit version reducing a mix column to a linear layer;

- **MODADD**: For modular addition one usually adopts some auxiliary variables to keep track of the carries. Here are the equations for an addition modulo 2^{32}:

$$\begin{cases} C[31] = 0 & \\ O[i] = I_1[i] \oplus I_2[i] \oplus C[i] & \text{for } i = 0,\dots,31 \\ C[i-1] = I_1[i] * I_2[i] \oplus I_1[i] * C[i] \oplus I_2[i] * C[i] & \text{for } i = 1,\dots,31; \end{cases}$$

- **ROTATION**: Rotation is usually represented together with some other operation since it is a renaming of variables. Here we have a left rotation of amount r of a word of n bits:

$$O[i] = I[i+r] \text{ for } i = 0,\dots,n-1;$$

- **SHIFT**: The same observation is valid for the shift. Here we have a left shift of amount s of a word of n bits:

$$\begin{cases} O[i] = I[i+s] & \text{for } i = 0,\dots,n-s-1 \\ O[i] = 0 & \text{for } i = n-s,\dots,n-1. \end{cases}$$

- **S-BOX**: if we have the structure of the S-box, that is the operations that compose it, then we can simply use the rules above to algebraically represent it. Istead, if we have only its lookup table the procedure is a bit longer. An n-bits input to m-bits output substitution box can be seen as m different n-bits input to one-bit output Boolean functions. To obtain the algebraic representation of a Boolean function one can follow these steps [19]:

  1. Write the truth table of the function f, i.e. a string of bits where the i-th bit from the left is the image of the integer i;
  2. Write the support of the function f, namely the set containing the inputs with nonzeros images. If the truth table of the function is (01000101) the support in binary is $supp(f) = \{001, 101, 111\}$;
  3. Having the support, we convert the function in the following form:

$$f(x) = \bigoplus_{c \in supp(f)} x_0^{(c_0)} x_1^{(c_1)} \cdots x_n^{(c_n)},$$

  where $x = x_0 x_1 \dots x_n$, $c = c_0 c_1 \dots c_n$ and $x_i^{(c_i)} = x_i$ if $c_i = 1$ and $x_i^{(c_i)} = 1 \oplus x_i$ if $c_i = 0$. In the previous case where $supp(f) = \{001, 101, 111\}$ the formula would be

$$f(x) = x_0^{(0)} x_1^{(0)} x_2^{(1)} \oplus x_0^{(1)} x_1^{(0)} x_2^{(1)} \oplus x_0^{(1)} x_1^{(1)} x_2^{(1)},$$

that is:

$$f(x) = x_2 \oplus x_0 * x_2 \oplus x_1 * x_2 \oplus x_0 * x_2 \oplus x_0 * x_1 * x_2.$$

Suppose to have the following toy S-Box:

0	1	2	3
0	3	1	2

, the truth tables would be (0101) and (0110) and the supports are $supp(f_1) = \{01, 11\}, supp(f_2) = \{01, 10\}$. Then the system of equations representing the S-Box is:

$$\begin{cases} O[0] = I[1] \\ O[1] = I[0] + I[1]. \end{cases}'$$

2.2 Attacks on cryptographic primitives

After obtaining the algebraic representation of the primitive, if dealing with a block or a stream cipher, one usually tries to mount a key-recovery, i.e. tries to retrieve the bits of the key. In both cases, we attempt a known-plaintext attack, meaning we suppose to have a couple (or some) (P, C) where P is a possible plaintext and C the respective ciphertext when encrypting with the key K we are trying to recover. In stream ciphers, with such a couple one can easily recover the keystream. By fixing the variables representing the bits of the keystream one can try to retrieve the remaining variables. In block ciphers, we can fix the variables representing the bits of the plaintext and the ciphertext.

In both cases, the non-fixed variables are the auxiliary ones or the ones representing the bits of the key. Solving the resulting system would then mean having a successful full key-recovery attack. Hence, an algebraic attack to a stream or block cipher or to a hash function consists in solving a system of polynomial equations over a finite field $\mathbb{K} = GF(q)$ with usually $q = 2$.

When attacking a hash function the situation is different since one doesn't usually have any key unless he is analysing some HMAC (keyed-hash message authentication code). Assuming the latter is not the case, the algebraic representation of a hash function can be exploited to mount a preimage attack by fixing the hash value and leaving the bits of the message unknown.

However, it is unlikely that this technique directly succeeds in retrieving a full key or a preimage so some tricks are used to make the attack practical or to give a theoretical attack whose expected complexity is better than a brute force one.

2.2.1 Fixing some key bits

One of the best ways of improving the solving and managing to partly retrieve the key is to fix some of the key bits randomly. Let k be the key bit length, then the idea is to fix n out of the k variables representing the key to a

certain random value. Be aware that only one (or just a few) combination of values is the correct one so one should try all the 2^n possible values to find the rest of the key. In all the other cases we expect the system to be unsatisfiable with any combination of values of the unfixed variables.

This is the idea underlying the **"Guess and Determine"** technique. This is the simple algorithm to follow:

1. Choose n variables to fix between the k variables of the key;
2. Choose a vector in $\mathbb{F}_2^n$ which wasn't previously chosen and fix the n variables to those values;
3. Solve the system of equations;
4. If the system is unsatisfiable, go back to step 2).
5. If the system is solved, test the key on another plaintext-ciphertext pair, or, in the case of stream ciphers, check it with the following bits of the keystream. If it isn't correct go back to step 2), otherwise, the rest of the key is retrieved.

In the last point, double-checking the key is needed to avoid taking the wrong key.

When working on block ciphers, it may be that a particular plaintext/-ciphertext couple is obtained with more than one key. However, if another couple is tested and the encryption with the key found matches the given ciphertext, then it is really unlikely the key is not the correct one. To be precise, considering a couple with a ciphertext of m bits, the probability that a key encrypts the plaintext to that specific ciphertext is $\frac{1}{2^m}$. Thus, the probability that this happens with two different pairs is $\frac{1}{2^{2m}}$, which makes it really unlikely to happen, considering we are trying at most 2^k different keys (that is the brute-force upper bound) with k being the key bit length and usually $k \leq m$.

But this is something to pay attention to especially when dealing with stream ciphers. The number of different possible solutions is strictly linked with the number of bits of keystream represented in the system of equations. It is clear that using more bits decreases the number of possible keys but also means more equations and more variables usually resulting in worst solving times. It is important to find a good tradeoff between the number of solutions and the performances when solving the system of equations. Multiple solutions may be accepted since the key can be checked with a few more keystream bits, but it's still important to keep this number small to avoid the double-check phase for many possible keys.

Considering the time for checking if a solution is spurious or not to be negligible, the expected runtime of a guess and determine approach is

$$2^n * \tau,$$

where τ is the average running time for solving a single (usually impossible) polynomial system.

Testing the algebraic strength of the primitive by fixing some bits of the key can be also justified by supposing another kind of attack, for instance a side-channel attack, provided us with that information.

Even if the two ideas of attack seem similar, there's a big difference when testing the resolution of the system. In the guess and determine one fixes randomly the values of the key bits and tests the solving time, which is in most cases the time the solver takes to find out the system is unsatisfiable. On the contrary, when using the second idea one has to fix the key values to the right ones and tests the solving time for finding the rest of the key. This can be crucial in choosing which kind of solver is the best for the specific kind of attack.

2.2.2 Choosing the set of variables to be fixed

When applying the above strategies one needs to choose a set of variables representing the key that needs to be fixed. One can simply take this set randomly but it would be much more efficient to choose it carefully in order to decrease the solving time of the system of equations. Testing often proves the resolution time can be drastically diminished by fixing the right key variables both if the system has a solution or if it has not.

In my research, the choice of the variables was always a core node when applying algebraic attacks to cryptographic primitives and in some cases, it highly improved performances outperforming the state of the art.

A simple starting point when working with any cryptographic primitive is to check which are the variables that occur more frequently in the system and give priority to them. However, during my research, I didn't come up with any general strategy which can be applied to any cipher or hash function for choosing the best variables to fix. Even if I did some investigation on the argument, it wasn't one of the main focuses and nothing about that will be included in the dissertation.

The choice will usually depend on the specific algorithm and is made after a careful analysis of its structure so that this argument will be discussed in more detail in other chapters.

2.2.3 Consider simplified versions of a primitive

Another way to make the resolution practical is to analyse simplified versions of the primitive.

In the case of block ciphers and hash functions, which are usually made up of a round function repeating itself a certain number of times this is done considering a version with a reduced number of rounds. Although a situation of this kind is not common in real life, this attack can be really useful to see how far it is possible to go with a practical attack. Moreover, this gives also an idea of the strength of the original cipher.

This idea is sometimes applied also to stream ciphers. For instance, this is the case of Trivium [27], a simple stream cipher designed by C. De Cannière and B.Preneel in 2005, with good performances in both hardware and

software applications. One year later, in [75], the author introduces Bivium, a reduced version of Trivium which was widely cryptanalysed in the following years. Bivium and Trivium will not be discussed in this dissertation even if I worked on the cryptanalysis of the first and I am currently working on the original cipher.

2.3 Possible solving methods

In this section, we present the two solving methods I mainly adopted in my research: SAT solvers and Gröbner basis-based solvers.

2.3.1 SAT solvers

A SAT solver is an algorithm that aims at solving Boolean satisfaction problems. To give a proper definition of what that is, we go through some of the bases of Boolean algebra.

Definition 4. *A **variable** is an unknown truth value, i.e. can take values in* $\{0,1\}$.

The variables can be input to three different operations (or connective): **AND**, **OR** and **NOT**, represented by $\wedge$, $\vee$ and $\neg$ respectively. When referring to these operations as connectives they can be also called conjunction, disjunction, and negation respectively.

Now, we can define what a formula is [79].

Definition 5. *Every variable is a **formula**. If F_1 and F_2 are formulas then $F_1 \wedge F_2$, $F_1 \vee F_2$ and $\neg F_1$ are formulas.*

Moreover, the rules in 5 constitute a minimal set in order to build every logic formula.

Definition 6. *An **assignment** is a function mapping each variable to* 0 *or* 1.

Definition 7. *A **Boolean Satisfaction problem**, given a formula or set of formulas, aims at finding an assignment such that the formula evaluates to 1, in which case we say the problem is satisfiable. On the contrary, if such an assignment doesn't exist, the problem is unsatisfiable.*

Algebraic normal form (ANF) and conjunctive normal form (CNF) are two different ways to encode a Boolean Satisfaction problem.

Definition 8 (Algebraic normal form)**.** *A logical formula is in Algebraic normal form if it is made up of some variables and truth values connected by the AND or XOR operations, namely a polynomial over* $\mathbb{F}_2$.

Definition 9 (CNF formula, clauses, literals, pure literal)**.** *A Boolean formula F is in Conjunctive Normal Form (CNF) if it is a conjunction of disjunctions, namely it is represented in the following form:*

$$F = C_1 \wedge C_2 \wedge \ldots C_m. \qquad \textit{(conjunction)},$$

where the C_is are again formulas called clauses and must have the following form

$$C_i = (l_{i,1} \vee l_{i,2} \vee \ldots \vee l_{i,k}) \qquad \textit{(disjunction)},$$

where the $l_{i,j}$ are literals, that is variables or variables negation:

$$l_{i,j} \in \{x_1, x_2, \ldots, x_n\} \cup \{\neg x_1, \neg x_2, \ldots, \neg x_n\}.$$

The polarity of a literal is an indicator of its positivity or negativity. A literal in a formula is pure if its negation never occurs in it or if its polarity is always positive.

Example 2.3.1. *Let us suppose we have the following formula in algebraic normal form:*

$$1 \oplus a \oplus ab = 0.$$

In its conjunctive normal form, it simply becomes:

$$a \wedge \neg b.$$

The formula is satisfiable and an assignment satisfying it is $a = 1$ and $b = 0$.

At the state of the art, the most commonly used solving procedures adopted by SAT solvers are the DPLL algorithm and the CDCL algorithm.

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm was introduced more than 50 years ago [25] but it's still the basis of most modern sat solvers. However, it was not until the 1990s that the first DPLL-based sat solvers started to appear adding some enhancements to the original procedure: clause learning, non-chronological backtracking, branching heuristics, restart strategies, and lazy data structures.

The basic idea is a simple backtracking algorithm:

1. Select a non-fixed variable and assign it a truth value. This is called a *decision*;
2. Simplify the formula, i.e. every clause in which the variable appears is canceled if it is set to 1 otherwise only the variable is removed (because some other literal in the clause has to be true for the clause to be satisfied);
3. Check if the simplified formula is satisfiable by applying the algorithm recursively;
4. If it is not, we go back to 1 and assign the chosen variable the other value;
5. If the resulting formula is still unsatisfiable we have to backtrack to some assignment made in a previous recursive step. If there is nowhere to backtrack the formula is unsatisfiable.

This simple method is enhanced with two techniques:

- **Unit propagation**: if a clause contains only one unassigned literal then that literal must be 1 for the formula to be true. Whenever a formula contains some unit clause, the literal is assigned to 1 and the formula is simplified avoiding exploring a large part of the naive search space.
- **Pure literal elimination**: a pure literal can be assigned to true in the formula without affecting its satisfiability. Essentially, before selecting a variable to make a decision all pure literals are removed from the formula by eliminating every clause containing them;

These two methods become part of the simplification step.

CDCL (conflict-driven clause learning) algorithm [65, 66] is the most used in modern sat solvers and is still based on the backtracking idea of the DPLL. The main difference between CDCL and DPLL is that CDCL's back jumping is non-chronological: when it finds a conflict the DPLL algorithm backtracks to the last decision made that still has a value to assign available. CDCL improves DPLL by introducing clause learning, which is what makes non-chronological backtracking possible. After a conflict is reached, CDCL looks at the guesses it made and the relative implications and builds an implication graph. Then, it can generate a clause, which is added to the formula to satisfy, that is the negation of the assignment that led to the conflict. Then the algorithm backtracks to the appropriate decision level (and not on the previous one as DPLL), that is where the first of the variables involved in the conflict is assigned.

The efficiency of state-of-the-art SAT solvers relies heavily on various features that have been developed, analysed, and tested over the last decade. One of them is randomized restart strategies, a procedure that stops the search and restarts it from decision level 0 keeping all clauses learned so far. This has been shown to greatly help in reducing the solution time. Most of the current SAT solvers employ aggressive restart policies such as geom [91], Glue [42] and Luby [63].

A parameter one can vary when solving a Boolean Satisfaction problem with SAT solvers is the polarity. This parameter establishes what value is assigned to a variable when making a decision.

- **Random**: polarity for the variable is chosen randomly;
- **True**: the variables are always assigned to true;
- **False**: the variables are always assigned to false;
- **Auto**: the polarity is equal to the last one used (also called 'caching').

2.3.2 Gröbner basis based solvers

Gröbner bases are a mathematical object introduced together with an algorithm to compute them in 1965 by B.Buchberger in his doctoral thesis [16]. Exploiting this object, he manages to give solutions to a large number of algorithmic problems. The applications of this object cover many different

fields such as algebraic geometry, optimization, control theory, robotics, and many others.

A Gröbner basis is a specific generating set of an ideal over a polynomial ring. More precisely, we are in the polynomial ring $R[x_1,\dots,x_n]$ with R any principal ideal ring with 1.

From now on, $R = \mathbb{F}_2$ and $P = \mathbb{F}_2[x_1,\dots,x_n]$.

Definition 10. *Given a monomial ordering on the set of all monomials in P, a finite subset G of an ideal $0 \neq I \subset P$ is a Gröbner bases of I if for any polynomial $f \in I$, there exists a polynomial $g \in G$ such that the largest monomial (w.r.t the chosen monomial ordering) of g divides the largest monomial of f.*

Among the others, an important use of the Gröbner basis is for solving systems of equations. Let us consider a system

$$f_1(x_1,\dots,x_n) = \cdots = f_m(x_1,\dots,x_n) = 0. \tag{2.1}$$

We can associate to that the ideal $I = \langle f_1,\dots,f_m\rangle \subset P$.

If we choose a monomial ordering that allows successive eliminations of variables, a Gröbner basis of the ideal I describes the set of solutions of the polynomial system, including the ones that lie in the algebraic closure of $\mathbb{F}_2$. If one wants to find only the solutions that are in $\mathbb{F}_2$ the so called "field equations" have to be added to the system of equations:

$$L = \langle x_1^2 - x_1,\dots,x_n^2 - x_n\rangle.$$

This is due to the following well-known result you can find for instance in [40].

Proposition 2.3.1. *Let $\overline{\mathbb{K}}$ be the algebraic closure of $\mathbb{K}$ and*

$$V(I) = \{(\alpha_1,\dots,\alpha_n) \in \overline{\mathbb{K}}^n \,|\, f_i(\alpha_1,\dots,\alpha_n) = 0 \text{ for } 1 \leq i \leq m\}.$$

Put $V_{\mathbb{K}}(I) = V(I) \cap \mathbb{K}^n$, that is $V_{\mathbb{K}}(I)$ is the set of the $\mathbb{K}$-solutions of the polynomial system 2.1. We have that $V(L) = \mathbb{K}^n$ and $V_{\mathbb{K}}(I) = V(I+L)$ where $I + L \subset P$ is a radical ideal.

Gröbner bases can be used for the resolution of polynomial systems thanks to the following result that can be derived from the Nullstellensatz Theorem for finite fields (see [40]).

Proposition 2.3.2. *Assume that the polynomial system 2.1 has a single or no $\mathbb{K}$-solution. Then, the (reduced) universal Gröbner bases G of the ideal $I + L$, that is, its Gröbner bases with respect to any monomial ordering of P is*

$$\begin{cases} \{x_1 - \alpha_1,\dots,x_r - \alpha_r\} & \text{if } \; V_{\mathbb{K}}(I) = \{(\alpha_1,\dots,\alpha_r)\}, \\ \{1\} & \text{otherwise.} \end{cases}$$

In other words, by obtaining the reduced universal Gröbner bases for the ideal containing the polynomials of the system and the field equations one

directly has the solution of the system if that exists. If not, one obtains the unit polynomial.

The current state-of-the-art algorithm to compute Gröbner bases is the F_4 algorithm from Faugère [34] and a more recent variant, named M4GB, was published in 2017 by Makarim and Stevens [64]. Despite many improvements, the theoretical complexity of existing Groebner bases algorithms is exponential in the number of variables.

For more theoretical results regarding Gröbner bases more can be found, for instance, in [62].

2.4 Guess and determine with Gröbner bases

Let's see the details on how a guess and determine strategy can be applied using Gröbner bases. This particular approach was introduced in section 2.2.1.

It is clear that, by brute-forcing, one can compute $V_{\mathbb{K}}(I)$ in

$$\lambda = 2^n \cdot c$$

by assuming it takes a constant time c to evaluate the polynomial f_i over a vector $(\beta_1, \ldots, \beta_n) \in \mathbb{K}^n$. To improve this upper bound and obtain an attack whose complexity is lower than the brute force one can solve the system for instance with SAT solvers [31, 20] or Gröbner bases [45].

In general, Gröbner bases performances for solving polynomial systems are not good since the complexity grows exponentially with the number of variables n. One can expect to obtain results even worst than the brute force upper bound. Despite this, some features of Gröbner bases that can result in good performances under certain conditions have to be underlined :

- If the number of variables n is not too big and the set of $\mathbb{K}$-solution of the system 2.1 is small, they become an effective tool for solving it;
- By proposition 2.1, we have that every monomial ordering can be chosen to solve the system so that one can use the most efficient ones;
- When the system is impossible its Gröbner basis is $G = \{1\}$ and the Buchberger algorithm stops as soon as a constant in $\mathbb{K}$ is obtained. We remark that this feature blends well, for instance, with the guess and determine strategy, which, as already explained, mostly consists of solving systems of equations without a solution.

From now on, we assume that the polynomial system 2.1 has a single $\mathbb{K}$-solution, i.e. $V_{\mathbb{K}}(I) = \{\alpha_1, \ldots, \alpha_n\}$. In the guess and determine technique explained in section 2.2.1 one solves many systems of equations obtained by evaluating some subset of variables, say $\{x_1, \ldots, x_s\}$ with $0 \leq s \leq n$, in all possible ways over the finite field $\mathbb{K}$. To work with Gröbner bases, one would define the ideal

$$E_{\beta_1,\ldots,\beta_s} = \langle x_1 - \beta_1, \ldots, x_s - \beta_s \rangle \subset P$$

for all vectors $(\beta_1,\ldots,\beta_s) \in \mathbb{K}^s$ and consider

$$I_{\beta_1,\ldots,\beta_s} = I + L + E_{\beta_1,\ldots,\beta_s}.$$

It is trivial that this is equivalent to the assignment $x_1 = \beta_1,\ldots,x_s = \beta_s$.

The generating set of $I_{\beta_1,\ldots,\beta_s}$ is

$$H_{\beta_1,\ldots,\beta_s} = \{f_1(\beta_1,\ldots,\beta_s,x_{s+1},\ldots,x_n),\ldots,f_n(\beta_1,\ldots,\beta_s,x_{s+1},\ldots,x_n), \\ x_{s+1}^2 - x_{s+1},\ldots,x_n^2 - x_n, x_1 - \beta_1,\ldots,x_s - \beta_s\} \subset P$$

One clearly has that:

$$V(I_{\beta_1,\ldots,\beta_s}) = \begin{cases} \{(\alpha_1,\ldots,\alpha_n)\} & \text{if } (\beta_1,\ldots,\beta_s) = (\alpha_1,\ldots,\alpha_s), \\ \emptyset & \text{otherwise.} \end{cases}$$

The sequential running time of the guess and determine is

$$\mu_s = \sum_{(\beta_1,\ldots,\beta_s)\in\mathbb{K}^s} \tau_{\beta_1,\ldots,\beta_s}$$

where $\tau_{\beta_1,\ldots,\beta_s}$ is the computation time of the Gröbner bases of $I_{\beta_1,\ldots,\beta_s}$ starting with the generating set $H_{\beta_1,\ldots,\beta_s}$. Considering the average runtime of this computation τ_s the time complexity becomes

$$\mu_s = 2^s\tau_s.$$

Having assumed the solution to be unique, we have that for all $2^s - 1$ vectors $(\beta_1,\ldots,\beta_s) \neq (\alpha_1\ldots,\alpha_s)$ the Gröbner bases is always $G = \{1\}$. As already stated, this represents a big advantage for Gröbner bases compared to, for instance, Sat solvers. Indeed, whereas a SAT solver has to explore the full space to ensure the system is unsatisfiable, a symbolic method just has to find some inconsistency of the type $1 = 0$.

These theoretical facts were confirmed in chapter 3 when we describe the results obtained in the testing activities of an algebraic attack to the stream cipher E0 where Gröbner bases resulted superior to Sat solvers and BDDs.

Chapter 3

Algebraic attack to E0

In this chapter, we will describe an algebraic attack to the stream cipher E0, used in Bluetooth, a well-known communication protocol present in almost all mobiles, personal computers, remote controllers, and many other devices. The results of this research are published in [55].

After becoming the Bluetooth protocol, E0 was subject to several cryptanalytic attacks which can be divided into two main classes: long or short keystream attacks. The best attacks come from the first category [5, 35, 41] but the scenario is not real: the protocol doesn't let using the same key to generate a keystream longer than 2745 bits.

Cryptanalysis of E0 can also be distinguished in correlation attacks or algebraic cryptanalysis. In the latter, it is common to use Binary Decision Diagrams (BDDs) as it is done in [52, 53, 54, 80], whereas other solvers as Gröbner basis or XL-algorithm are more rarely applied [4, 21].

In our paper [55] we show that Gröbner bases can be a powerful tool when dealing with systems with a few numbers of solutions. This is the case of E0, even with a small number of keystream bits, as we show in our analysis. This fact alone can already be a matter of concern.

We carried out a comparison of the performances of Gröbner bases, SAT solvers, and BDDs using a large test set concluding the firsts are the best possible tool for this problem. Moreover, our results with BDDs confirm and improve the ones in [54] where one can find another practical algebraic attack to E0 based on the same kind of solvers.

Our attack is a known-plaintext attack based on a 60 bits keystream. This kind of algebraic attack for stream cipher was already discussed in chapter 2. We remark that in the Bluetooth protocol the initial state of E0 is reinitialized using the last 128 keystream bits. Since we use only 60 our attack can work in the real case by simply applying it twice.

3.1 The stream cipher E0

As usual for stream ciphers, the core is the keystream generation in figure 3.1.

The 132-bit state is divided into 4 linear feedback shift registers (LFSR) of length respectively 25,31,33 and 39 bits and a finite state machine (FSM) of 4 bits which are stored in a pair of 2-bit elements. We will denote the four LFSRs in the state x, y, z, u, respectively. In this chapter, to denote the bits we

will use round brackets (they can also be intended as images of a function in the clock t): the state at any clock t will be

$$x(t)\dots x(t+24),\ y(t)\dots y(t+30),\ z(t)\dots z(t+32),\ u(t)\dots u(t+38),$$

so, at clock 0,

$$x(0)\dots x(24),\ y(0)\dots y(30),\ z(0)\dots z(32),\ u(0)\dots u(38),$$

where 0 is used for the most significative bit. Similarly, the state machine is denoted by $c(t)c(t+1)d(t)d(t+1)$. The four LFSRs are respectively defined by the following primitive polynomials with coefficients in $\mathbb{F}_2$

$$\begin{aligned} p_1 &= x^{25} \oplus x^{17} \oplus x^{13} \oplus x^5 \oplus 1, \\ p_2 &= x^{31} \oplus x^{19} \oplus x^{15} \oplus x^7 \oplus 1, \\ p_3 &= x^{33} \oplus x^{29} \oplus x^9 \oplus x^5 \oplus 1, \\ p_4 &= x^{39} \oplus x^{35} \oplus x^{11} \oplus x^3 \oplus 1. \end{aligned}$$

Hence, at clock $t+1$ the state is

$$\mathrm{x(t+1)}\dots\mathrm{x(t+25)},\ \mathrm{y(t+1)}\dots\mathrm{y(t+31)},\ \mathrm{z(t+1)}\dots\mathrm{z(t+33)},\ \mathrm{u(t+1)}\dots\mathrm{u(t+39)},$$

where

$$\begin{cases} x(t+25) = x(t+20) \oplus x(t+12) \oplus x(t+8) \oplus x(t) \\ y(t+31) = y(t+24) \oplus y(t+16) \oplus y(t+12) \oplus y(t) \\ z(t+33) = z(t+28) \oplus z(t+24) \oplus z(t+4) \oplus z(t) \\ u(t+39) = u(t+36) \oplus u(t+28) \oplus u(t+4) \oplus u(t). \end{cases}$$

For what concerns the FSM, it is important to define two integer numbers:

$$\begin{cases} C(t) = d(t)2 + c(t) \\ C(t+1) = d(t+1)2 + c(t+1). \end{cases}$$

At clock $t+1$ the FSM is $c(t+1)c(t+2)d(t+1)d(t+2)$ where:

$$(d(t+2),c(t+2)) = (g_1(t+1),g_0(t+1)) + (d(t+1),c(t+1)) + T(d(t),c(t)) \tag{3.1}$$

where $T : \mathbb{F}_2^2 \to \mathbb{F}_2^2$ is the linear bijection

$$(d(t),c(t)) \mapsto (c(t),d(t)+c(t))$$

and the 2-bit vector $(g_1(t+1),g_0(t+1)) \in \mathbb{F}_2^2$ is defined as follows. Consider the sum

$$F(t) = x(t) + y(t+6) + z(t) + u(t+6) \in \mathbb{Z}$$

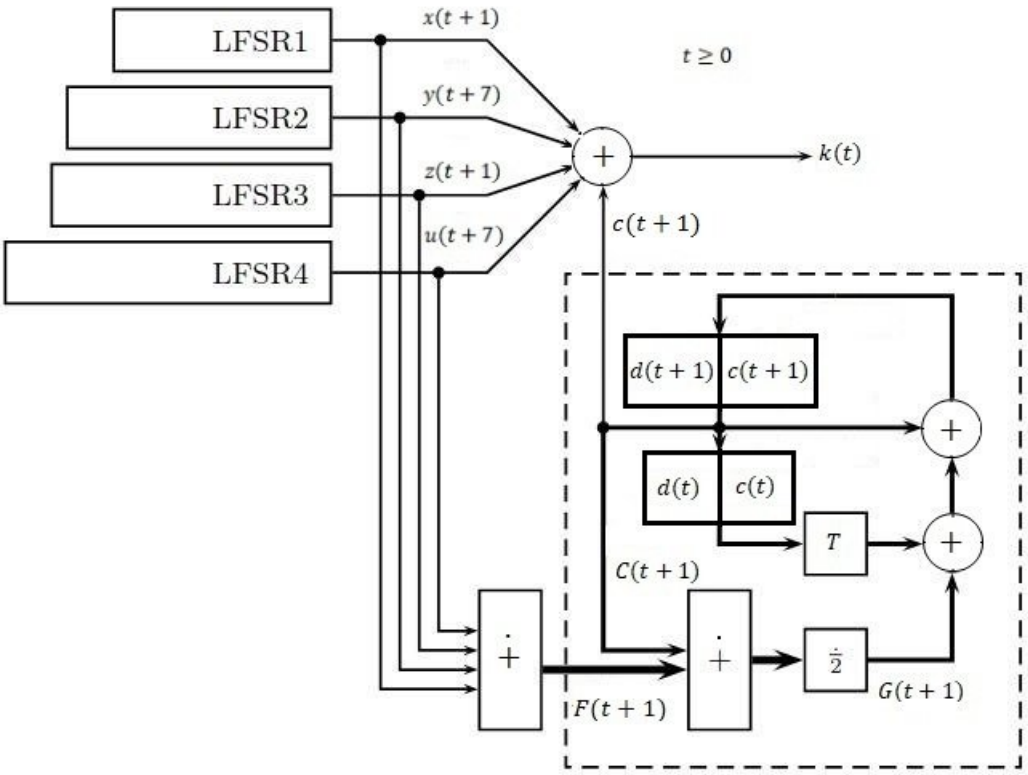

FIGURE 3.1: E0 keystream generation

and define the integer

$$G(t+1) = \left\lfloor \frac{F(t+1) + C(t+1)}{2} \right\rfloor.$$

Since $0 \leq G(t+1) \leq 3$, we define the 2-bit element $(g_1(t+1), g_0(t+1)) \in \mathbb{F}_2^2$ as the binary representation of $G(t+1)$, namely

$$G(t+1) = g_1(t+1)2 + g_0(t+1).$$

To show the explicit relation between the new FSM bits at clock $t+1$ and the FSM at clock t we need some more steps. At first, since $0 \leq F(t) \leq 4$, we can express it in its binary form:

$$F(t) = f_2(t)2^2 + f_1(t)2 + f_0(t).$$

We can obviously view $F(t)$ as a function $\mathbb{F}_2^4 \to \mathbb{Z}$ with variables set $\{x(t), y(t+6), z(t), u(t+6)\}$ and therefore $f_0(t), f_1(t), f_2(t)$ as Boolean functions $\mathbb{F}_2^4 \to \mathbb{F}_2$ with the same variables set. We can get the Algebraic Normal Form of these functions following the procedure shown in section 2.1.2 which one can find

also in [19]. We obtain that

$$\begin{aligned} f_0(t) &= x(t) \oplus y(t+6) \oplus z(t) \oplus u(t+6), \\ f_1(t) &= x(t)y(t+6) \oplus x(t)z(t) \oplus x(t)u(t+6) \oplus y(t+6)z(t) \\ &\quad \oplus y(t+6)u(t+6) \oplus z(t)u(t+6), \\ f_2(t) &= x(t)y(t+6)z(t)u(t+6). \end{aligned}$$

We consider now the following sum of integers

$$F(t) + C(t) = f_2(t)2^2 + f_1(t)2 + f_0(t) + d(t)2 + c(t)$$

whose binary representation, as a result of carries of the addition modulo 2^2, is

$$\begin{aligned} F(t) + C(t) = (f_2(t) \oplus f_1(t)d(t) \oplus f_0(t)c(t)d(t) \oplus f_1(t)f_0(t)c(t))2^2 \\ + (f_0(t)c(t) \oplus f_1(t) \oplus d(t))2 + (f_0(t) \oplus c(t)). \end{aligned}$$

By dividing by 2, we obtain that

$$\begin{aligned} \frac{F(t) + C(t)}{2} = (f_2(t) \oplus f_1(t)d(t) \oplus f_0(t)c(t)d(t) \oplus f_1(t)f_0(t)c(t))2 \\ + (f_0(t)c(t) \oplus f_1(t) \oplus d(t)) + (f_0(t) \oplus c(t))2^{-1}. \end{aligned}$$

and therefore, since the term $f_0(t) \oplus c(t)$ can be only equal to 0 or 1, it holds that

$$\begin{aligned} \left\lfloor \frac{F(t) + C(t)}{2} \right\rfloor = (f_2(t) \oplus f_1(t)d(t) \oplus f_0(t)c(t)d(t) \oplus f_1(t)f_0(t)c(t))2 \\ + (f_0(t)c(t) \oplus f_1(t) \oplus d(t)). \end{aligned}$$

Since, by definition, we have that

$$G(t+1) = g_1(t+1)2 + g_0(t+1) = \left\lfloor \frac{F(t+1) + C(t+1)}{2} \right\rfloor$$

we can compare the terms and easily conclude that

$$\begin{aligned} g_1(t+1) &= f_2(t+1) \oplus f_1(t+1)d(t+1) \oplus f_0(t+1)c(t+1)d(t+1) \\ &\quad \oplus f_1(t+1)f_0(t+1)c(t+1), \\ g_0(t+1) &= f_0(t+1)c(t+1) \oplus f_1(t+1) \oplus d(t+1). \end{aligned}$$

By using again the definition 3.1 we obtain that

$$\begin{aligned} d(t+2) &= f_2(t+1) \oplus f_1(t+1)d(t+1) \oplus f_0(t+1)c(t+1)d(t+1) \\ &\quad \oplus f_1(t+1)f_0(t+1)c(t+1) \oplus d(t+1) \oplus c(t), \\ c(t+2) &= f_0(t+1)c(t+1) \oplus f_1(t+1) \oplus d(t+1) \\ &\quad \oplus c(t+1) \oplus d(t) \oplus c(t). \end{aligned}$$

Finally, we can substitute the ANF of the functions $f_0(t), f_1(t), f_2(t)$ and putting all together obtain the evolution of the state from $t = 0$ to $t = 1$.

$$\begin{cases} x(25) &= x(0) \oplus x(5) \oplus x(13) \oplus x(17), \\ y(31) &= y(0) \oplus y(7) \oplus y(15) \oplus y(19), \\ z(33) &= z(0) \oplus z(5) \oplus z(9) \oplus z(29), \\ u(39) &= u(0) \oplus u(3) \oplus u(11) \oplus u(35), \\ c(2) &= g_0', \\ d(2) &= g_1' \end{cases} \tag{3.2}$$

where the non-linear polynomials g_0', g_1' are defined as

$$\begin{aligned} g_0' &= x(1)c(1) \oplus y(7)c(1) \oplus z(1)c(1) \oplus u(7)c(1) \oplus x(1)y(7) \oplus x(1)z(1) \\ &\quad \oplus x(1)u(7) \oplus y(7)z(1) \oplus y(7)u(7) \oplus z(1)u(7) \oplus c(1) \oplus d(1) \\ &\quad \oplus c(0) \oplus d(0), \\ g_1' &= x(1)y(7)z(1)u(7) \oplus x(1)y(7)d(1) \oplus x(1)z(1)d(1) \oplus x(1)u(7)d(1) \\ &\quad \oplus y(7)z(1)d(1) \oplus y(7)u(7)d(1) \oplus z(1)u(7)d(1) \oplus x(1)c(1)d(1) \\ &\quad \oplus y(7)c(1)d(1) \oplus z(1)c(1)d(1) \oplus u(7)c(1)d(1) \oplus x(1)y(7)z(1)c(1) \\ &\quad \oplus x(1)y(7)u(7)c(1) \oplus x(1)z(1)u(7)c(1) \oplus y(7)z(1)u(7)c(1) \\ &\quad \oplus d(1) \oplus c(0). \end{aligned}$$

Finally, for all $t \geq 0$ the keystream bits of the cipher E0 are computed as the sum

$$k(t) = x(t \oplus 1) \oplus y(t \oplus 7) \oplus z(t \oplus 1) \oplus u(t \oplus 7) \oplus c(t \oplus 1) \in \mathbb{F}_2.$$

The system 3.2 can be regarded as the evolution of the internal state $v(t) \in \mathbb{F}_2^{132}$ of the difference stream cipher E0. The reader can find a theoretical introduction and some useful results regarding difference stream ciphers in [78].

By Bluetooth specifications [14], to initialize the state, the following steps are followed:

1. The key encryption is used together with the 48 bits Bluetooth address and the 26 bits master counter to fill the 132 bits state;

2. The $v(t)$ transformation is performed until $t = 39$ (when the last bit from the starting state disappears) without generating keystream. Both c_{39}, c_{39-1} are reset so that their values don't have any importance to this point;

3. Other 200 clocks are performed generating output symbols, getting to $t = 239$. At this point, the last 128 bits of the keystream produced are used to fill the keystream generator's initial state.

As anticipated, this means that to retrieve the original encryption key an attack to the keystream generator should be applied twice. Moreover, this strategy can be used only if the attack is based on a short keystream (i.e. less than 128 keystream bits).

3.2 The algebraic attack on E0

Thanks to the results in [78], we have that the explicit difference system (3.2) of E0 is invertible with inverse system

$$\begin{cases} x(25) &= x(20) \oplus x(12) \oplus x(8) \oplus x(0), \\ y(31) &= y(24) \oplus y(16) \oplus y(12) \oplus y(0), \\ z(33) &= z(28) \oplus z(24) \oplus z(4) \oplus z(0), \\ u(39) &= u(36) \oplus u(28) \oplus u(4) \oplus u(0), \\ c(2) &= h_0, \\ d(2) &= h_1 \end{cases}$$

where the polynomials h_0, h_1 are defined as

$$\begin{aligned} h_0 &= x(24)y(24)z(32)u(32) \oplus x(24)y(24)z(32)c(1) \oplus x(24)y(24)u(32)c(1) \\ &\oplus x(24)z(32)u(32)c(1) \oplus y(24)z(32)u(32)c(1) \oplus x(24)y(24)d(1) \\ &\oplus x(24)z(32)d(1) \oplus y(24)z(32)d(1) \oplus x(24)u(32)d(1) \oplus y(24)u(32)d(1) \\ &\oplus z(32)u(32)d(1) \oplus x(24)c(1)d(1) \oplus y(24)c(1)d(1) \oplus z(32)c(1)d(1) \\ &\oplus u(32)c(1)d(1) \oplus d(1) \oplus d(0), \\ h_1 &= x(24)y(24)z(32)u(32) \oplus x(24)y(24)z(32)c(1) \oplus x(24)y(24)u(32)c(1) \\ &\oplus x(24)z(32)u(32)c(1) \oplus y(24)z(32)u(32)c(1) \oplus x(24)y(24)d(1) \\ &\oplus x(24)z(32)d(1) \oplus y(24)z(32)d(1) \oplus x(24)u(32)d(1) \oplus y(24)u(32)d(1) \\ &\oplus z(32)u(32)d(1) \oplus x(24)c(1)d(1) \oplus y(24)c(1)d(1) \oplus z(32)c(1)d(1) \\ &\oplus u(32)c(1)d(1) \oplus x(24)y(24) \oplus x(24)z(32) \oplus y(24)z(32) \oplus x(24)u(32) \\ &\oplus y(24)u(32) \oplus z(32)u(32) \oplus x(24)c(1) \oplus y(24)c(1) \oplus z(32)c(1) \\ &\oplus u(32)c(1) \oplus c(1) \oplus c(0) \oplus d(0). \end{aligned}$$

The invertibility of the system (3.2) allows us to attack equivalently any internal state. A convenient choice consists hence in attacking the state corresponding to the clock where the keystream starts to output.

This made possible the usage of a very low number of auxiliary variables (see subsection 2.1.1). Thanks to elimination by the linear difference equation, the polynomial system is indeed defined over the following variable set:

$$\begin{aligned} \{&x(0),\ldots,x(24),y(0),\ldots,y(30),z(0),\ldots,z(32), \\ &u(0),\ldots,u(38),c(0),\ldots,c(B),d(0),\ldots,d(B)\}, \end{aligned}$$

where B is the clock bound. We observe that other partial eliminations could be considered, including the total elimination of all variables except for the initial ones that are

$$\begin{aligned} \{&x(0),\ldots,x(24),y(0),\ldots,y(30),z(0),\ldots,z(32), \\ &u(0),\ldots,u(38),c(0),c(1),d(0),d(1)\}. \end{aligned}$$

However, we have experimented that all these variants increase the degree of the eliminated polynomials in a way that either makes them impossible to be computed or leads to polynomial systems which are more difficult to solve.

Our attack is based on a short keystream. Precisely, we focused on a number K of keystream bits with $51 \leq K \leq 63$ since in this range we have found

very few $\mathbb{F}_2$-solutions of the resulting system. To avoid a huge amount of testing possibilities we choose to consider K odd. We remark that the number of equations and variables and consequently the cost of solving grows significantly with K. Nonetheless, when reducing the number of keystream bits considered one obtains more spurious solutions, namely keys different from the right one but generating the right keystream bits. However, comparing a few extra keystream bits isn't usually a very time-consuming task. A good trade-off consists in using a value of K that lies approximately in the middle of the range $51 \leq K \leq 63$.

In our guess-and-determine strategy, we choose a subset of 83 variables which are exhaustively evaluated over $\mathbb{F}_2^{83}$, leading to Gröbner bases computations that take a few tens of milliseconds on average.

3.3 The 83 variables set

The set of the 83 variables is the following one

$$\{x(0),\ldots,x(24),y(0),\ldots,y(26),y(29),z(0),\ldots,z(10),z(29),\ldots,z(32),\\ u(0),\ldots,u(9),u(35),u(36),u(37),c(0),d(0)\}.$$

The starting point for the choice of this set are 14 "special" variables we have found analysing the detailed structure of the keystream generator we described above. The remaining 69 variables have been obtained through experimental optimization.

The monomial ordering that we use for computing Gröbner bases and normal forms during the attack to E0 is defined as the DegRevLex-ordering over the following variable set

$$\{x(0),y(0),z(0),u(0),x(1),y(1),z(1),u(1),\ldots\}\\ \cup\{c(0),c(1),\ldots\}\cup\{d(0),d(1),\ldots\}$$

induced by putting

$$x(0)\prec y(0)\prec z(0)\prec u(0)\prec x(1)\prec y(1)\prec z(1)\prec u(1)\prec\ldots\\ \prec c(0)\prec c(1)\prec\ldots\prec d(0)\prec d(1)\prec\ldots$$

Let us derive the 14 "special" variables we mentioned above. We start considering the following polynomials which belong to the difference ideal I corresponding to the explicit difference system (3.2)

$$\begin{aligned}
C_0 &= c(2)+x(1)c(1)+y(7)c(1)+z(1)c(1)+u(7)c(1)+x(1)y(7)+x(1)z(1)\\
&\quad +x(1)u(7)+y(7)z(1)+y(7)u(7)+z(1)u(7)+c(1)+d(1)+c(0)+d(0),\\
C_1 &= c(3)+x(2)c(2)+y(8)c(2)+z(2)c(2)+u(8)c(2)+x(2)y(8)+x(2)z(2)\\
&\quad +x(2)u(8)+y(8)z(2)+y(8)u(8)+z(2)u(8)+c(2)+d(2)+c(1)+d(1),\\
D_0 &= d(2)+x(1)y(7)z(1)u(7)+x(1)y(7)d(1)+x(1)z(1)d(1)+x(1)u(7)d(1)\\
&\quad +y(7)z(1)d(1)+y(7)u(7)d(1)+z(1)u(7)d(1)+x(1)c(1)d(1)\\
&\quad +y(7)c(1)d(1)+z(1)c(1)d(1)+u(7)c(1)d(1)+x(1)y(7)z(1)c(1)\\
&\quad +x(1)y(7)u(7)c(1)+x(1)z(1)u(7)c(1)+y(7)z(1)u(7)c(1)+d(1)+c(0).
\end{aligned}$$

If you look carefully, you can see that these equations are from the combiner. In particular, they are the equations representing how to generate $d(2), c(2), c(3)$ from the initial state. We also consider the following polynomials corresponding to the first 3 keystream bits, say $k_0, k_1, k_2 \in \mathbb{F}_2$, of E0

$$\begin{aligned} K_0 &= x(1) + y(7) + z(1) + u(7) + c(1) + k_0, \\ K_1 &= x(2) + y(8) + z(2) + u(8) + c(2) + k_1, \\ K_2 &= x(3) + y(9) + z(3) + u(9) + c(3) + k_2. \end{aligned}$$

We can now eliminate the variables $c(1), c(2), c(3)$ from the polynomials C_0, C_1, D_0 by reducing them modulo the polynomials K_0, K_1, K_2. We get

$$\begin{aligned} G_1 &= d(1) + A_1, \\ G_2 &= d(1) + d(2) + A_2, \\ G_3 &= A_3 d(1) + d(2) + A_4 \end{aligned}$$

where

$$\begin{aligned} A_1 &= u(7)x(1) + u(7)y(7) + u(7)z(1) + x(1)y(7) + x(1)z(1) + y(7)z(1) \\ &\quad + k_0(x(1) + y(7) + z(1) + u(7)) + c(0) + d(0) + x(2) + y(8) + z(2) \\ &\quad + u(8) + k_0 + k_1, \\ A_2 &= u(8)x(2) + u(8)y(8) + u(8)z(2) + x(2)y(8) + x(2)z(2) + y(8)z(2) \\ &\quad + k_1(x(2) + y(8) + z(2) + u(8)) + x(1) + x(3) + y(7) + y(9) + z(1) \\ &\quad + z(3) + u(7) + u(9) + k_0 + k_1 + k_2, \\ A_3 &= u(7)x(1) + u(7)y(7) + u(7)z(1) + x(1)y(7) + x(1)z(1) + y(7)z(1) \\ &\quad + (k_0 + 1)(x(1) + y(7) + z(1) + u(7)) + 1, \\ A_4 &= u(7)x(1)y(7)z(1) + (k_0 + 1)(u(7)x(1)y(7) + u(7)x(1)z(1) \\ &\quad + u(7)y(7)z(1) + x(1)y(7)z(1)) + c(0). \end{aligned}$$

Hence, a part of the polynomial system will result in

$$\begin{cases} d(1) + A_1 &= 0 \\ d(1) + d(2) + A_2 &= 0 \\ A_3 d(1) + d(2) + A_4 &= 0 \end{cases}$$

It is clear that this is impossible if and only if and only if

$$G = (A_3 + 1)A_1 + A_2 + A_4 \neq 0.$$

Note now that the set of $\mathbb{F}_2$-solutions of the equation $G = 0$ is a preimage of the Boolean function $\mathbb{F}_2^{14} \to \mathbb{F}_2$ corresponding to the polynomial G in the 14 variables

$$\{x_1, x_2, x_3, y_7, y_8, y_9, z_1, z_2, z_3, u_7, u_8, u_9, c_0, d_0\}.$$

Note in fact that, when solving the system, the bits of the keystream k_0, k_1, k_2 are evaluated to a certain value. By computing the $\mathbb{F}_2$-dimension of the quotient algebra

$$\mathbb{F}_2[x_1, \ldots, d_0] / \langle G, x_1^2 + x_1, \ldots, d_0^2 + d_0 \rangle$$

we have that the number $\mathbb{F}_2$-solutions of $G = 0$ is exactly 2^{13}, for all bits

$b_0, b_1, b_3 \in \mathbb{F}_2$. Let's point out that, when evaluating the 14 variables, the polynomials A_1, A_2, A_3, A_4 become some constant bits c_0, c_1, c_2, c_3.

$$\begin{cases} d(1) + c_1 & = & 0 \\ d(1) + d(2) + c_2 & = & 0\,, \\ c_3 d(1) + d(2) + c_4 & = & 0 \end{cases}$$

and as soon as an inconsistency comes out of this system the Gröbner basis output 1. This implies that for half of the evaluations of the 14 variables the resolution is extremely fast. In theory, we can precompute the $\mathbb{F}_2$-solutions of the equation $G = 0$ once given the first 3 keystream bits k_0, k_1, k_2 in order to avoid useless Gröbner bases computations.

Remark finally that in our attack to E0, before performing Gröbner bases computations, we eliminate also the variables $c(t+1)$ $(t \geq 0)$ by means of the polynomials

$$x(t+1) + y(t+7) + z(t+1) + u(t+7) + c(t+1) + k(t)$$

where $k(t)$ denotes the keystream bit at clock t.

3.4 Experimental results

In this section, we report the result of the testing activity. After coding the attack on the difference stream cipher E0 by means of Gröbner bases, SAT solvers, and Binary Decision Diagrams we run it on a couple of servers. I was mainly involved in the first two, but results from the latter will also be shown in order to give a full comparison. The servers have the following hardware configurations:

- Intel(R) Core(TM) i9-10900 CPU@2.80GHz, 10 Cores, 20 Threads and 64 Gb of RAM — server A, for short;
- 2 x Intel(R) Xeon(R) Gold 6258R CPU@2.7GHz, 56 Cores, 112 Threads and 768 Gb of RAM — server B, for short.

On both these machines, we install a Debian-based Linux distribution as operating system.

The comparison between Gröbner and SAT solvers was executed on server A. Every single combination of parameters is tested 2^{20} times. More specifically, we evaluate the set of variables with 2^{17} different random guesses and we do this for 2^3 different keys (and so keystreams).

We tested all the odd values of K with $51 \leq K \leq 63$ measuring average, minimum and maximum computing time for performing the DegRevLex Gröbner bases and for SAT solving. All the results are gathered in Table 3.1. The timings are expressed in milliseconds that are denoted as "ms".

Gröbner bases were implemented in SLIMGB of the computer algebra system SINGULAR [44] whereas as a SAT solver we chose CRYPTOMINISAT [82],

a solver derived from Minisat but dedicated to cryptography. We remember that SAT solvers usually work on Conjunctive Normal Form (see section 2.3.1) so that an ANF-to-CNF conversion is needed when trying this resolutive method. However, the conversion time is essentially negligible since we apply it only once and for each evaluation of the 83 variables we just add the corresponding linear equations to the CNF.

TABLE 3.1: GB vs SAT

K	GB avg	GB min/max	SAT avg	SAT min/max
51	31ms	1/411ms	196ms	105/1007ms
53	34ms	2/480ms	220ms	121/876ms
55	41ms	2/522ms	230ms	134/638ms
57	52ms	3/620ms	245ms	143/645ms
59	64ms	3/799ms	283ms	161/777ms
61	79ms	3/1115ms	300ms	174/732ms
63	96ms	4/1287ms	326ms	191/862ms

From the results in the table, one can claim that for this specific problem Gröbner bases do better than SAT solvers with solving times that are about four times better.

According to Section 3.3, one can expect that minimum computing times are obtained for guesses of the 14 special variables such that $G \neq 0$. This also explains the quite significant difference between max and min computing timings.

Another main testing activity involved the estimation of the number of solutions for every different K. In this direction, we gathered information regarding the degree of the Gröbner bases and the average number of spurious solutions we compute by means of such bases. Note that the degree of a basis is the highest degree of its elements up to field equations. Hence, a basis with degree 0 means an impossible system, i.e. no spurious solutions, and a degree 1 basis means we are dealing with a linear basis that can be solved efficiently.

For every K, we tested again 2^{20} polynomial systems and we give the results as a percentage of the total number of Gröbner bases in the table 3.2 below.

TABLE 3.2: GB data

K	deg(GB)=0	deg(GB)=1	deg(GB)=2	deg=1 avg sol	deg=2 avg sol
51	83.781%	15.243%	0.975%	1.442	3.154
53	94.023%	5.971%	0.005%	1.047	3
55	98.438%	1.561%	0.0001%	1.011	3
57	99.613%	0.386%	0%	1.004	
59	99.901%	0.098%	0%	1	
61	99.976%	0.023%	0%	1	
63	99.993%	0.006%	0%	1	

Data gathered show that the average number of spurious solutions for each Gröbner basis drops down very quickly as the number K of keystream

bits slightly increases. Whereas for $K = 51,53,55$ the number of degree 1 bases is quite high and some degree 2 bases appear (even if rarely) when setting $K \geq 59$, more than 99.9% of the Gröbner bases provide no spurious solution. The remaining bases are of degree 1 and have a single solution as the average number of solutions shows. That unique solution can be read immediately from the basis and detected as a spurious one by using a few additional keystream bits. In fact, for $K = 63$ the probability to have spurious solutions is already close to zero. In our tests, Gröbner bases of degrees strictly greater than 2 were not found.

To have a full comparison, we also code a BDD-based algebraic attack to E0 and compare new results with those presented in Table 3.1 and in the literature [54, 80]. Indeed, BDDs have been generally considered the standard in E0 cryptanalysis. For the testing activity we installed BUDDY library package 2.4 [59] on our machines.

At first, we considered the same set of 83 variables previously used with Gröbner bases and SAT solvers. More precisely, we set $57 \leq K \leq 63$, collect several random guesses of 83 variables, use 2^3 random keys and try to recover the remaining 49 key bits.

Regarding the ANDing of the set of BDDs, among the various approaches described in [54] such as sequential ANDing, ANDing with a fixed interval, random ANDing, RSAND and so on, we decide to adopt RSAND because it takes the overall used memory under control, reducing (recursively) the number of BDDs by half until it gets the final BDD.

Despite using RSAND, experimental activities show that none of these tests ended due to lack of memory of our servers even when running on our server B. Therefore, we had to increase the number of fixed variables to 93 leaving only 39 unknown key bits to be recovered. On server A, using a single-thread configuration and a few Mb of memory, we are able to recover 39 unknown key bits in about 0.15 seconds which is much better than the 5 seconds resolution time on a personal computer by the authors of [54]. Interestingly, the best results do not come from the same set of variables found with Gröbner bases and SAT solvers but from evaluating all key bits but a chunk of consecutive bits which includes all 39 variables of the fourth LFSR, namely $u(0),\ldots,u(38)$. Experimental results showed that to increase the number of variables to be solved from 39 to 40,41,42 the best choice is to "free" the last variables of the third LFSR, that is, $z(32),z(31),z(30),\ldots$.

We measured the performances starting from 39 free variables and increasing the count by one variable at a time. As said, when leaving 42 unknown key bits (and 90 fixed), the choice is

$$u(0),\ldots,u(38),z(32),z(31),z(30)\ldots.$$

Table 3.3 shows the results of the experimental activity with BUDDY library 2.4 on server A which is slightly faster than server B. The timings are given in seconds that will be denoted as “s”.

Although we fixed the number K to much lower values than with Gb and

TABLE 3.3: BDD with BUDDY 2.4

key bits	K	exec time	mem used	# of threads
39	40	0.15s	60Mb	1
40	41	1.07s	240Mb	1
41	42	4.75s	725Mb	1
42	43	19.3s	3Gb	1
43	44	94.5s	13Gb	1
44	45	–	out of mem	1

SAT, data show high time and memory-consuming which also grows exponentially fast with the number of unknown key bits. However, one has also to notice only 1 thread was used since BUDDY library does not provide the possibility to run the code on all threads of our servers. Hence, we installed SYLVAN [29], a decision diagram package which support multi-core architectures. Whereas SYLVAN performs slower than BUDDY 2.4 when executed in single thread mode, exploiting the power of the modern multi-core architectures, the advantage of its use becomes more and more evident as the number of unknown key bits to recover increases.

This is proved by our extensive testing whose results are summarized in table 3.4. The testing is performed for $K = 41,43,45$, and for each value we evaluate $91,89,87$ variables with 2^9 random guesses collecting average execution time and memory used. As for GB and SAT, we repeated the tests for 2^3 random keys. However, the last four rows of this table do not refer to 2^{12} different tests — 2^9 random guesses and 2^3 random keys — but to a single execution with a random key due to excessive time consumption.

TABLE 3.4: BDD with SYLVAN

key bits	K	exec time	mem used	# of threads
39	41	1.19s	1.47Gb	112
40	41	1.55s	1.51Gb	112
41	41	1.95s	1.60Gb	112
39	43	1.34s	1.60Gb	112
40	43	2.03s	1.76Gb	112
41	43	4.71s	3.46Gb	112
39	45	3.58s	3.44Gb	112
40	45	5.23s	3.86Gb	112
41	45	14.65s	7.41Gb	112
43	45	68.29s	30Gb	112
44	45	128.37s	36Gb	112
45	46	517.42s	233Gb	112
46	47	—	out of mem	112

3.5 Expected runtime of the attack

The experimental activity showed that our Gröbner bases algebraic attack to E0 is better than the one performed with Sat solvers and with BDDs. We remark that it should be preferred not only when comparing computing times,

but also in memory: our Gröbner bases computations run in less than 0.5 Gb of memory for $K = 63$. For $K = 59$, which we showed to be an acceptable value due to a few spurious solutions, the average solving time employing a Gröbner basis of the polynomial system is about 60 milliseconds. By the observations in section 2.2.1, the sequential running time is about $2^{83} * 60$ ms, that is about 2^{79} seconds, which improves any previous attempt to attack E0 using a short keystream. The complexity 2^{83} also improves the one obtained by BDD-based cryptanalysis which is generally estimated as 2^{86} [50, 54]. We finally observe that the parallelization of the brute force on the 83 variables can be easily used to scale down further the runtime.

Chapter 4

SHA1

In this chapter, we will describe an algebraic attack to SHA-1 (short for Secure Hash Algorithm 1), one of the four cryptographic hash functions in the SHA family. The attack can also be found in [9].

The function was designed by the United States National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST). It is described in FIPS 180-4 Secure Hash Standard [37].

The algorithm, as its predecessor SHA-0, takes as input a bit string of arbitrary length and produces an output message digest of 160-bit. SHA-0 was published in 1993 but only two years after it was replaced by SHA-1 due to a security flaw. SHA-1 was built on top of SHA-0 and fixes the weakness found in its predecessor, and was heavily cryptanalysed since its appearance.

4.1 Related works

A vast literature studied the collision resistance of SHA-1 and SHA-0, see e.g. [1, 10, 11, 17, 18, 26, 43, 58, 83, 84, 85, 92],

In 2005, SHA-1 was also found to be insecure [92], since Wang et al. demonstrated that SHA-1 has a collision resistance strength that is considerably less than 80 bits. Since 2005, attacks to the collision resistance improved their results until in 2017 the first real collision of the full SHA-1 was discovered by Stevens et al. [85]. The practical attack had a computational complexity of $2^{63.1}$ SHA-1 applications.

In 2008, the NIST document SP 800-17 [24], stated that SHA-1 hash function presents *collision resistance* strength of less than 80 bits, *a second preimage resistance* strength of 105-160 bits, and *a preimage resistance* strength of 160 bits. At the time it was written, there had been no known shortcuts for finding preimages of the hash values generated by SHA-1. Because of that, the adoption of SHA-1 is still widespread and even allowed in certain applications by some of the most important standards. To mention one, the use of SHA-1 is allowed in the construction of the widely adopted message authentication code HMAC-SHA1 (see e.g. [8, 24]).

However, preimage and second preimage attacks against SHA-1 do exist even if, to date, none of the attacks present in the literature cover the full SHA-1 algorithm.

The second preimage attack can be seen as a particular case of a collision attack. The best second preimage attack to SHA-1 is a general result for all

narrow-pipe Merkle-Damgård hash functions [49]. When $n = 160$, as is the case for SHA-1, it will take 2^{106} computations to find a second preimage in a 60-byte message.

Anyway, many applications of hash functions rely only on the preimage resistance. The first attack affecting this kind of security was published only in 2008 by De Cannière and Rechberger [28]. The attack is a combination of a "reversing the inversion problem" technique and an algorithm exploiting random graphs theory. Reduced SHA-1 (45 rounds) is attacked with complexity (2^{157}).

Some more attacks followed in the years. Table 4.1 shows the main ones.

TABLE 4.1: preimage attacks on SHA-1

# of rounds	# of blocks	Complexity	Technique	Year	Ref
45	1	2^{157}	Reversing inversion + Graph	2008	[28]
48	1	2^{159}	MiTM	2008	[3]
57	1	$2^{158.7}$	Differential MiTM	2012	[51]
57	2	$2^{158.8}$	Differential MiTM	2012	[51]
62	1	2^{159}	Higher order diff. MiTM	2015	[33]
62	2	$2^{159.3}$	Higher order diff. MiTM	2015	[33]

All the approaches in table 4.1 try to maximize the number of rounds attacked keeping the complexity below the brute force upper bound.

One less explored approach is to attack a hash function practically, that is, with an attack that retrieves the unknown data in a more reasonable time (1 day/week) and to investigate how many rounds can be broken in this way. Be aware that the computational time of the attack may heavily depend on the computational power the attacker has at his disposal.

The most recent work we could find in this direction is from Nejati et al. in 2017 [38]. It is an algebraic attack, therefore a reduced version of SHA-1 is represented as a system of equations and solved with different SAT solvers. The authors claim that, with this method, preimages for more than 23 steps cannot be constructed in a reasonable amount of time even with the latest techniques and hardware. [70, 56, 57]. Moreover, they assert that 27 rounds is their best result when leaving only 40 bits of the message free.

This is also the kind of attack we proposed: an algebraic attack trying to maximize the number r of rounds of the compression function that can be attacked practically.

4.2 The hash function SHA-1

SHA-1 hash function adopts the *Merkle–Damgärd construction* described in section 1.3.3.

The padding for SHA-1 has the purpose of making the total length of a padded message a multiple of 512. The last 64 bits of the padding must contain the length of the message in binary form. The other bits are a 1 followed by 0s until reaching those last 64 bits.

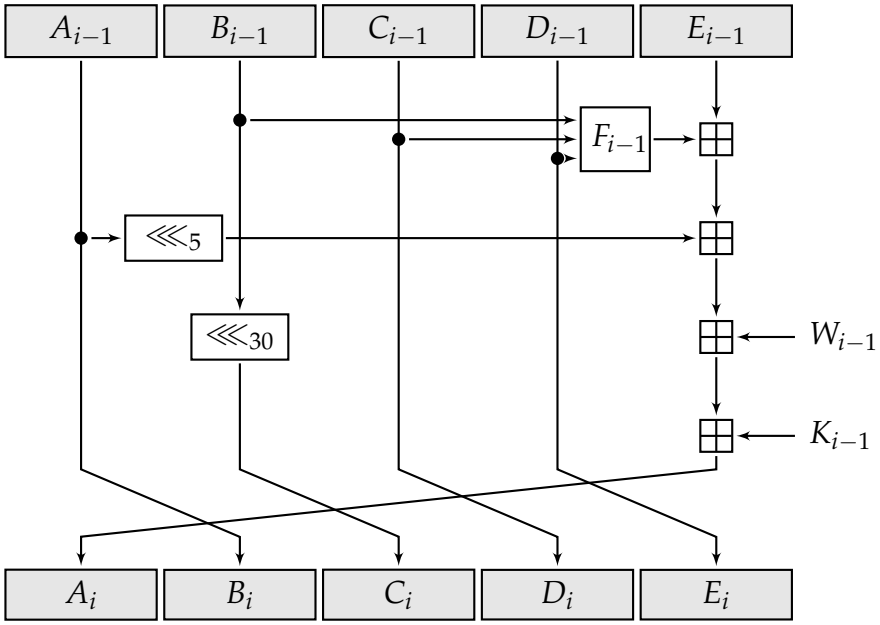

FIGURE 4.1: SHA-1: round function

Then SHA-1 sequentially processes blocks of 512 bits when computing the message digest. The 512-bit blocks are in turn divided into sixteen 32-bit words, which a message schedule extends into eighty 32-bit words applying some linear operations. The message schedule, for $16 \leq i \leq 79$ (starting enumerating the w_i from 0), generates the remaining message words in the following way:

$$w_i = (w_{i-3} \oplus w_{i-8} \oplus w_{i-14} \oplus w_{i-16}) \lll_1 . \tag{4.1}$$

An alternative representation can be found in [89].

At round i the 32-bit word w_{i-1} is used together with the round constant K_{i-1} to modify the 160-bit state $A_{i-1}, B_{i-1}, C_{i-1}, D_{i-1}, E_{i-1}$ by means of the round function in figure 4.1:

$$A_i = K_{i-1} \boxplus (W_{i-1} \boxplus ((A_{i-1} \lll 5) \boxplus (E_{i-1} \boxplus F_{i-1}(B_{i-1}, C_{i-1}, D_{i-1})))), \tag{4.2}$$
$$B_i = A_{i-1}, \tag{4.3}$$
$$C_i = B_{i-1} \lll 30, \tag{4.4}$$
$$D_i = C_{i-1}, \tag{4.5}$$
$$E_i = D_{i-1}. \tag{4.6}$$

The function F_i is defined as:

$$\begin{cases} F(B,C,D) = (B \wedge C) \vee ((\neg B) \wedge D) & \text{if } 0 \leq i \leq 19, \\ F(B,C,D) = B \oplus C \oplus D & \text{if } 20 \leq i \leq 39, \\ F(B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{if } 40 \leq i \leq 59, \\ F(B,C,D) = B \oplus C \oplus D & \text{if } 60 \leq i \leq 79. \end{cases}$$

At the beginning of the algorithm, the state is initialized with a 160-bit

word called initialization vector (IV). Every next round the input state is the output of the preceding application of the round function. The round function is repeated eighty times and the 160-bit final output is the input to a new state that is processed with a new message block. If all the blocks have been used the 160-bit output is the message digest (or hash value).

4.3 Modelling SHA-1 as a system of equations

In this section, we briefly describe how the hash function SHA-1 was algebraically represented for our attack.

Since the algorithm is the repeated application of the compression function, which in turn is a sequence of eighty (almost) equal rounds, it will be enough to explain how one of these rounds is transformed into equations. Actually, our attack is a one-block preimage attack, so that our model does represent the algorithm working on only one block.

Equations 4.3, 4.4, 4.5, and 4.6 are simply equalities (and a rotation) so we will always avoid to use auxiliary variables for these passages. In fact, what every round produces is only a 32-bit word entering the state to the left. The rest of the operations can be seen as a right shift of the words of the state.

Instead, in equation 4.2, we have two non linear operations: addition modulo 2^{32} and the Boolean function F_t. Details regarding how to obtain the representation of these operations can be found in section 2.1.2.

In 4.2, we show the number of variables and equations of the polynomial system over $GF(2)$ for various problems appearing in the SHA-1 hash function.

operation	no. of variables	no. of equations
addition mod 2^n	$4n$	$2n$
F_t	$4n$	n
1-round of SHA-1 compression function	480	608
r-rounds of SHA-1 $(r \geq 16)$	$320r + 480$	$320r - 192$
r-rounds of SHA-1 preimage $(r \geq 16)$	$320r + 480$	$320r + 128$

TABLE 4.2: Number of variables and equations for SHA-1 polynomial system over $\mathbb{F}_2$ as a function of the number r of rounds.

Note that in the second last line we have the number of equations and variables used for representing algebraically r rounds of the compression function whereas in the last line we have the ones for the system representing the preimage attack. The only equations that we need to add are the ones for fixing the input state to the IV and the hash value to the output value we want to find the preimage of and therefore $160 \cdot 2$ equations are needed.

In 4.3, we want to gather information regarding the number of additions and multiplications in various operations. This is useful to get an idea of the amount of "nonlinearity" of the system.

Operation	# addition	# multiplication	Parameters
$\boxplus$ (addition mod 2^{32})	189	93	-
F_t	96	64	$0 \leq t < 20$
	96	96	$40 \leq t < 60$
	96	0	$20 \leq t < 40, 60 \leq t < 80$
The t-th Round	852	436	$0 \leq t < 20$
	852	468	$40 \leq t < 60$
	852	372	$20 \leq t < 40, 60 \leq t < 80$
Message Expansion	$4(r-16)$	0	$r \geq 16$
Final addition	945	465	-

TABLE 4.3: Number of addition (Boolean XOR) and multiplication (Boolean AND) for the system of equations of SHA-1.

4.4 The algebraic attack on SHA-1

Our attack is a preimage attack as intended in section 2.2. We fix the hash value besides the initialization vector and the round constants which are constant values. We work on reduced versions of the compression function trying to maximize the number r of rounds reached by the attack. To do so, we fix some of the message bits, mainly following two different ideas:

- A first approach is to leave at least 160 bits of the message free. Indeed, since SHA-1 hash function has a hash value of 160 bits one could expect to find at least a preimage if so many variables are not fixed. This leaves the attacker the freedom to fix from 0 to 352 variables of the message block.
- We also decided to test the strength of the compression function when leaving less than 160 bits of the message free. In this case, the resulting system is probably unsatisfiable. We decided to fix the values to the correct ones of the preimage thinking of a real case in which the attacker has retrieved those bits in some other way, for instance by a side-channel attack.

The systems obtained are transformed into the CNF form and solved with the SAT solver CryptoMiniSat v5.8.0. [82]. We tested also the performances of Gröbner bases but since they were much worst than SAT ones the results are not shown.

4.5 Experimental results

In this section, I report our experimental results when trying to find a preimage of SHA-1 using SAT solvers.

We run our code on a server and a cluster. The first one is equipped with:

- `CPU: Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz;`
- `Memory: 768GB 2933 MHz;`

- `OS: Ubuntu 18.04.5 LTS.`

The cluster is equipped with:

- `CPU: POWER9, altivec supported`
- `Memory: 319GB 2666 MHz`
- `OS: Red Hat Enterprise Linux Server 7.6 (Maipo)`

Regarding the first kind of attack described in the previous section 4.4, we reached the round 23, confirming the claims in [38]. It is interesting to notice that the SAT solver does not always benefit from having fewer free variables. In fact, the results of the experiments showed that the best solving times when leaving at least 160 bits from the message free are obtained when the free bits are 440. Actually, this is not even counter-intuitive: although setting some values means reducing the number of variables and simplifying many of the equations it also reduces the number of possible preimages. A Sat solver may suffer from a reduction in the number of the solutions because of how its solving algorithms work and because it stops as soon as a single solution is found.

Since reversing 23 rounds is quite time-consuming extensive testing, which provides an average from 100 tests, has been done for 20 and 21 rounds. In these tests we vary (look also at section 2.3.1):

- The number of **threads** used;
- The **polarity** mode: auto, false, random, true;
- The **restart** policy: geom, glue, luby;

The results were similar both with the server and the cluster with the latter being slightly more performing so we will omit data collected on the server. They are gathered in Table 4.4. We also set a time limit for the solving: 10 seconds for round 20 and 180 seconds for round 21. Not all the systems are solved within the time limit: in the table, red numbers represent the number of systems solved out of 100.

The table shows that using a higher number of threads gives usually better results. The best results are usually obtained with polarity modes "auto" or "random" and with the restart policy "geom". However, none of the combinations (polarity mode-restart policy) performs always particularly better than the others.

Fixing only a few message bits (72), choosing the best possible set of variables is not crucial. But when working on the second approach, where more than 352 of the variables representing the input block are fixed, it becomes really important to choose them carefully.

To do so, we had to carefully analyse the message schedule introduced in section 4.2. Firstly, in table 4.5 it is shown what message words are used for computing r rounds of SHA-1 function. In particular, the green color is used for a message word that is used in the exact round we are considering while

polar	restart	n.o.t.	round 20	round 21
auto	geom	1	100/1.551	16/163.679
		2	100/0.980	35/144.212
		4	100/0.703	46/135.598
		8	100/0.595	50/131.754
	glue	1	82/4.916	21/158.565
		2	100/1.195	28/154.708
		4	100/0.852	30/152.602
		8	100/0.694	25/157.295
	luby	1	100/1.903	29/153.644
		2	100/0.966	28/153.768
		4	100/0.783	40/141.568
		8	100/0.627	33/148.349
false	geom	1	100/1.577	15/166.735
		2	100/1.162	19/162.413
		4	100/0.838	30/152.825
		8	100/0.593	50/137.270
	glue	1	98/2.148	16/166.044
		2	100/1.080	15/164.878
		4	100/0.875	25/158.758
		8	100/0.700	21/161.685
	luby	1	100/1.955	15/167.913
		2	100/0.980	18/162.320
		4	100/0.660	33/152.766
		8	100/0.591	44/139.372

polar	restart	n.o.t.	round 20	round 21
rnd	geom	1	100/1.537	11/169.224
		2	100/0.872	17/165.370
		4	100/0.539	28/152.521
		8	100/0.448	67/110.260
	glue	1	98/3.336	8/174.637
		2	100/1.147	15/166.956
		4	100/0.771	17/164.757
		8	100/0.604	13/169.859
	luby	1	98/2.198	9/170.210
		2	100/1.016	19/161.224
		4	100/0.638	39/147.540
		8	100/0.483	43/136.528
true	geom	1	100/1.472	32/150.264
		2	100/0.856	33/150.108
		4	100/0.729	35/147.536
		8	100/0.611	59/124.257
	glue	1	99/2.507	26/157.044
		2	100/1.071	39/142.236
		4	100/0.877	33/145.281
		8	100/0.701	31/153.147
	luby	1	99/2.330	22/163.257
		2	100/0.954	33/152.180
		4	100/0.748	45/139.531
		8	100/0.579	45/133.082

TABLE 4.4: Average times to reverse SHA-1 on the cluster with Cryptominisat.

the red one is used for words that are used in some previous round function applications. Round 22 is the first one where all the words from the input block are used and this seems to affect the solving complexity of the system, which increases considerably with respect to round 21.

		Message words															
		w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}
Rounds	17																
	18																
	19																
	20																
	21																
	22																

TABLE 4.5: Message words involved at different rounds.

Moreover, considering that the elements used to compute round i are a) the output state of the previous round b) the known constant K_{i-1} c) the message word W_{i-1}, one has that fixing the first s message words, that is $32 \cdot s$ variables, gives the possibility to compute the first s rounds and obtain the state A_s, B_s, C_s, D_s, E_s. We tested the performances when fixing all words but two consecutive ones (64 bits free) and the best results are obtained when only the last two (w_{14}, w_{15}) have to be retrieved, confirming the intuition above. Times are a result of a single computation, not an average, and are shown in table 4.6.

This technique lets us get to reverse 27 rounds of SHA-1 matching the results [38] where however the authors were leaving only 40 bits of the preimage free.

		rounds						
		21	22	23	24	25	26	27
unknown words	14&15	×	0.02	0.03	0.52	5.02	3.12	1059.25
	13&14	0.01	0.02	0.04	12.52	15.30		
	12&13	0.01	0.14	0.61	104.52			
	11&12	0.02	0.02	0.54	118.29			
	10&11	0.01	3.29	99.65				
	9&10	1.73	127.35					
	8&9	367.66						
	7&8	0.08						
	6&7	1.21						
	5&6	35957.59						

TABLE 4.6: SAT-based attack on two unknown words. Time is in seconds.

Following a similar idea, it was decided to solve the polynomial system by fixing instead the last message words, that is, when trying to retrieve a 64-bit preimage for r rounds, fixing the words $w_{r-14}, \ldots, w_{r-1}$. We remark that being the message schedule a linear expansion, we can try to retrieve any sixteen consecutive words and generate the first sixteen from them. The new approach seems really promising: when fixing those words, the output state $A_{r-14}, B_{r-14}, C_{r-14}, D_{r-14}, E_{r-14}$ can be immediately computed deterministically. In table 4.7, the results obtained solving the resulting systems for 30 different assignments of the variables are shown. In particular, minimum, maximum, and average times are provided.

rounds	time (s)	min	max
26	1.37s	0.00	17.32
27	4.98s	0.19	16.01
28	131.02s	11.06	334.56
29	4588.24s	7.45	35114.09
30	9924.96s		
31	16255.33s		

TABLE 4.7: Words $w_{r-14} \rightarrow w_{r-1}$ fixed

With the new technique, we managed to reverse 31 rounds of the SHA-1 hash function highly improving the attack in [38]. Due to time constraints rounds 30,31 are executed a single time.

The same technique was applied when leaving fewer preimage bits free (80,96,112) and tested on a single execution. Results are shown in table 4.8. Note that for 80 and 112 free bits, we fix the last 13 and 12 words plus the second half of the 14-th and 13-th last words respectively.

Number of free bits			
rounds	80	96	112
20			0.02s
21		0.01s	0.43s
22		0.85	6.98s
23	0.02s	0.02s	4.87s
24	2.03s	19.62s	40.91s
25	1.70s	699.00s	498.99s
26	15.99s	172552.81s	528294.05s
27	134.28s		
28	4717.71s		
29	593366.71s		

TABLE 4.8: Results for 80,96,112 bits free when fixing the last words

4.6 Final observations

The algebraic attack to SHA1 reveals some weaknesses in the message schedule which let us outperform all the previous results obtained in the literature in the same direction. However, a practical attack on the whole primitive-function seems very unlikely. Indeed, even supposing some preimage bits are recovered by some other attack, this attack breaks 31 rounds which is quite more than what showed in any other work but still really far from the 80 rounds of the full algorithm.

It was still interesting also investigating the behavior of SAT solvers when solving polynomial systems with different features and comparing different restart policies and polarity modes.

Part II

Differential and linear cryptanalysis

Chapter 1

Differential cryptanalysis

Differential cryptanalysis, first proposed by Biham and Shamir in 1990 [13], is a sort of statistical cryptanalysis that may be used to attack block and stream ciphers or even cryptographic hash functions. It consists in studying how a certain input difference applied to the plaintext propagates throughout the algorithm resulting in a final output difference. Since it is really unlikely to find an input-output difference couple holding with probability 1 for a full cipher, the attacker looks for the input-output difference couple which is most likely to appear for any input message M. Then, if a couple with a high probability is found, one can mount a chosen-plaintext partial key-recovery attack to the primitive.

In this chapter we will use the following notations:

- P is the plaintext (or message);
- C is the ciphertext;
- K is the key;
- Δ_P is the input difference;
- Δ_k is the input difference applied to the key in a related key scenario;
- Δ_C is the output difference;
- $P' = P \oplus \Delta_P$ is the plaintext obtained by applying the input difference to the original plaintext;
- $C' = C \oplus \Delta_C$ is the ciphertext obtained applying the output difference to the original ciphertext;
- $K' = K \oplus \Delta_k$ is the key obtained by applying the input difference to the original key.

When the difference applied to the key Δ_k is fixed at 0 the scenario is called "single-key" whereas the scenario is called "related-key" if the difference Δ_k is left free to vary.

1.1 Formal definitions

Let's introduce this in a more formal way starting by defining what is a difference. In my research the focus was on xor differential cryptanalysis, that is the difference is applied to the input message by XORing it to the original message. Another possibility is for example to apply the difference by a modular addition.

Definition 11 (Difference)**.** *Let $(\mathcal{G},+)$ be an Abelian group. The difference between two elements x and x' is defined as $x + x'^{-1}$, where $+$ is the group operation and x'^{-1} is the inverse element of x' in $\mathcal{G}$.*

When dealing with cryptographic primitives the group will usually be $\mathbb{F}_2^n$ and so the $+$ is the XOR boolean operator and the inverse of an element is the element itself.

The differential probability of a vectorial Boolean function (BF) is defined as follows.

Definition 12 (Differential probability)**.** *Let $(\mathcal{G}_1,+_{\mathcal{G}_1})$ and $(\mathcal{G}_2,+_{\mathcal{G}_2})$ two groups whose elements can be represented with bit strings of, respectively, n_1 and n_2. Let $\Delta x \in \{0,1\}^{n_1}, \Delta y \in \{0,1\}^{n_2}$ be fixed n-bit strings, representing two group differences in, respectively, $\mathcal{G}_1$ and $\mathcal{G}_2$. The Differential Probability (DP) of a function f (dp^f) from $\mathcal{G}_1$ to $\mathcal{G}_2$ is the probability with which Δx propagates to Δy through the function f, computed over all n-bit input x:*

$$\mathsf{dp}^f(\Delta x \to \Delta y) = 2^{-n} \cdot |\{x \in \mathcal{G}_1 : (f(x +_{\mathcal{G}_1} \Delta x) = f(x) +_{\mathcal{G}_2} \Delta y\}|.$$

Definition 13 (DDT)**.** *A* difference distribution table *(DDT) of a vectorial BF is a table that shows the probability of every possible input and output difference couple.*

Note that the DDT of a $n_1 \times n_2$ Boolean function can be represented as a $2^{n_1} \times 2^{n_2}$ matrix and thus only the DDT of a function of small size is practical to be generated. This is the case for instance of S-Boxes which usually have 4 or 8-bit input and output words and therefore the DDT is a 256×256 matrix in the latter. Instead, it is clearly infeasible to obtain the DDT of a block cipher, considering its large block size and the fact that every master key instantiates a different permutation. Nonetheless, it is important to consider that block ciphers are usually iterative functions, that is a function of the type $f_{r-1} \circ \ldots \circ f_0$.

Definition 14 (Differential characteristic)**.** *For an iterative function $f = f_{r-1} \circ \ldots \circ f_1 \circ f_0$, a sequence of differences*

$$\Delta_0 \xrightarrow{f_0} \Delta_1 \xrightarrow{f_1} \ldots \Delta_{r-1} \xrightarrow{f_{r-1}} \Delta_r$$

is called an r-round differential characteristic of f.

Definition 15 (Differential)**.** *A differential $(\Delta x \to \Delta y)$ over $f = f_{r-1} \circ \ldots \circ f_1 \circ f_0$, contains all differential characteristics with $\Delta_0 = \Delta x$, and $\Delta_r = \Delta y$.*

Lemma 1.1.1. *The probability of a differential is the sum of that of all its characteristics.*

The base 2 logarithm of the reciprocal of the probability is called weight and is usually what the attacker keeps track of. We remark that, if working on weights, one has to find the differential with the lowest weight, which corresponds to the highest probability.

The difference between differential and differential characteristics is crucial. In a partial key-recovery attack such as the one we will describe one actually needs the best differential and its probability. Nevertheless, many papers in the literature look for the best possible differential characteristics of a primitive and base their attack on that since it is much simpler than finding the best differential. It is also what I've been doing in my research (see chapter 3). But actually one can have a big gap between the two probabilities (for more, look at [2]).

1.2 Differential analysis of main operators

A cryptographic primitive is usually a sequence of linear and nonlinear operations starting from the plaintext (and eventually the key) and ending at the ciphertext. Differential characteristics can be obtained by concatenating all input/output difference couples and their probabilities obtained for every single operation. Computing the precise probability of a differential characteristic isn't simple anyway. In particular, to "concatenate" the probabilities one has to do some fundamental assumptions, such as *Markov cipher* assumption, the *Hypothesis of stochastic equivalence* and the *Hypothesis of independent round keys* (see e.g. [61, Section 2.2.1]).

In my research I worked on the assumption of independence between the various operations so that the resulting probability when considering two consecutive BFs is computed by the following:

Proposition 1.2.1. *Let f_1 and f_2 be two boolean functions*

$$f_1 : \mathbb{F}_2^l \to \mathbb{F}_2^m$$
$$f_2 : \mathbb{F}_2^m \to \mathbb{F}_2^n.$$

and let $\Delta_x \in \mathbb{F}_2^l$, $\Delta_y \in \mathbb{F}_2^m$ and $\Delta_z \in \mathbb{F}_2^n$ be three differences such that

$$\mathsf{dp}^{f_1}(\Delta x \to \Delta y) = p_1 \qquad \mathsf{dp}^{f_2}(\Delta y \to \Delta z) = p_2.$$

Then, we have

$$\mathsf{dp}^{f_2 \circ f_1}(\Delta_x \to \Delta_z) = p_1 \cdot p_2.$$

Note that the relative weights of the functions will be summed.

In this section, we will provide a detailed description of how differential cryptanalysis deals with the most common operations in a cryptographic primitive.

Firstly, we remark that no differences are applied to the constants used in the algorithm (by definition of what a constant is).

Linear operations will never generate a probability, meaning a certain input difference corresponds deterministically to a certain output difference which can be obtained using the following:

Proposition 1.2.2. *Let $f : \mathbb{F}_2^m \to \mathbb{F}_2^n$ be a linear function and let $\Delta_x \in \mathbb{F}_2^m$ be an input difference. Then*

$$f(x + \Delta_x) = f(x) + f(\Delta_x),$$

that is $\Delta_y = f(\Delta_x)$.

The most common linear operations are:

- **XOR**: exclusive or between $x_1, \ldots, x_m$ with $x_i \in \mathbb{F}_2^n$ for every i can be seen as a function is $f : \mathbb{F}_2^{m \cdot n} \to \mathbb{F}_2^n$, where the input $\boldsymbol{x}$ is $x_1 \parallel x_2 \parallel \ldots \parallel x_m$. Note that in a xor of an input with a constant the input difference is simply XORed to 0;
- **linear layer** and **mixcolumn**: for linear layer proposition 1.2.2 directly applies considering the linear function f to be the multiplication with the matrix. As in section 2.1.2, mix column can be transformed simply in a linear layer. Actually, mix column will be one of the main characters in the strategy we will explain in section 1.3;
- **rotation**;
- **shift** is a linear operation and the proposition 1.2.2 could be applied but one has to be careful with dependencies. For instance, let us suppose a part of the algorithm is the following:

  $$(S(\boldsymbol{x_1}) \parallel S(\boldsymbol{x_2}) \parallel \ldots \parallel S(\boldsymbol{x_8})) \ll_{-4},$$

  with S being a 4×4 S-Box, namely a 32-bit word that is the concatenation of eight parallel S-Boxes is shifted by 4 to the left. Then, the probability generated by the first left S-Box shouldn't be considered: the 4-bit output disappears in favor of the 4 zeros added to the right.

In a nonlinear operation usually for a single nonzero input difference we have that more than one output difference is possible so that a probability is generated. A special case is the **NOT** operation which is treated as the xor with the constant word of all 1s.

For an operation with small input and output bit sizes, one can always compute the DDT. This can be done for example for **AND**, **OR** that operate bitwise or for **S-Boxes** which have usually 4 or 8 input and output bit size.

Unfortunately, it is practically inefficient to use DDTs when dealing with addition modulo 2^{32} (**MODADD**) because its dimension would be $2^{32} \times 2^{32}$. For computing the probability of modular addition one can use the Lipmaa-Moriai algorithm proposed in [60]. For any x, y and z we define $eq(x,y,z) := (\neg x \oplus y) \wedge (\neg x \oplus z)$, that is $eq(x,y,z) = 1 \Leftrightarrow x_i = y_i = z_i$, and for any n, $mask(n) := 2^n - 1$. w_h is the hamming weight of a string of bits, namely the number of bits equal to 1. The algorithm works as follows:

Algorithm 1: Lipmaa-Moriai algorithm for modular addition

Input : $\Delta_{x_1}, \Delta_{x_2}, \Delta_y$
Output: $\mathsf{dp}^{modadd}(\Delta_{x_1} \| \Delta_{x_2} \mapsto \Delta_y)$
if $eq(\Delta_{x_1} \ll 1, \Delta_{x_2} \ll 1, \Delta_y \ll 1) \wedge (\Delta_{x_1} \oplus \Delta_{x_2} \oplus \Delta_y \oplus (\Delta_{x_2} \ll 1)) \neq 0$ **then**
  then return 0 ;
else
  return $2^{-w_h(\neg eq(\Delta_{x_1}, \Delta_{x_2}, \Delta_y) \wedge mask(n-1))}$;
end if

1.3 Two steps strategy

Differential cryptanalysis has two main limits: the best differential characteristics (or differential) for r rounds of the function a) may have a too low probability or b) may be impossible to compute in a practical time. For this reason, the attacker usually tests the attack on a reduced version of the cipher.

A really common approach that tries to improve the efficiency of the search of the trail is to divide the search into two steps [39].

In the first step, the state of the function evolving from plaintext to ciphertext is divided into words of a certain length. Then a bit is used for each of these words. These bits have value 1 if a nonzero difference is applied to the word they represent and 0 otherwise.

Example 1.3.1. *Let* $\Delta_x = 11010101010011110000000010100000$ *be an input difference and the word size we work with is* 4 *then we define* $\Delta W_x = 11110010$.

The first step aims at finding a trail that minimizes the number of active S-Boxes, that are the S-Boxes with an active input word, which is usually called a "truncated trail". This trail doesn't describe how a bit difference propagates throughout the function but describes only how the activity of the words behaves.

In the second step, the attacker tries to find a valid trail with the bit values matching the active words of one of the solutions from step one, i.e. a certain word has nonzero value only if the corresponding bit in the first step solution is 1.

Note that this strategy works pretty well with functions containing S-Boxes and mix columns, whose input and output bit size is usually used as the word size chosen. It is also acceptable to have rotations/shifts of an amount multiple of the word size. Instead, if one has a modular addition the strategy usually isn't useful at all: when a modular addition between two input differences is computed every word of the output differences is likely to be activated because of the carries.

Let's explain how to deal with some functions when performing step one of the search. For step two one can just look at the previous section.

- **XOR**: since XOR is a bitwise operation it is sufficient to look at how it works on single words. We recall that a word is defined by a single bit. The XOR works as follows:

X	Y	Z
0	0	0
1	0	1
0	1	1
1	1	0/1

.

Consider that if two input words are nonzero their sum is 0 if and only if all their bits are equal and then the last line holds because in step one we don't see the bit values of the words. This fact arises the problem of impossible solutions (solutions from step one to which no possible bit solution exists): a solver trying to minimize the number of active S-Boxes will usually prefer taking the 0 output when XORing two nonzero words instead of the 1 but this means making the strong assumption that the two input words are bitwise equal.

- **SBOX**: S-Boxes are trivial to work with in step one. They usually operate on a single word and the output word is active if and only if the input word is.
- **Mix column**: for mix column we use an object related to linear transformations acting on byte vectors called the *Branch number* (see [23] for more). We can define a function *weight* W that, applied to a vector, counts the number of nonzero words in that vector, with a word being a string of a certain amount of bits (if the words are single bits this is usually called the Hamming weight). Then, the branch number of a linear transformation F is

$$\min_{a \neq 0}(W(a) + W(F(a))),$$

where a are all the possible inputs of the linear function. The word size depends usually on the dimensions of the S-Boxes and is commonly 4 or 8. For implementing a mix column in the first step we constrain the total number of 1s in the input and output bits to be either 0 or greater or equal to the branch number, which can be computed easily. Let M denote the matrix representing the transformation, than the branch number is the minimum weight of the rows of matrix $G = [I|M]$ with I the identity matrix.
- **Rotation/Shift**: they work as in the ordinary case but the amount of the rotation or the shift is divided by the word size. For example, if a 32-bit word I is left-rotated by 8 and our word size = 4 then we will left rotate the 8-bit word representing the activity of I by 2.

1.4 Partial key-recovery differential attack

Once a differential is found the attacker can mount a chosen-plaintext partial key-recovery attack to the cipher. We will briefly present the method described in [47] for a single key scenario.

Let f_i be the function for i rounds and (Δ_P, Δ_C) a differential for $r-1$ rounds holding with a suitably large enough probability p. Let's denote by K the key to recover and by K_i the word coming from the key schedule that is used in the i-th round of the function, called subkey for round i. The targets of the attack are some of the bits from the last subkey K_r, and will be called target subkey bits. In particular, we consider the bits of the subkey that are used to partially decrypt the last round of the cipher, i.e. to decrypt only the parts of the ciphertext corresponding to S-Boxes with a nonzero output difference (or active S-Boxes).

For all possible combinations of values assigned to the target partial subkey one proceeds as follows:

1. Set a counter $c = 0$;
2. Take one of the couples of plaintexts (P, P') with $P \oplus P' = \Delta_P$ and relative ciphertexts after r rounds (C_r, C'_r);
3. Partially decrypt the last round of the function f to obtain some parts of (C_{r-1}, C'_{r-1}). Compute the partial output difference between the ciphertexts for round $r-1$: if it matches the expected difference Δ_C increase the counter by 1;
4. If there are other couples (P, P') available go to 2) otherwise return counter c.

The combination of values for which the counter reaches the highest number is the one we believe to be correct. Indeed, if the correct subkey bits are used the partial one round decryption is correct and it is true that $C_{r-1} = f_{r-1}(K, P)$ and $C'_{r-1} = f_{r-1}(K, P')$ whereas with wrong values of the subkeys C_{r-1} and C'_{r-1} should have arbitrary values. Then, we expect to obtain the highest probability with the correct partial subkey bits and a probability close to $1/2$ for all the other choices.

The complexity of the attack is estimated in terms of the number of plaintext couples needed. We assume that if we have any number of such couples, we are able to process all of them. The number of plaintexts we expect to be needed is

$$N_D \approx \frac{c}{p},$$

where c is some small constant. Consider that, being p the probability of the input/output difference couple, the output difference Δ_C is expected to appear once every p plaintext couples. Testing some multiple of p plaintext couples should be enough to underline the difference in the counter between a random target subkey bits guess and the correct one.

Chapter 2

Linear cryptanalysis

Linear cryptanalysis was introduced by Matsui (see e.g. [67] and [68]) in the early 1990s and it is one of the most widely used attacks on block ciphers, together with the differential one introduced in chapter 1. Even so, it can be efficiently applied also against stream ciphers.

Linear cryptanalysis aims to find a linear relation between the plaintext, the key, and the ciphertext holding with a probability as higher or lower as possible of $\frac{1}{2}$. If a linear relation with a good probability is found, one can mount a known-plaintext partial key-recovery attack to the primitive.

We talk about a single key scenario if the key and the key schedule are fully ignored in the linear trail search, a related key scenario otherwise.

2.1 Formal definitions

More formally, in linear cryptanalysis one would like to obtain a relation of this kind:

$$P_{i_1} \oplus \cdots \oplus P_{i_l} \oplus K_{j_1} \oplus \cdots \oplus K_{j_m} \oplus C_{k_1} \oplus \cdots \oplus C_{k_n} = 0, \tag{2.1}$$

holding with a certain probability p for randomly given plaintext and corresponding ciphertext.

The following definition lets us give another representation of this equation:

Definition 16 (Linear combination)**.** *Let $x, \Gamma \in \mathbb{F}_2^n$ be two bit strings. The* linear combination *of x and Γ is the standard dot product between the two, i.e. $\Gamma \cdot x = \Gamma_{n-1}x_{n-1} + \ldots + \Gamma_1 x_1 + \Gamma_0 x_0 \in \mathbb{F}_2$. Sometimes Γ is also called a mask for x.*

In this chapter Γ_P, Γ_K, and Γ_C are the masks applied to the plaintext, the key, and the ciphertext, respectively. For the rest of the notations, one can still refer to chapter 1.

Then we can rewrite equation 2.1:

$$\Gamma_P \cdot P \oplus \Gamma_k \cdot K \oplus \Gamma_C \cdot C = c,$$

where $w_h(\Gamma_P) = l$, $w_h(\Gamma_K) = m$, $w_h(\Gamma_C) = n$.

Let us call X the left side of equation 2.1. We use the capital letter X because it can be seen as a random variable taking values in $\{0,1\}$. Then we define $p = P(X = 0)$ and $\epsilon = p - \frac{1}{2}$ so that $\epsilon \in \left[-\frac{1}{2}, \frac{1}{2}\right]$.

In linear cryptanalysis, we are looking for a linear relation holding with probability as far as possible from 1/2, i.e. with the highest possible $|\epsilon|$. To do this we usually don't work either on the probability or the bias but on the correlation, which we define below.

Definition 17 (Correlation). *Let $f : \mathbb{F}_2^{n_1} \mapsto \mathbb{F}_{2^{n_2}}$ be a vectorial Boolean function. Assume that the masks for input x and output $f(x)$ are Γ_{in} and Γ_{out}. The correlation of the linear approximation is defined as*

$$\mathsf{Corr}(\Gamma_{in},\Gamma_{out}) = P(X=0) - P(X=1) =$$

$$= \frac{\#\{x \in \mathbb{F}_2^{n_1} | \Gamma_{in} \cdot x \oplus \Gamma_{out} \cdot f(x) = 0\} - \#\{x \in \mathbb{F}_2^{n_1} | \Gamma_{in} \cdot x \oplus \Gamma_{out} \cdot f(x) = 1\}}{2^{n_1}}.$$

It is trivial to see that

$$\epsilon = \frac{\mathsf{Corr}(\Gamma_{in},\Gamma_{out})}{2}.$$

As for the differential case, when dealing with a vectorial boolean function we can build a table containing all the correlations.

Definition 18 (LAT). *A* linear approximation table *(LAT) of a vectorial BF is a table that shows the correlation of every possible couple of masks applied to the input/output couples of the function.*

As for the DDT, for functions with small input sizes, for instance an S-Boxox, the LAT can be efficiently generated. Indeed, we have again that a $n_1 \times n_2$ Boolean function can be represented as a $2^{n_1} \times 2^{n_2}$ LAT.

Let $f = f_{r-1} \circ \ldots \circ f_1 \circ f_0$ be an iterated function. We define a linear trail (or characteristic) as below.

Definition 19 (Linear characteristic). *For an iterative function $f = f_{r-1} \circ \ldots \circ f_1 \circ f_0$, a sequence of linear approximations*

$$\Gamma_0 \xrightarrow{f_0} \Gamma_1 \xrightarrow{f_1} \ldots \Gamma_{r-1} \xrightarrow{f_{r-1}} \Gamma_r$$

is called an r-round linear trail *(or* characteristic*) of f.*

In order to estimate the correlation of a linear trail, meaning $\mathsf{Corr}(\Gamma_0,\Gamma_r)$ for the full function f, one has to use the Piling Up Lemma.

Lemma 2.1.1 (Piling-Up Lemma). *Let $(X_1,\ldots,X_n)$ be independent, binary-valued random variables with $p_1,\ldots,p_n$ and $\epsilon_1,\ldots,\epsilon_n$ defined as before. Then we have that*

$$P(X_1 \oplus X_2 \oplus \cdots \oplus X_n = 0) = 1/2 + 2^{n-1} \prod_{i=1}^{n} \epsilon_i$$

or

$$\epsilon = 2^{n-1} \prod_{i=1}^{n} \epsilon_i,$$

where ϵ is the total bias of the sum of the variables.

Applying the lemma to the correlation of the iterative function f yields

$$\mathsf{Corr}(\Gamma_{in}, \Gamma_{out}) = \prod_{0}^{r-1} \mathsf{Corr}(\Gamma_i, \Gamma_{i+1}).$$

As with the bias, we want to maximize the correlation of the trail. Therefore, we have to avoid any 0 correlation because if any appears then the probability of the full linear trail is $1/2$. As in the differential case, we often refer to the weight of the correlation, that is its base 2 logarithm.

Definition 20 (Linear hull [71])**.** *A* linear hull *(or* linear approximation*)* $(\Gamma_{in} \mapsto \Gamma_{out})$ *over* $f = f_{r-1} \circ \ldots \circ f_1 \circ f_0$ *contains all linear trails with* $(\Gamma_0 = \Gamma_{in})$*, and* $(\Gamma_r = \Gamma_{out})$*,*

Lemma 2.1.2. *The correlation of a linear hull is the sum of the correlations of all the linear trails as predicted by the correlation matrix [22].*

2.2 Linear analysis of main operations

Similarly to the differential case, the correlation of a linear trail can be computed by concatenating all input/output masks obtained for every single operation multiplying the respective correlations by Lemma 2.1.1. Even in this case, assumptions on the independence of the operations have to be made.

Although a two-step strategy is possible also in the linear case, it will not be investigated in the dissertation.

In this section, we will provide a detailed description of how linear cryptanalysis deals with the most common operations in a cryptographic primitive.

At first, we have to do a preliminary observation on what happens when there is a **branching**, i.e. when an input of the cipher or the output of a certain intermediate function is the input to more than one component of the cipher. In the differential case, nothing happens: the difference applied on the word that branches simply runs on all the new branches.

Proposition 2.2.1. *Let f_1 and f_2 be two functions such that $f_1 : \mathbb{F}_2^n \mapsto \mathbb{F}_2^{n_1}$ and $f_2 : \mathbb{F}_2^n \mapsto \mathbb{F}_2^{n_2}$. Let's consider the function*

$$\begin{aligned} f : \mathbb{F}_2^n &\mapsto \mathbb{F}_2^{n_1} \times \mathbb{F}_2^{n_2} \\ x &\mapsto (f_1(x), f_2(x)), \end{aligned}$$

then

$$\mathsf{Corr}_f(\Gamma_{in}^f, \Gamma_{out}^f) = \mathsf{Corr}_{f_1}(\Gamma_{in}^{f_1}, \Gamma_{out}^{f_1}) \cdot \mathsf{Corr}_{f_2}(\Gamma_{in}^{f_2}, \Gamma_{out}^{f_2}),$$

for every $(\Gamma_{in}^{f_1}, \Gamma_{in}^{f_2})$ and $(\Gamma_{out}^{f_1}, \Gamma_{out}^{f_2})$ such that $\Gamma_{in}^f = \Gamma_{in}^{f_1} \oplus \Gamma_{in}^{f_2}$ and $\Gamma_{out}^f = (\Gamma_{out}^{f_1}, \Gamma_{out}^{f_2})$.

Proof. It is enough to observe that

$$\begin{aligned} \Gamma_{in}^f \cdot x \oplus \Gamma_{out}^f \cdot f(x) &= (\Gamma_{in}^{f_1} \oplus \Gamma_{in}^{f_2}) \cdot x \oplus (\Gamma_{out}^{f_1}, \Gamma_{out}^{f_2}) \cdot (f_1(x), f_2(x)) \\ &= (\Gamma_{in}^{f_1} \cdot x \oplus \Gamma_{out}^{f_1} \cdot f_1(x)) \oplus (\Gamma_{in}^{f_2} \cdot x \oplus \Gamma_{out}^{f_2} \cdot f_2(x)). \end{aligned}$$

Then, we conclude by using the definition 17 and the Piling Up Lemma. □

By proposition 2.2.1, one has that if a certain word O becomes the input of two different components, the inputs masks to the component should be Γ_{I_1} and Γ_{I_2} with $\Gamma_O = \Gamma_{I_1} \oplus \Gamma_{I_2}$.

Then, we need to do a second remark regarding the **sign** of the correlation. Observe that some operations in the cipher may change the constant value 0 to 1 on the right side of equation 2.1. When this happens, one can keep the constant as it is and just change the sign of the correlation (or bias). However, we recall that we want to maximize the absolute value of the correlation and we can actually ignore the sign while searching for the best linear trail.

Let's see now how to work with the other possible operations:

- **XOR**: if $x_1 \oplus x_2 = y$, then $\Gamma_{x_1} = \Gamma_{x_2} = \Gamma_{out} = \Gamma$. Indeed,

  $$\Gamma_{x_1} \cdot x_1 \oplus \Gamma_{x_2} \cdot x_2 = \Gamma_{out} \cdot (x_1 \oplus x_2),$$

  in this case holds with probability 1, that is $\mathsf{Corr}(\Gamma \parallel \Gamma, \Gamma) = \frac{1}{2}$, whereas it is easy to observe that any other combination yields correlation 0. If one of the inputs is a constant c we do the same and the sign of the correlation can be calculated by $\Gamma_c \cdot c$. The technique is easily generalized to the multiple inputs case;

- **Linear layer/Mix column**: as already stated linear layer is a vector per matrix multiplication $x \cdot M$. Thus

  $$\begin{aligned} \Gamma_{in} \cdot x = \Gamma_{out} \cdot x \cdot M &\Leftrightarrow x \cdot \Gamma_{in} = x \cdot M \cdot \Gamma_{out} \\ &\Leftrightarrow \Gamma_{in} = M \cdot \Gamma_{out}. \end{aligned}$$

- **Rotation**: it is enough in this case to apply the rotation to the input mask, indeed:

  $$\Gamma_{in} \cdot x = R(\Gamma_{out}) \cdot R(x)$$

with probability 1, where R is the rotate function;

- **Shift**: for a shift of amount s to the left applied to a n-bit word, the input/output mask couple is determined in this way: $\Gamma_{in}[i] = 0$ for $0 \leq i \leq s-1$, i.e. the positions that will "shift away" are not involved in the linear relation. Then,

$$\Gamma_{out}[i] = \begin{cases} \Gamma_{in}[i+s] & \text{for } 0 \leq i \leq n-s-1 \\ 0/1 & \text{for } n-s-1 \leq i \leq n \end{cases}.$$

  We have:

$$\Gamma_{in} \cdot x = \Gamma_{out} \cdot S(x)$$

  with probability 1, where S is the rotate function, whereas for any other choice of the masks the correlation is 0;

- **NOT**: we can again consider the not as the xor with a constant string of all 1s. Then, the mask on the output is the same as the one of the input and the sign of the correlation is determined by the parity of Γ_{in};

- **AND/OR/Sboxes**: since they are operators with small input and output bit size (and/or operate bitwise), it is enough to compute the LAT;

- **Modular addition**: as in the differential case, because of the excessive big size, we can't use the LAT for the modular addition and we need a clever algorithm. This is provided by Nyberg and Wallén in [90, 72] and we will use the version presented in [36].

2.3 Partial key recovery linear attack

Once a linear hull is found the attacker can mount a known-plaintext partial key recovery attack to the cipher. We will briefly present algorithm 2 presented in [67].

First of all, note that in this case the attack is not chosen-plaintext but "just" known-plaintext. Indeed, whereas in the differential attack we needed precise plaintexts obtained by applying the input difference to a random plaintext, in the linear case we can accept any plaintext/ciphertext couple.

A second observation should be done on the linear equation 2.1 we exploit in the attack: the key from the bits, which are fixed but unknown, can be removed from the relation. In fact, they will only affect the constant value on the right side of the equation and so the sign of the correlation.

Let f_i be the function for i rounds and (Γ_0, Γ_{r-1}) a linear hull for $r-1$ rounds holding with a suitably large enough correlation.

Using the notations of section 1.4, the target subkey bits are the ones of the subkey K_r that are used to partially decrypt the last round of the cipher, i.e. to decrypt only the parts of the ciphertext involved in the linear relation.

For all possible combinations of values assigned to the target partial subkey one proceeds as follows:

1. Set a counter $c = 0$;
2. Take a (P, C) couple with C being the ciphertext after r rounds of the encryption function;
3. Partially decrypt the last round of the function f to obtain some bits of (C_{r-1}, C'_{r-1}), that are the ciphertexts after $r-1$ rounds. Compute the linear relation substituting the bits of the plaintext and the ciphertext found: if it is satisfied increase the counter by 1;
4. If there are other couples (P, C) available go to 2) otherwise return $|c - n/2|$, with n the number of plaintext/ciphertext samples used.

The combination of values for which the absolute value calculated in the last step is higher is assumed to be the correct one. Indeed, if the real subkey bits are used then the partial decryption is correct and it is true that $C_{r-1} = f_{r-1}(K, P)$ and $C'_{r-1} = f_{r-1}(K, P')$ whereas with wrong values of the subkeys C_{r-1} and C'_{r-1} should have arbitrary values. We expect the linear relation for these arbitrary ciphertexts to be satisfied about half of the times whereas to be satisfied (or not) significantly more frequently than half the times for the correct ones if n is high enough.

As for the differential attack, the complexity is estimated in terms of the number of samples used. Matsui estimated the number of plaintexts we expect to be needed is

$$n \approx \frac{1}{\epsilon^2}$$

where ϵ is the bias for the linear hull and in practice we expect that taking some small multiple of ϵ^{-2} is enough.

Chapter 3

Automatic tool for differential and linear cryptanalysis

In this chapter, I will present another important slice of my research work which involved the implementation of an automatic tool that generates models for differential and linear trail search.

The task of finding the best differential or linear characteristics is actually a Constraint Satisfaction Problem (CSP), or a Constraint Optimisation Problem.

CSP and COP ask to find a satisfying *assignment* for a set of *variables* with associated *domains* under a set of *constraints*. In a CSP, the implementer wants to find any solution, in our case, a trail with any probability or correlation. In a COP, he wants to obtain the optimal solution under a given *objective function*, for instance the differential trail with the highest probability.

These problems are usually entrusted to one of the following four dedicated solvers:

- **SAT**: in SAT solvers (see section 2.3.1), the variables can take only Boolean values and the constraints are in the form of clauses. SAT is usually a really efficient way for trail search but the most difficult way to implement it;
- **SMT**: SMT is an extension to SAT, which allows for more variable types, and additional constraints;
- **MILP**: in MILP (Mixed Integer Linear Programming), the variable domains are the integer, real or binary numbers while the constraints are expressed as linear inequalities;
- **CP**: in CP (Constraint Programming), the variables can be on any domain, and all types of constraints (including nonlinear) are permitted. Constraint programming is very common in linear and differential cryptanalysis since the search can be easily implemented.

The automatic tool, implemented in the software Python, takes as an input the graph representation of a cipher and outputs a constraint programming model written in the Minizinc language. The graph representation of a cipher is a Python dictionary from which one can recover all the specifics of the cipher and was developed by the Technology and Innovation Institute, a company I collaborated with during my Ph.D. To create a model, the

Python code is run in the free open-source mathematics software SageMath accessible via Python-based language and useful for its many mathematics functions.

Thus, in the next section, we will show what CSP and COP are formally and present constraint programming in an extended way.

3.1 Constraint Programming (CP)

Constraint Programming (CP) is a powerful technique for solving combinatorial search problems presented in the form of arbitrary constraints.

Definition 21. *A Constraint Satisfaction Problem (CSP) is defined by a tuple* (X, C, D) *where:*

- $X = (x_1, \ldots, x_n)$ *is the set of* ***variables*** *involved in the problem.;*
- $D = (D_1, \ldots, D_n)$ *is the set of* ***domains*** *of the variables. For every* $x_i \in X$ *we must have that* $X_i \in D_i$*;*
- $C = (C_1, \ldots, C_m)$ *is the set of constraints. A* ***constraint*** *is defined by a couple* (R_i, X_i) *where* R_i *is a relation and* X_i *is the subset of X containing the variables that must satisfy it, say* $X = \{x_{j_1}, \ldots, x_{j_k}\}$*. We recall that, by the definition of relation,* R_i *is a subset of the Cartesian product* $D_{j_1}, \ldots, D_{j_k}$ *which specifies the allowed combinations of values for the variables in X;*

A Constraint Optimization Problem (COP) adds an objective function to the set of variables, constraints, and domains, for instance, to minimize a certain variable of the model.

In CP there is no strict restriction on the domains apart from the ones for the specific problem.

Definition 22. *An* ***assignment*** *is a couple* (V, A) *where* $V = (x_{i_1}, \ldots, x_{i_s}) \subset X$ *is the set of selected variables and* $A = (a_{i_1}, \ldots, a_{i_s})$ *are the values assigned to each of the variables with* $a_{i_j} \in D_{i_j}$ *for every* i_j*. An assignment is* ***total*** *if* $X = V$*,* ***partial*** *otherwise.*

Definition 23. *An assignment satisfies a constraint* $C_i = (R_i, X_i)$ *if the assigned values satisfy the relation* R_i*.*

Definition 24. *A solution to a CSP problem is a total assignment that satisfies all the constraints.*

The strength of CP lies in the fact that the structure of the constraints could be very different so that many algorithms can be used at the same time to solve a problem. In solving a real-world problem, the search for a good model is crucial. The same problem can be hard to solve if represented with a poorly chosen model while easy if the model is efficient.

In fact, many times CP is used to solve Constraint Optimization Problems (COP) instead of CSP. In this case, the constraints could be more relaxed,

leaving many solutions available and the solver is asked to find only a few of them according to some preferences (e.g. a variable needs to be minimized).

Constraint programming is closely related to artificial intelligence: many ideas from AI regarding knowledge representation and reasoning can be applied to CP and at the same time CP techniques can give a contribution to the development of AI. This is probably because AI problems can be often seen as a form of searching in an enormous graph and also CSPs or COPs ask to look for solutions in a big search space.

3.1.1 Solving algorithms

The main resolution methods for CP rely on constraint propagation and backtracking.

In CP the problems are conjunctions of sub-problems and the first step, called **filtering**, runs an adapt algorithm for each of them to eliminate impossible values from the domains of the variables. Practically, for every constraint, the associated resolution method is used to prune the search space.

When a certain domain is restricted, a new domain restriction may be possible if looking back at an already considered constraint. Thus, every time the domain of a variable is restricted, all the constraints involving that variable need to be reconsidered and the respective filtering algorithm applied again. This step is called the **constraint propagation**.

These mechanisms are applied until arc consistency is reached for every constraint.

Definition 25 (Support)**.** *A certain value $a \in D_j$ is consistent with a constraint C_i, where $x_j \in X_i$, if $\exists \tau \in R_i$ such that $\tau[x_j] = a$, namely if setting $x_j = a$ one can find an assignment for the other variables involved in C_i so that R_i is satisfied. In this case, we say that τ is a support of a in C_i.*

Definition 26 (Arc Consistency)**.** *A constraint C_i is Arc Consistent (AC) iff $\forall x_j \in X_i$, $\forall a \in D_j$, there exist a support for a.*

In practice, this condition occurs when no more pruning can be done so that the solver needs to make a guess on a variable.

Hence, in the last step, a value is assigned successively to each variable. Whenever this is done, filtering and propagation are triggered. An assignment of a variable may lead to a dead end: all the elements from the domain of a variable that is not instantiated are removed. This means the guess was incorrect, so the algorithm backtracks, and a new assignment is tried. This last step is called **search**.

During the years, this general idea was carefully studied to try to improve the different mechanisms implied in it. Many techniques exist for the search, the backtrack, and the filtering algorithms. More about this can be found in [77].

3.1.2 Global constraints

One of the most important classes of constraints is the global ones. A global constraint is a relation between a set of variables encapsulating more than one constraint. These kinds of constraints are really powerful because their filtering algorithms work more efficiently than splitting the problem back to the subconstraints and solving each of them separately. The classical and most widespread example is the *all-different* constraint. Applied to a set of variables, it requests that all the values assigned to those variables must be pairwise different. An *all-different* constraint could be split in many binary constraints. However, doing that would not only slow down the resolution but also leave some impossible values in the domains of the variables. In [76], a detailed survey on the most frequently used constraints and filtering algorithms is provided.

3.1.3 Practical example

In the following, we will give a practical example of how CP works showing a Minizinc code to solve the famous puzzle called "Sudoku". In classical Sudoku one has to fill a 9x9 grid with digits from 1 to 9 with the following rules: each row, column and each of the 3x3 subgrids which compose it must contain all the nine digits. It is easy to use constraint programming to solve a Sudoku. We report here the code written in Minizinc IDE:

```
include "alldifferent.mzn";
set of int: RANGE = 1..9;
% The 9x9 Grid
array[RANGE, RANGE] of var RANGE: cell;

% Constraint fixing the known values
constraint cell[2,5]= 6 /\ cell[2,8]= 2 ... /\ cell
   [8,8]= 1;
% Constraint for having all different digits in every
   row
constraint forall(i in RANGE)(alldifferent([cell[i,j]|j
    in RANGE]));
% Constraint for having all different digits in every
   column
constraint forall(j in RANGE)(alldifferent([cell[i,j]|i
    in RANGE]));
% Constraint for having all different digits in every 3
   x3 subgrid
constraint forall(a in 1..3,b in 1..3)(alldifferent([
   cell[i+(a-1)*3,j+(b-1)*3]|i in 1..3, j in 1..3]));

output [show(row(cell,i)) ++ "\n"| i in RANGE];
```

Let us consider the scheme in 3.1.

Here the black digits are the ones of the original scheme, while the blue ones are obtained by the solver with constraint propagation. We will denote

							6	
				6			2	
6								
5	7	2	1	4	6	9	8	3
							5	
							7	
			7				9	
			5				1	
			6					

FIGURE 3.1: Sudoku example

by $S(i,j)$ the entrance of the scheme at row i and column j, with $1 \leq i,j \leq 9$. The problem can be modeled as a CSP, by definition 21, in the following way:

- $X = (x_1,\ldots,x_{81})$, where $x_{9(i-1)+j} = S(i,j)$ for every i,j;
- $D = (D_1,\ldots,D_{81})$, where $D_i = \{1,\ldots,9\}$ for every i;
- $C = (C_1,\ldots,C_{44})$, where $C_1 - C_{17}$ are for the cells with fixed digits, $C_{18} - C_{26}$ are the all-different constraints for the rows, $C_{27} - C_{35}$ for the columns and $C_{36} - C_{44}$ for the subgrids;

At first, the filtering algorithm for the *all-different* constraint lets the solver conclude that $S(4,6)$ is a 6. Then, constraint propagation is applied since one of the variables has been assigned a value. Through that $S(9,4)$ and then $S(1,8)$ are uniquely determined. In this case, we know that $A = (6,6,6)$ is a partial assignment that satisfies all the constraints.

The all-different constraint plays a fundamental role: let's see that in column 8 understanding how the algorithm would operate step by step.

1. By the all different constraint for column 8, the domain of $S(1,8)$ is restricted from $\{1,\ldots,9\}$ to $\{3,4,6\}$;
2. Similarly, by the all different constraint for column 8 and the all different constraint for row 3 the domain of $S(3,8)$ is restricted to $\{3,4\}$. At this moment filtering algorithm is again applied for the all different constraint of row 8 because a variable involved changed its domain but there is no effect on the domain of $S(1,8)$;
3. Samely, by the all different constraint for column 8 and the all different constraint for row 9 the domain of $S(9,8)$ is restricted to $\{3,4\}$. Again, the filtering algorithm for the all-different constraint of column 8 is applied and the domain of $S(1,8)$ is restricted to $\{6\}$.

Note that in the third step it is possible to determine that $S(1,8) = 6$ only because the constraint is a global constraint and considers the three variables' domains all at once. If one works only with binary constraints the arc consistency is reached when the domain of $S(1,8)$ is still $\{3,4,6\}$.

Constraint propagation is again applied and the domains restricted: for every cell, the solver calculates the digits available. At this point, none of the

FIGURE 3.2: MiniZinc Challenge 2020.

Category	Gold	Silver	Bronze
Fixed	SICStus Prolog	JaCoP	Choco 4
Free	OR-tools	PicatSAT	Mistral 2.0
Parallel	OR-tools	PicatSAT	Mistral 2.0
Open	OR-tools	Sunny-cp	PicatSAT
Local Search	Yuck	OscaR/CBLS	

domains is restricted to only one value so that the solver has to guess a digit for a cell (supposedly from the ones whose domain is smaller). This is what we have called the search. After the guess, constraints are again propagated, and search and propagations are alternated until a solution is found. Note that in this case the problem has many solutions. If not restricted, the solver will try to find all of them.

To model and solve a CP problem we have at our disposal different languages and solvers. We will use MiniZinc, a free and open-source constraint modeling language. Through MiniZinc it is possible to implement COPs and CSPs in a high-level solver-independent way and then compile the model into FlatZinc, a language that many solvers can understand.

The most widely used solvers are Chuffed, OR-Tools and Choco. In Figure 3.2 we can see the rank of solvers in a MiniZinc Challenge from 2020 organized for various categories:

- Fixed search: solvers must use the search strategy defined in the problem;
- Free search: solvers can use any search strategy;
- Parallel search: solvers are allowed to use parallelization, so multiple threads or cores to solve the problem;
- Open class: portfolio solvers are allowed too. Portfolio solver are essentially collection of solvers that, when facing a new unseen problem p, try to predict the best solver(s) to solve p;
- Local search: specific for local search solvers. This kind of search starts from a "local" solution and tries to improve it to get a possibly better solution in a "neighbourhood", that is a solution that is not too much different from the starting one.

3.2 CP constraints for the main operators in differential search

In this section, we will provide the code for implementing in Minizinc the constraints for differential trail search for the main operators adopted in cryptographic primitives. In section 1.2 one can find a theoretical explanation of the constraints. Note that this is a step behind the implementation of the tool in Python.

A common strategy applied by the tool to any kind of constraint is to create a Python list containing all the names of the variables representing the input bits and then generate the constraints for the operations bitwise in Minizinc avoiding declaring the array of all inputs. This is done because it is common to have inputs formed by more than one word (e.g. a concatenation of half of the plaintext and of the key is rotated). For almost every component we will also show a "bit version" of the implementation.

Constraints for the **NOT** function will not be shown since, as already explained, in differential trail search one has simply to constraint equality between input and output difference.

The input will be always denoted with the letter I ($I_1,\ldots,I_n$ if more than one) the output with O ($O_1,\ldots,O_n$ if more than one).

3.2.1 XOR

```
include "globals.mzn";

int : n;
array [0..n-1] of var 0..1: I_1;
array [0..n-1] of var 0..1: I_2;
array [0..n-1] of var 0..1: O;

constraint O = Xor_2(I_1, I_2);

% XOR of 2 arrays
function array[int] of var 0..1: Xor_2(array[int] of
    var 0..1: a, array[int] of var 0..1: b)=
array1d(0..(length(a)-1), [(a[j]+b[j]) mod 2  | j in
    0..(length(a)-1)]);
```

The automatic code for two addends XOR with four 8-bit inputs would be the following.

```
include "globals.mzn";

int : n;
array [0..7] of var 0..1: I_1;
array [0..7] of var 0..1: I_2;
array [0..7] of var 0..1: I_3;
array [0..7] of var 0..1: I_4;
array [0..15] of var 0..1: O;
```

```
9
10 constraint O_[0] = (I_1[0] + I_3[0]) mod 2;
11 ...
12 constraint O_[7] = (I_1[7] + I_3[7]) mod 2;
13 constraint O_[8] = (I_2[0] + I_4[0]) mod 2;
14 ...
15 constraint O_[15] = (I_2[7] + I_4[7]) mod 2;
```

3.2.2 Rotation/Shift

In the following, we show the code for left rotate and shift (the right ones are really similar).

```
1 include "globals.mzn";
2
3 int : n;
4 array [0..n-1] of var 0..1: I_1;
5 array [0..n-1] of var 0..1: O_1;
6 array [0..n-1] of var 0..1: O_2;
7
8 constraint O_1 = LRot(I_1, m);
9 constraint O_2 = LShift(I_1, m);
10
11 % Left rotation of X by val positions
12 function array[int] of var 0..1: LRot(array[int] of var
       0..1: X, var int: val)=
13 array1d(0..(length(X)-1), [X[(j+val) mod length(X)] | j
       in 0..(length(X)-1)]);
14
15 % Left shift of X by val positions
16 function array[int] of var 0..2: LShift(array[int] of
     var 0..2: X, var int:val)=
17 array1d(0..(length(X)-1), [if j<length(X)-val then X[(j
     +val) mod length(X)] else 0 endif | j in 0..(length(
     X)-1)]);
```

As for the XOR, we use a bit version for the rotation or the shift in the automatic model. We show a left rotate and shift of 3 of two 8-bit concatenated inputs.

```
1 include "globals.mzn";
2
3 int : n;
4 array [0..7] of var 0..1: I_1;
5 array [0..7] of var 0..1: I_2;
6 array [0..15] of var 0..1: O_1;
7 array [0..15] of var 0..1: O_2;
8
9 % Rotate
10 constraint O_1[0] = I_1[3];
11 ...
12 constraint O_1[5] = I_2[0];
13 ...
```

```
constraint O_1[12] = I_2[7];
constraint O_1[13] = I_1[0];
constraint O_1[14] = I_1[1];
constraint O_1[15] = I_1[2];

% Shift
constraint O_2[0] = I_1[3];
...
constraint O_2[5] = I_2[0];
...
constraint O_2[12] = I_2[7];
constraint O_2[13] = 0;
constraint O_2[14] = 0;
constraint O_2[15] = 0;
```

At its actual status, the automatic model doesn't address the dependencies issues regarding the shift explained in section 1.2.

3.2.3 Linear Layer/Mix Column

```
include "globals.mzn";

int : n;
array [0..n-1] of var 0..1: I;
array [0..n-1,0..n-1] of 0..1: M;
array [0..n-1] of var 0..1: O;

constraint forall(i in 0..n-1)(O[i] = sum([I[j]*M[j,i]|
   j in 0..n-1]) mod 2);
```

In the automatic tool, we don't declare the matrix in Minizinc but we directly write the modulo 2 sums. Let M be the following 4×4 matrix:

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix},$$

then the code would be the following:

```
include "globals.mzn";

int : n;
array [0..n-1] of var 0..1: I;
array [0..n-1] of var 0..1: O;

constraint O[0] = (I[1] + I[3]) mod 2;
constraint O[1] = (I[0] + I[1] + I[2]) mod 2;
constraint O[2] = (I[0] + I[2]) mod 2;
constraint O[3] = (I[1] + I[2] + I[3]) mod 2;
```

3.2.4 Sboxes/AND/OR

For S-Boxxes/AND/OR one can simply use the DDT table and exploit the table constraint by Minizinc, a powerful constraint that constrains an array to take only some possible combination of values explicited in the rows of the table. In particular, a row of the table will contain the following three elements concatenated: an input difference, an output difference, and the weight of the probability for the input/output difference couple. The DDT can be easily generated in Sagemath using the function "*difference_distribution_table*()".

```
include "globals.mzn";

int : n;
array [0..n-1] of var 0..1: I;
array [0..n-1] of var 0..1: O;
var int: w;
% DDT: m is the number of possible combinations
array [0..m, 1..2*n+1] of int: DDT = array2d(0..m,1..2*
   n+1,[0,1,...,1,2,1,0, ...,0,3,0,1, ... ]);

constraint table(I ++ O ++ [w], DDT);

solve minimize(p);
```

The bitwise case used in the automatic tool is quite similar but one concatenates single integers instead of arrays inside the table constraint. The code is omitted.

3.2.5 Modadd

For simplicity, even in the automatic tool, the constraints for modular addition involve the preliminary step of declaring the two arrays representing the addends.

```
include "globals.mzn";

int : n;
array [0..n-1] of var 0..1: I_1;
array [0..n-1] of var 0..1: I_2;
array [0..n-1] of var 0..1: O;
var int: w;

array [0..n-1] of var 0..1: Shi_I_1 = LShift(I_1,1);
array [0..n-1] of var 0..1: Shi_I_2 = LShift(I_2,1);
array [0..n-1] of var 0..1: Shi_O = LShift(O,1);
array [0..n-1] of var 0..1: eq = Eq(Shi_I_1, Shi_I_2,
   Shi_O);

constraint forall(j in 0..n-1)(if eq[j] = 1 then (sum([
   I_1[j], I_2[j], O[j]]) mod 2) = Shi_I_2[j] else true
    endif) /\ w = n-sum(eq);
```

```
15
16 % Eq function
17 function array[int] of var 0..1: Eq(array[int] of var
       0..1: a, array[int] of var 0..1: b, array[int] of
       var 0..1: c)=
18 array1d(0..(length(a)-1), [all_equal([a[j],b[j],c[j]])
       | j in 0..length(a)-1]);
19
20 solve minimize(w);
```

with the LShift and Xor_2 already defined in the previous code.

The constraint is based on algorithm 1. The first if of the constraint is for the if at line 1 of the algorithm. In the case the first part of the end is satisfied (i.e. eq = 1 or True) we want to ensure the second part is not, otherwise the algorithm returns probability 0 which means the input/output difference couple can't be valid. Instead, if the first part is not satisfied we are already good.

Finally, the rest of the constraint simply calculates the weight as in line 4 of the algorithm: shifting the arguments of the eq function as we do in the implementation has the same effect as the AND with $mask(n-1)$ in the algorithm.

3.3 Constraints and automatic tool for two steps strategy

In this section, we explain how to work with different components in the two steps strategy for differential trail search. The dimension of the arrays n will always be the result of the original length of the word divided by the word size.

Rotate/Shift and S-Boxxes are simple:

- **Rotate/Shift**: are equal to one step case if one inserts the rotate/shift_amount in the code already divided by the word_size.
- **Sboxes/AND/OR**: they are just equalities.

3.3.1 XOR

For a two inputs XOR the code is the following:

```
1 include "globals.mzn";
2
3 int : n;
4 array [0..n-1] of var 0..1: I_1;
5 array [0..n-1] of var 0..1: I_2;
6 array [0..n-1] of var 0..1: O;
7 array [0..4,1..3] of 0..1: XOR_table = array2d
      (0..4,1..3,[0,0,0,0,1,1,1,0,1,1,1,0,1,1,1]);
8
```

```
constraint forall(i in 0..n-1)(table([I_1[i]] ++ [I_2[i
    ]] ++ [O[i]], XOR_table));
```

The automatic model trivially develops the "forall".

3.3.2 Mix column

```
include "globals.mzn";

int : n;
array [0..n-1] of var 0..1: I;
array [0..n-1] of var 0..1: O;
array [0..m,1..2*n] of 0..1: mix_column_table = array2d
    (0..m,1..2*n,[TABLE]);

constraint table(I++O, mix_column_table);
```

"TABLE" stands for the full mix column table that we can generate using the branch number already discussed. In the automatic model, the branch number and subsequently the mix column table are calculated by a Python function.

To obtain a trail with the two steps strategy, the automatic model performs the following steps:

1. Build and solve the Minizinc model for finding the minimum number of active S-Boxxes;
2. Build and solve the Minizinc model for finding all the possible solutions with the number of active S-Boxxes in step 1);
3. Build and solve a Minizinc model which finds a full trail constraining the values of the bits to match with the values representing the activity of the words from any of the solutions found in step 2);
4. If no solution is found, that is all the truncated trails of step 2) are impossible, go back to 2) increasing by 1 the number of active S-Boxxes.

3.4 CP constraints for the main operators in linear search

This section is similar to 3.2 but dedicated to the linear trail search. For the notations, one can still refer to the one used in 3.2. We omit some simple or already given code:

- **XOR**: is simply the implementation of some equalities;
- **Rotate**: works as in the differential case;
- **Linear layer/Mix column**: works as in the differential case but I and O have to be exchanged;

- **Sbox/AND/OR**: one can do exactly as in the differential case but using the LAT instead of the DDT. The LAT can be easily generated in Sagemath using the function "*linear_approximation_table*()".

3.4.1 Branching

Suppose a certain word I has to be used as input to two different operations. The code for branching is the following, where I_1, I_2 are the inputs to the operations.

```
include "globals.mzn";

int : n;
array [0..n-1] of var 0..1: I;
array [0..n-1] of var 0..1: I_1;
array [0..n-1] of var 0..1: I_2;

constraint forall(i in 0..n-1)(I[i] = I_1[i] + I_2[i]
   mod 2);
```

The bitwise model generated automatically is trivial and one can look at the XOR in the differential case.

3.4.2 Shift

In the following, we show the code for the left shift (the right one is really similar).

```
include "globals.mzn";

int : n;
int : shift_amount;
array [0..n-1] of var 0..1: I;
array [0..n-1] of var 0..1: O;

constraint forall(i in 0..shift_amount-1)(I[i] = 0);
constraint forall(i in 0..n-shift_amount-1)(O[i] = I[i+
   shift_amount]);
```

The automatic tool for a left shift of amount 3 will generate:

```
include "globals.mzn";

int : n;
array [0..n-1] of var 0..1: I;
array [0..n-1] of var 0..1: O;

constraint I[0] = 0;
constraint I[1] = 0;
constraint I[2] = 0;
constraint O[0] = I[3];
...
constraint O[12] = I[15];
```

3.4.3 Modadd

As in the differential case, even in the automatic tool, the constraints for modadd involve the preliminary step of declaring the two arrays representing the addends.

```
include "globals.mzn";

int : n;
array [0..n-1] of var 0..1: I_1;
array [0..n-1] of var 0..1: I_2;
array [0..n-1] of var 0..1: O;
var int: w;

constraint modadd_linear(I_1,I_2,O,w);

% XOR of 3 arrays
function array[int] of var 0..1: Xor_3(array[int] of
   var 0..1: a, array[int] of var 0..1: b, array[int]
   of var 0..1: c)=
array1d(0..(length(a)-1), [(a[j]+b[j]+c[j]) mod 2  | j
   in 0..(length(a)-1)]);

%Modular addition for xor linear
predicate modadd_linear(array[int] of var 0..1: a,
   array[int] of var 0..1: b, array[int] of var 0..1: c
   , var int:w) = (
let {
array [0..length(a)] of var 0..1: state,
array [0..length(a)-1] of var 0..1: prob,
array [0..length(a)-1] of var 0..1: X=Xor_3(a,b,c)
} in
state[0]=0 /\
forall (i in 0..length(a)-1)(
    if state[i]==0 then all_equal([a[i],b[i],c[i]])
   else true endif /\
    state[i+1]=((X[i]+state[i]) mod 2) /\
    if state[i]==1 then prob[i]=1 else prob[i]=0 endif)
    /\
w=sum(prob)
);

solve minimize(w);
```

3.5 Experimental result

In this section we present the results of some experimental results on the automatic tool for the search of both differential and linear trails. The initial goal of the experiments was also to show how the automatic tool performs with respect to ad hoc models constructed for the search. Unfortunately, it

was really difficult to find proper papers in the literature showing the time required for the search of the trails. Most of the papers, indeed, only show the weight of the trail, or, at most, the trail itself. In particular, only one was found and compared.

Even though, we don't have enough timings to compare, it is still interesting to show some results in order to get an idea of the performances of the software.

The experiments were run on a server equipped with:

- `CPU: Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz;`
- `Memory: 768GB 2933 MHz;`
- `OS: Ubuntu 18.04.5 LTS.`

In table 3.1, the tests for differential and linear trails search in a single-key scenario on three ciphers are shown: Present [15], Simon [7] and TinyJAMBU [93], all lightweight algorithms. The latter is a finalist in the NIST lightweight cryptography challenge. As already mentioned, only for the differential trails of the block cipher Present we have a reference to compare with.

The columns of the table show:

- The cipher and the parameters chosen for the attack;
- The solver used for solving the model;
- The building and solving times, that is the timings for building the CP model and the one to solve it respectively;
- The timings declared for the solving from a paper in the literature. There is no building time since the model is written ad hoc;
- A reference and the year of publication of the paper we are comparing the results with;
- The weight, that is intended of the probability for the differential and of the correlation for the linear;

As shown in the table, the results from the comparison with the only paper found are good: the software seems to perform better than the ad hoc model when using the solver 'Chuffed'.

For what concerns the solvers, Chuffed and Gecode, there is not one that is clearly better than the other.

TABLE 3.1: Experimental results for differential and linear trails search with the automatic tool

Cipher	# of rounds	Plaintext/key sizes	Solver	Building time (s)	Solving time (s)	Time (lit)	Ref	Year	Weight
Differential									
Present	3	64/80	Chuffed	0.016	0.090	0.2	[88]	2017	8
	4			0.011	2.728	11.4			10
	5			0.011	586.513	3404.5			12
Present	3	64/80	Gecode	0.009	0.092	-	-	-	8
	4			0.009	2.420				10
	5			0.011	397.237				12
Simon	6	32/64	Chuffed	0.001	17.749	-	-	-	12
	7			0.002	75.472				14
	8			0.002	741.311				18
Simon	6	32/64	Gecode	0.001	29.811	-	-	-	12
	7			0.002	136.825				14
	8			0.002	1215.01				18
TinyJAMBU	180	128/128	Chuffed	0.020	0.205	-	-	-	4
	210			0.02	75.472				5
	250			0.027	827.164				7
TinyJAMBU	180	128/128	Gecode	0.020	0.947	-	-	-	4
	210			0.023	11.605				5
	250			0.027	952.657				7
Linear									
Present	3	64/80	Chuffed	0.102	0.060	-	-	-	4
	4			0.085	2.253				6
	5			0.108	51.572				8
Present	3	64/80	Gecode	0.079	0.091	-	-	-	4
	4			0.084	2.907				6
	5			0.108	56.511				8
Simon	6	32/64	Chuffed	0.015	11.443	-	-	-	6
	7			0.021	62.708				7
	8			0.027	559.339				9
Simon	6	32/64	Gecode	0.016	10.731	-	-	-	6
	7			0.020	54.797				7
	8			0.027	662.754				9
TinyJAMBU	180	128/128	Chuffed	64.391	3.428	-	-	-	2
	210			94.303	5.574				2
	250			143.637	119.291				3
TinyJAMBU	180	128/128	Gecode	64.703	0.066	-	-	-	2
	210			93.586	0.080				2
	250			143.539	15.302				3

Part III

Conclusion

Chapter 1

Conclusion

In this dissertation, we provided a wide overview of the most common and modern cryptanalytic techniques used to attack symmetric cryptographic primitives.

In the first part, we presented algebraic cryptanalysis, which tries to attack the key by solving a polynomial system representing the function. In my research this kind of attack was explored in detail and attempted on many ciphers: for instance, several ciphers from NIST lightweight cryptography competition. Nonetheless, in the paper we showed only the most significant two attacks, that are on the cryptographic hash function SHA-1 and on the Bluetooth cipher E0.

In the first, a preimage attack is mounted on a reduced version of the compression function of the primitive, following two directions: leaving more than 160 bits of the message free we reached round 23 matching the state of the art results whereas leaving only 64 free bits round 31 was attacked successfully outperforming the state of the art.

The attack to E0 is a key-recovery attack based on a short keystream, hence fitting with Bluetooth specifications, performed with the guess and determine technique. In the choice of the variables to fix, it was crucial the discovery of a fourteen variables special set that makes the resolution of the polynomial system particularly easy for half of the assignments. The system is solved by means of Gröbner bases, SAT solvers, and BDDs, and the first technique seems to obtain the best results. The expected runtime of the attack is about 2^{79} seconds and to the best of our knowledge improves any previous attack based on a short keystream.

The second part of the thesis concerns the implementation of an automatic tool implemented in Python that generates Minizinc models for the search of the best differential and linear trails of cryptographic primitives. Minizinc is a free and open-source constraint programming language.

Differential and linear cryptanalysis are presented giving a detailed explanation of how to deal with every possible component of a cipher. This also comprehends, in the last chapter, the Minizinc implementation of the main operations. Moreover, in most cases, a slightly different Minizinc implementation is also provided, closer to the one obtained when running the automatic tool.

Finally, the experimental results for the differential and linear trail search are shown providing also a comparison with a single paper on the lightweight block cipher PRESENT.

1.1 Future directions

The research activity could be extended in various directions.

For what concerns algebraic cryptanalysis, it would be interesting to investigate the possible employment of artificial intelligence for selecting the best possible set of key variables to fix. Indeed, in my research, this choice was always a result of an analysis of the structure of the cipher made "by hand".

Secondly, the automatic tool can be improved:

- New techniques for making the two steps strategy more efficient could be implemented;
- New constraints, such as the ones for the "shift by variable amount" component from Raiden block cipher [74], could be added;
- Search for an automatic strategy for finding the best differential instead of the best differential characteristics;

Finally, one could extend the automatic tool to work also for other kinds of cryptanalysis, such as impossible differential or rotational cryptanalysis.

Bibliography

[1] Andrew V. Adinetz and Evgeny A. Grechnikov. "Building a Collision for 75-Round Reduced SHA-1 Using GPU Clusters". In: *Euro-Par 2012 Parallel Processing*. Ed. by Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis. Springer Berlin Heidelberg, 2012, pp. 933–944.

[2] Ralph Ankele and Stefan Kölbl. *Mind the Gap - A Closer Look at the Security of Block Ciphers against Differential Cryptanalysis*. Cryptology ePrint Archive, Paper 2018/689. 2018.

[3] Kazumaro Aoki and Yu Sasaki. "Meet-in-the-middle preimage attacks against reduced SHA-0 and SHA-1". In: *Annual International Cryptology Conference*. Springer. 2009, pp. 70–89.

[4] Frederik Armknecht and Gwenolé Ars. "Algebraic Attacks on Stream Ciphers with Gröbner Bases". In: *Gröbner Bases, Coding, and Cryptography*. Ed. by Massimiliano Sala, Shojiro Sakata, Teo Mora, Carlo Traverso, and Ludovic Perret. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 329–348.

[5] Frederik Armknecht and Matthias Krause. "Algebraic Attacks on Combiners with Memory". In: *Advances in Cryptology - CRYPTO 2003*. Ed. by Dan Boneh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 162–175.

[6] Seminararbeit Bartkewitz. "Building Hash Functions from Block Ciphers, Their Security and Implementation Properties". In: (Jan. 2009).

[7] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. *The SIMON and SPECK Families of Lightweight Block Ciphers*. Cryptology ePrint Archive, Paper 2013/404. 2013.

[8] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying hash functions for message authentication". In: *Annual international cryptology conference*. Springer. 1996, pp. 1–15.

[9] Emanuele Bellini, Alessandro De Piccoli, Rusydi Makarim, Sergio Polese, Lorenzo Riva, and Andrea Visconti. "New Records of Pre-image Search of Reduced SHA-1 Using SAT Solvers". In: *Proceedings of the Seventh International Conference on Mathematics and Computing*. Ed. by Debasis Giri, Kim-Kwang Raymond Choo, Saminathan Ponnusamy, Weizhi Meng, Sedat Akleylek, and Santi Prasad Maity. Singapore: Springer Singapore, 2022, pp. 141–151.

[10] Eli Biham and Rafi Chen. "Near-Collisions of SHA-0". In: *Advances in Cryptology – CRYPTO 2004*. Ed. by Matt Franklin. Vol. 3152. Springer Berlin Heidelberg, 2004, pp. 290–305.

[11] Eli Biham, Rafi Chen, Antoine Joux, Christophe Carribault Patrick and Lemuet, and William Jalby. "Collisions of SHA-0 and Reduced SHA-1". In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by Ronald Cramer. Vol. 3494. Springer Berlin Heidelberg, 2005, pp. 36–57.

[12] Eli Biham and Orr Dunkelman. "A framework for iterative hash functions-HAIFA". In: *IACR Cryptology ePrint Archive* 2007 (Jan. 2007), p. 278.

[13] Eli Biham and Adi Shamir. "Differential cryptanalysis of DES-like cryptosystems". In: *Journal of CRYPTOLOGY* 4.1 (1991), pp. 3–72.

[14] *Bluetooth Core Specification*. revision v5.2. Bluetooth. 2019. URL: https://www.bluetooth.com/specifications/bluetooth-core-specification.

[15] 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*. Ed. by Pascal Paillier and Ingrid Verbauwhede. Springer Berlin Heidelberg, 2007, pp. 450–466.

[16] Bruno Buchberger. "Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems". In: *Aequationes Mathematicae* 4 (Jan. 1970), pp. 374–383.

[17] Christophe De Cannière and Florian Mendel and Christian Rechberger. "Collisions for 70-Step SHA-1: On the Full Cost of Collision Search". In: *Selected Areas in Cryptography*. Ed. by Carlisle Adams and Ali Miri and Michael Wiener. Vol. 4876. Springer Berlin Heidelberg, 2007, pp. 56–73.

[18] Florent Chabaud and Antoine Joux. "Differential collisions in SHA-0". In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by Hugo Krawczyk. Vol. 1462. Springer Berlin Heidelberg, 1998, pp. 56–71.

[19] Dengguo Feng Chuan-Kun Wu. "Boolean Functions and Their Applications in Cryptography". In: (2016).

[20] Nicolas T. Courtois and Gregory V. Bard. "Algebraic Cryptanalysis of the Data Encryption Standard". In: *Cryptography and Coding*. Ed. by Steven D. Galbraith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 152–169.

[21] Nicolas T. Courtois and Willi Meier. "Algebraic Attacks on Stream Ciphers with Linear Feedback". In: *Advances in Cryptology — EUROCRYPT 2003*. Ed. by Eli Biham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 345–359.

[22] Joan Daemen, René Govaerts, and Joos Vandewalle. "Correlation matrices". In: *International Workshop on Fast Software Encryption*. Springer. 1994, pp. 275–285.

[23] Joan Daemen and Vincent Rijmen. "The Block Cipher Rijndael". In: *Smart Card Research and Applications*. Ed. by Jean-Jacques Quisquater and Bruce Schneier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 277–284. ISBN: 978-3-540-44534-0.

[24] Quynh Dang. *Recommendation for applications using approved hash algorithms*. US Department of Commerce, National Institute of Standards and Technology, 2008.

[25] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (July 1960), pp. 201–215.

[26] Christian De Cannière Christophe and Rechberger. "Finding SHA-1 Characteristics: General Results and Applications". In: *Advances in Cryptology - ASIACRYPT 2006*. Ed. by Kefei Lai Xuejia and Chen. Vol. 4284. Springer Berlin Heidelberg, 2006, pp. 1–20.

[27] Christophe De Cannière. "Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles". In: *Information Security*. Ed. by Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 171–186.

[28] Christophe De Cannière and Christian Rechberger. "Preimages for Reduced SHA-0 and SHA-1". In: *Advances in Cryptology – CRYPTO 2008*. Ed. by David Wagner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 179–202. ISBN: 978-3-540-85174-5.

[29] Tom van Dijk and Jaco Pol. "Sylvan: multi-core framework for decision diagrams". In: *International Journal on Software Tools for Technology Transfer* 19 (Nov. 2017). DOI: 10.1007/s10009-016-0433-2.

[30] Georgios Doukidis, Nikolaos Mylonopoulos, Nancy Pouloudi, and Jill Shepherd. "What is the Digital Era?" In: Jan. 2004, pp. 1–18. ISBN: 9781591401599. DOI: 10.4018/978-1-59140-158-2.ch001.

[31] Tobias Eibach, Enrico Pilz, and Gunnar Völkel. "Attacking Bivium Using SAT Solvers". In: *Theory and Applications of Satisfiability Testing – SAT 2008*. Ed. by Hans Kleine Büning and Xishun Zhao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 63–76.

[32] Jeremy Erickson, Jintai Ding, and Chris Christensen. "Algebraic cryptanalysis of SMS4: Gröbner basis attack and SAT attack compared". In: *International Conference on Information Security and Cryptology*. Springer. 2009, pp. 73–86.

[33] Thomas Espitau, Pierre-Alain Fouque, and Pierre Karpman. "Higher-Order Differential Meet-in-the-middle Preimage Attacks on SHA-1 and BLAKE". In: *Advances in Cryptology – CRYPTO 2015*. Ed. by Rosario Gennaro and Matthew Robshaw. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 683–701. ISBN: 978-3-662-47989-6.

[34] Jean-Charles Faugere. "A new efficient algorithm for computing Gröbner bases (F4)". In: *Journal of pure and applied algebra* 139.1-3 (1999), pp. 61–88.

[35] Scott Fluhrer and Stefan Lucks. "Analysis of the E0Encryption System". In: *Selected Areas in Cryptography*. Ed. by Serge Vaudenay and Amr M. Youssef. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 38–48.

[36] Kai Fu, Meiqin Wang, Yinghua Guo, Siwei Sun, and Lei Hu. "MILP-based automatic search algorithms for differential and linear trails for speck". In: *International Conference on Fast Software Encryption*. Springer. 2016, pp. 268–288. ISBN: 978-3-662-52992-8. DOI: 10.1007/978-3-662-52993-5_14.

[37] Patrick Gallagher and Acting Director. "Secure hash standard (shs)". In: *FIPS PUB* 180 (1995), p. 183.

[38] Vijay Ganesh. "Adaptive Restart and CEGAR-Based Solver for Inverting Cryptographic Hash Functions". In: *Verified Software. Theories, Tools, and Experiments: 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers*. Vol. 10712. Springer. 2017, p. 120.

[39] David Gérault, Pascal Lafourcade, Marine Minier, and Christine Solnon. "Revisiting AES related-key differential attacks with constraint programming". In: *Information Processing Letters* 139 (2018), pp. 24–29. ISSN: 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2018.07.001. URL: https://www.sciencedirect.com/science/article/pii/S002001901830139X.

[40] Sudhir R. Ghorpade. *A Note on Nullstellensatz over Finite Fields*. 2018.

[41] Jovan Dj. Golić, Vittorio Bagini, and Guglielmo Morgari. "Linear Cryptanalysis of Bluetooth Stream Cipher". In: *Advances in Cryptology — EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 238–255.

[42] Carla Gomes, Bart Selman, and Henry Kautz. "Boosting Combinatorial Search Through Randomization". In: *Proceedings of the National Conference on Artificial Intelligence* (Mar. 2003).

[43] Evgeny A Grechnikov. "Collisions for 72-step and 73-step SHA-1: Improvements in the Method of Characteristics." In: *IACR Cryptology ePrint Archive* 2010 (2010), p. 413.

[44] G. M. Greuel, G. Pfister, and H. Schönemann. "SINGULAR: A Computer Algebra System for Polynomial Computations". In: *ACM Commun. Comput. Algebra* 42.3 (2009), pp. 180–181.

[45] Gert-Martin Greuel and Pfister Gerhard. *A Singular Introduction to Commutative Algebra*. 1st. Heidelberg: Springer Berlin, 2002.

[46] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 212–219.

[47] Howard Heys. "A Tutorial on Linear and Differential Cryptanalysis". In: *Cryptologia* 26 (June 2001).

[48] Oleksandr Kazymyrov, Roman Oliynykov, and Håvard Raddum. "Influence of addition modulo 2 n on algebraic attacks". In: *Cryptography and Communications* 8.2 (2016), pp. 277–289. DOI: 10.1007/s12095-015-0136-7. URL: https://doi.org/10.1007/s12095-015-0136-7.

[49] John Kelsey and Bruce Schneier. "Second preimages on n-bit hash functions for much less than 2^n work". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2005, pp. 474–490.

[50] Andreas Klein. *Stream Ciphers*. Dec. 2013.

[51] Simon Knellwolf and Dmitry Khovratovich. "New Preimage Attacks against Reduced SHA-1". In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 367–383. ISBN: 978-3-642-32009-5.

[52] Matthias Krause. "BDD-Based Cryptanalysis of Keystream Generators". In: *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology*. EUROCRYPT '02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 222–237.

[53] Matthias Krause and Dirk Stegemann. "Reducing the Space Complexity of BDD-Based Attacks on Keystream Generators". In: *Fast Software Encryption*. Ed. by Matthew Robshaw. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 163–178.

[54] Harish Kumar Sahu, Indivar Gupta, N. Rajesh Pillai, and Rajendra Kumar Sharma. "BDD-based cryptanalysis of stream cipher: a prac- tical approach". English. In: *IET Information Security* 11 (3 May 2017), 159–167(8).

[55] Roberto La Scala, Sergio Polese, Sharwan K. Tiwari, and Andrea Visconti. *An algebraic attack to the Bluetooth stream cipher E0*. Cryptology ePrint Archive, Paper 2022/016. 2022.

[56] Florian Legendre, Gilles Dequen, and Michaël Krajecki. "Encoding hash functions as a sat problem". In: *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*. Vol. 1. IEEE. 2012, pp. 916–921.

[57] Florian Legendre, Gilles Dequen, and Michaël Krajecki. "Logical Reasoning to Detect Weaknesses About SHA-1 and MD4/5." In: *IACR Cryptol. ePrint Arch.* 2014 (2014), p. 239.

[58] Gaëtan Leurent and Thomas Peyrin. "From Collisions to Chosen-Prefix Collisions Application to Full SHA-1". In: *Advances in Cryptology – EUROCRYPT 2019*. Vol. 11478. Advances in Cryptology – EUROCRYPT 2019. Springer, 2019, pp. 527–555. DOI: 10.1007/978-3-030-17659-4_18.

[59] Jørn Lind-Nielsen. "BuDDy : A binary decision diagram package." In: 1999.

[60] Helger Lipmaa and Shiho Moriai. "Efficient Algorithms for Computing Differential Properties of Addition". In: *FSE 2001, Lecture Notes in Computer Science*. Vol. 2355. Springer. 2001, pp. 336–350.

[61] Yunwen Liu. "Techniques for Block Cipher Cryptanalysis". Available at: https://www.esat.kuleuven.be/cosic/publications/thesis-306.pdf. PhD thesis. KU Leuven, Faculty of Engineering Science, Sept. 2018.

[62] Philippe Loustaunau and William W. Adams. *An Introduction to Grobner Bases*. American Mathematical Soc., 1994.

[63] Michael Luby, Alistair Sinclair, and David Zuckerman. "Optimal speedup of Las Vegas algorithms". In: *Information Processing Letters* 47.4 (1993), pp. 173–180.

[64] Rusydi H Makarim and Marc Stevens. "M4GB: an efficient Gröbner-basis algorithm". In: *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*. 2017, pp. 293–300.

[65] J.P. Marques Silva and K.A. Sakallah. "GRASP-A new search algorithm for satisfiability". In: *Proceedings of International Conference on Computer Aided Design*. 1996, pp. 220–227.

[66] Joao P Marques-Silva and Karem A Sakallah. "GRASP: A search algorithm for propositional satisfiability". In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521.

[67] Mitsuru Matsui. "Linear Cryptanalysis Method for DES Cipher". In: *EUROCRYPT. Lecture Notes in Computer Science*. Vol. 765. Springer. 1993, pp. 386–397.

[68] Mitsuru Matsui and Atsuhiro Yamagishi. "A New Method for Known Plaintext Attack of FEAL Cipher". In: *EUROCRYPT. Lecture Notes in Computer Sciences*. Vol. 658. Springer. 1992, pp. 81–91.

[69] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. Discrete Mathematics and Its Applications. Taylor & Francis Inc, 1996.

[70] Vegard Nossum. "SAT-based preimage attacks on SHA-1". MA thesis. University of Oslo, 2012.

[71] Kaisa Nyberg. "Linear approximation of block ciphers". In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1994, pp. 439–444.

[72] Kaisa Nyberg and Johan Wallén. "Improved Linear Distinguishers for SNOW 2.0". In: *Fast Software Encryption*. Ed. by Matthew Robshaw. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 144–162.

[73] Sergio Polese and Andrea Visconti. "Survey: Attacks on hash functions". In: *DE CIFRIS cryptanalysis: selected papers from the itasec2020 workshop "Cryptanalysis, a key tool in securing and breaking ciphers"*. Ed. by aracne. Collectio Ciphrarum. Mar. 2022, p. 71.

[74] Javier Polimón, Julio Hernandez-Castro, Juan Tapiador, and Arturo Ribagorda. "Automated design of a lightweight block cipher with Genetic Programming". In: *KES Journal* 12 (Mar. 2008), pp. 3–14. DOI: 10.3233/KES-2008-12102.

[75] Havard Raddum. "Cryptanalytic results on TRIVIUM." In: eSTREAM, ECRYPT Stream Cipher Project, 2006, Report 2006/039.

[76] Jean-Charles Régin. "Global constraints: A survey". In: *Hybrid optimization*. Ed. by Pascal van Hentenryck and Michela Milano. New York, NY: Springer New York, 2011, pp. 63–134. ISBN: 978-1-4419-1644-0. DOI: 10.1007/978-1-4419-1644-0_3. URL: https://doi.org/10.1007/978-1-4419-1644-0_3.

[77] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. USA: Elsevier Science Inc., 2006. ISBN: 0444527265.

[78] Roberto La Scala and Sharwan K. Tiwari. "Stream/block ciphers, difference equations and algebraic attacks". In: *CoRR* abs/2003.14215 (2020).

[79] U. Schöning and J. Torán. *The Satisfiability Problem: Algorithms and Analyses*. Mathematik für Anwendungen. Lehmanns Media, 2013.

[80] Yaniv Shaked and Avishai Wool. "Cryptanalysis of the Bluetooth E0 Cipher Using OBDD's". In: *Information Security*. Ed. by Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 187–202.

[81] Claude Shannon. *A Mathematical Theory of Cryptography*. 1945.

[82] Mate Soos. *CryptoMiniSat 5. An advanced SAT solver*. 2021. URL: https://www.msoos.org/cryptominisat5.

[83] Marc Stevens. "Attacks on hash functions and applications". PhD thesis. Mathematical Institute, Faculty of Science, Leiden University, Jan. 2012.

[84] Marc Stevens. "New Collision Attacks on SHA-1 Based on Optimal Joint Local-Collision Analysis". In: *Advances in Cryptology - EUROCRYPT 2013*. Vol. 7881. Lecture Notes in Computer Science. Springer, 2013, pp. 245–261. DOI: 10.1007/978-3-642-38348-9_15. URL: https://www.iacr.org/archive/eurocrypt2013/78810243/78810243.pdf.

[85] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. "The First Collision for Full SHA-1". In: *CRYPTO*. Springer, 2017, pp. 570–596. DOI: 10.1007/978-3-319-63688-7_19.

[86] Douglas Robert Stinson and Maura Paterson. *Cryptography: theory and practice*. Textboook in Mathematics. Taylor & Francis Ltd, 2018.

[87] *Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process*. 2018. URL: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf.

[88] Siwei Sun, David Gerault, Pascal Lafourcade, Qianqian Yang, Yosuke Todo, Kexin Qiao, and Lei Hu. "Analysis of AES, SKINNY, and Others with Constraint Programming". In: *IACR Transactions on Symmetric Cryptology* 2017.1 (Mar. 2017), pp. 281–306.

[89] A. Visconti and F. Gorla. "Exploiting an HMAC-SHA-1 Optimization to Speed up PBKDF2". In: *IEEE Transactions on Dependable and Secure Computing* 17.4 (2020), pp. 775–781.

[90] Johan Wallén. "Linear Approximations of Addition Modulo 2n". In: *Fast Software Encryption*. Ed. by Thomas Johansson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 261–273.

[91] Toby Walsh. "Search in a Small World". In: *IJCAI International Joint Conference on Artificial Intelligence* 2 (Apr. 2002).

[92] Xiaoyun Wang and Yiqun Lisa Yin and Hongbo Yu. "Finding collisions in the full SHA-1". In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by Victor Shoup. Vol. 3621. Springer Berlin Heidelberg, 2005, pp. 17–36.

[93] Hongjun Wu and Tao Huang. "TinyJAMBU : A Family of Lightweight Authenticated Encryption Algorithms (Version 2)". In: 2019.