Mutant Algebraic Side-Channel Attack

Mutierte Algebraische Seitenkanalangriffe
Master-Thesis von Qi Zhang aus Darmstadt
Tag der Einreichung:

1. Gutachten: Prof. Dr. Johannes A. Buchmann
2. Gutachten: Dr. Mohamed Saied Emam Mohamed

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Institut CDC

Mutant Algebraic Side-Channel Attack
Mutierte Algebraische Seitenkanalangriffe

Vorgelegte Master-Thesis von Qi Zhang aus Darmstadt

1. Gutachten: Prof. Dr. Johannes A. Buchmann
2. Gutachten: Dr. Mohamed Saied Emam Mohamed

Tag der Einreichung:

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-12345
URL: http://tuprints.ulb.tu-darmstadt.de/1234

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
Namensnennung – Keine kommerzielle Nutzung – Keine Bearbeitung 2.0 Deutschland
http://creativecommons.org/licenses/by-nc-nd/2.0/de/

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den December 2, 2015

(J. Walker)

Abstract

Algebraic side-channel attacks (ASCA), combining side-channel attacks with algebraic techniques in a very effective manner, have been introduced as a potentially powerful cryptanalysis technique against block ciphers for years [34]. However, though the feasibility of ASCA has been successfully proven, yet its capability was not exploited to the greatest extent. In order to conquer this, one primary action is to reduce the huge size of the algebraic system constructed by ASCA.

In this master project, a more efficent algebraic side-channel attack named as *Mutant algebraic side-channel attack* (MASCA) has been proposed. Not only reduces MASCA the size of an algebraic system but also optimizes the system. The so-called "Mutants" indicate the short and simple clauses which are obtained through *exhaustive search* and optimization of the algebraic representation of the leaked side-channel information and can substitute the long clauses gained through standard representation of a Boolean function employed by Renauld et al. [27]. Subsequently, the mutants are inserted into the algebraic system of a cryptographic algorithm, which eventually brings a good influence on the performance, such as speeding up the process of solving SAT problems, increasing the success rate, etc. In this way, the optimization is the focus of this work and it is achieved through two filters proposed in this master project. So far, all side-channel information with which MASCA deals is correct Hamming weight leakages. However, MASCA is also able to handle incorrect Hamming weight leakages and such an ability is called *error tolerance*. In addition, the optimization works also well in the case of erroneous Hamming weight leakages.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

1 Introduction

For the last decades, considerable cryptograhic algorithms have been developed and proposed. In the course of implementing a cryptographic algorithm, e.g. a block cipher, not only is the security of this algorithm itself important, but also that the implementation of this algorithm doesn't unintentionally leak any information about the processed data plays a significant roll. The attacks which can utilize such leaked information to retrieve the secret key of a cryptographic primitive are called side-channel attacks. In a classical cryptanalysis which is aiming at block ciphers, adversaries are often able to acquire the input/ouput pairs of a target cryptographic algorithm. Side-channel attacks supply the adversaries with some additional information about the intermediate values leaked by a device on which the implementation of the cryptographic algorithm is carried out. Moreover, there are two common examples for leakage models, *Hamming distance* and *Hamming weight* models [8]. In this work, the *Hamming weight* model is adopted. This kind of attacks are more powerful and less general because of their individual peculiarity in terms of the implementation of the specific cryptographic algorithms [34].

The recently introduced *algebraic side-channel attack* (ASCA) associates side-channel attacks with algebraic cryptanalysis and gains benefits from both classical attacks to a great extent [8]. Combining the information acquired from a side-channel attack with an algebraic system which represents a cryptographic primitive assists attackers to effectively retrieve the secret key even though the number of traces in an attack phase is too low (e.g. only a single trace) for a statistical side-channel attack. In addition, due to the adaptability and descriptiveness of the algebraic representation, any processed intermediate values, e.g. the side-channel information, can be inserted into an algebraic system so that the process of retrieving the secret key can be accelerated and the accuracy is also accordingly improved, which makes ASCA a very powerful side-channel attack when an attacker is assumed to be profiling based [34, 33].

ASCA constructs a system of algebraic equations describing a cryptographic algorithm and side-channel information leaked by a device. Since the system is usually of a considerably great size, it may be inefficient and perhaps very hard to find solutions for the system. In this context, a variant of *algebraic side-channel attacks* (ASCA) is proposed in this master project, the *Mutant algebraic side-channel attack (MASCA)* which is more efficient. This variant focuses on the optimization of the representation of algebraic systems by simplifying the representation in two aspects, minimizing the number of clauses and shortening the length of clauses. Because both great length and great quantity of clauses hinder SAT solvers from finding a solution for the algebraic system in a reasonable time [7]. For the improvement in the length of clauses, a proper length ℓ ($\ell \in \mathbb{N}$ and $\ell \leq 4$) is determined in this work. All the clauses of length $\geq \ell_{max}$ where $\ell_{max} = 4$ are not considered. For the improvement in the number of clauses, two filters are proposed to remove the redundancy existing among the identified set of clauses and choose clauses more carefully. After the optimization, these resulting clauses are the so-called "*mutants*". Inserting theses mutants into the algebraic system may lead to higher efficiency and success rate of MASCA than ASCA.

The above work is based on the hypothesis that all the leaked Hamming weights are error-free. However, the erroneous Hamming weights cannot be avoided in real attacks and injecting the clauses describing such Hamming weights into the system may result in incorrect solutions or even the unsolvability of the system. Therefore, the capability of handling incorrect Hamming weights is considerably necessary for attackers. MASCA is designed to be able to deal with this problem and this ability is called *error tolerance*.

This master thesis is organized as follows. In section 2, some preliminary techniques are introduced, such as algebraic cryptanalysis, side-channel attacks, as well as block ciphers, in order to make this master project more clear and easier to understand. The key algorithms of MASCA are specified in Section 3. Section 4 presents and illustrates the comparisons of the experimental results of MASCA and other attacks to support the better performance of MASCA. Besides, Section 5 explains how MASCA copes with erroneous Hamming weights and shows the corresponding experimental results. Finally, Section 6 concludes this master project.

2 Preliminaries

In this section, some preliminaries as the cornerstones of this master project are introduced. They are helpful to better and more clearly understand the ideas proposed in this work.

2.1 Block Ciphers

A block cipher is a function which can encrypt plaintexts (n-bit blocks and the set of these blocks is denoted as $\mathcal{P}$) to ciphertexts (n-bit blocks and the set of these blocks is denoted as $\mathcal{C}$) where n is the *block length*. The encryption function must be invertible so that the unique decryption is allowed. Furthermore, a k-bit key K which is taken from the *key space* $\mathcal{K}$ is utilized to parameterize the function and is generally hypothesized to be selected randomly. Besides, the data expansion can be avoided because of the usage of plaintext- and ciphertext-blocks of the same size n. The encryption function is a bijection for n-bit plaintext $\mathcal{P}$ and ciphertext blocks $\mathcal{C}$ and a fixed key $\mathcal{K}$ [26].

2.1.1 Specification

A block cipehr is an invertible function which performs encipherment and decipherment on data blocks of fixed size. It can be denoted as a tuple $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ where $\mathcal{P}$ is the set of plaintext blocks, $\mathcal{C}$ is the set of ciphertext blocks, and $\mathcal{K}$ is the set of keys. In addition, $\mathcal{E}$ and $\mathcal{D}$ express the encryption and decryption functions, respectively. The two functions are in the following forms:

$$\mathcal{E}_K : \mathcal{P} \times \mathcal{K} \longrightarrow \mathcal{C}$$
$$\mathcal{D}_K : \mathcal{C} \times \mathcal{K} \longrightarrow \mathcal{P}$$

where $K \in \mathcal{K}$ [26] and $\mathcal{D}_K = \mathcal{E}_K^{-1}$.

2.1.2 Security and Attacks

A block cipher devote itself to supply the confidentiality, while an attacker dedicates itself to break the block cipher which is to recover the secret key to the best of its ability. If the secret key is retrieved, the block cipher is totally broken. By contrast, if the part of plaintext is recovered from ciphertext, the block cipher is partially broken. What needs to be noted are two assumptions for assess the security of block ciphers — (I) all data which are transmitted through the ciphertext channel are accessible to adversaries and (II) (*Kerckhoffs' assumption*) adversaries know all details of the encryption/decryption function other than the secret key. Based on these two hypotheses, attacks are categorized in terms of what kind of information is known to adversaries except for the intercepted ciphertext. Three prominent examples are [26]:

- *ciphertext-only* — only the intercepted ciphertext is known to adversaries.
- *known-plaintext* — except for the intercepted ciphertext, plaintext is also known to adversaries.
- *chosen-plaintext* — adversaries choose a plaintext by themselves, the corresponding ciphertext is available.

2.1.3 Iterated Block Ciphers

If a block cipher is susceptible to recover the secret key, then the security is increased by encrypting the same data block more than once. In other words, a more complex relation between plaintexts, ciphertexts, and a key is constructed by performing a simple transformation used to plaintexts iteratively. This model is called *multiple encryption*, a.k.a. *iterated block ciphers*. Meanwhile, the relationship between the encryption and decryption functions stays unchanged — $\mathcal{D} = \mathcal{E}^{-1}$. Two common examples of *multiple encryption* are *double* and *triple* encryption which are illustrated in the Figure 1 and Figure 2. In addition, *double encryption* is defined to be $c = \mathcal{E}(p) = \mathcal{E}_{K_2}(\mathcal{E}_{K_1}(p))$ and *triple encryption* to be $c = \mathcal{E}(p) = \mathcal{E}_{K_3}(\mathcal{E}_{K_2}(\mathcal{E}_{K_1}(p)))$ [26]. Except for these two cases, certainly there are some other cases with different number of rounds, such as AES, PRESENT, etc.

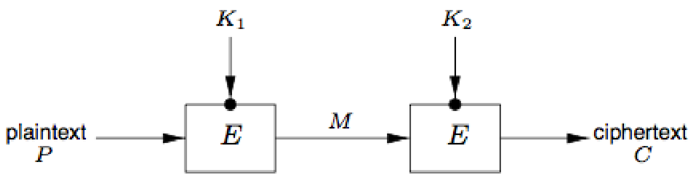

Figure 1: The scheme of double encrypton

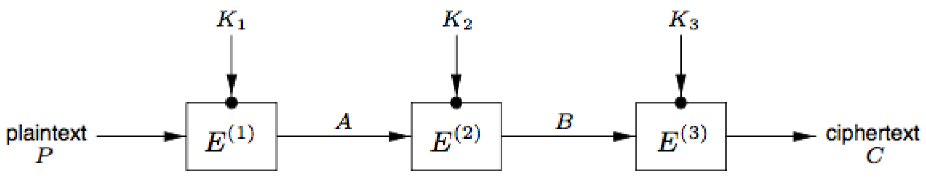

Figure 2: The scheme of triple encrypton

2.1.4 AES Algorithm

The Advanced Encryption Standard (AES) algorithm, which was first introduced in 1998, is a symmetric block cipher and can use different cipher keys which are the sequences of 128, 192 or 256 bits to encipher and decipher data blocks of a fixed length (128 bits) [29].

In order to describe the AES algorithm clearly, some definitions and notations are declared here. In AES algorithm, both the input and output blocks have the same fixed length — 128 bits. This length is denoted through $Nb = 4$, which means the data blocks consist of four 32-bit words. Analogously, the length of cipher keys is represented as $Nk = 4, 6$ or 8, which indicates Nk 32-bit words comprise the secret key. Furthermore, the number of rounds which is expressed as Nr has a mapping relationship with the key size or with the version of AES. According to the different values of Nk, Nr can be determined to be 10, 12 or 14. The detailed mapping relationship is shown in the Table 1. Besides, there still is a very important notion which is *state*. *State* describes the intermediate values during the course of encryption or decryption and can be signified as a rectangular byte-array with four rows and Nb columns [29].

	key length (Nk)	block size (Nb)	rounds (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Table 1: Key-Block-Round Comibinations

For both encryption and decryption, the AES algorithm applies a round function which is comprised of four distinct byte-oriented transformations which are shown as follows:

- substitution through a substitution table which is S-Box
- shifting rows of state
- mixing columns of state
- addition of a round key to state

The first transformation is a non-linear operation of the AES algorithm. It is performed independently on each byte of the state. Once an initial addition of round key is finished, the four transformations are performed in sequence for the first $Nr - 1$ rounds, while it is slightly different for the last round to not execute the transformation of mixing columns. These four transformations are accordingly denoted as *SubBytes()*, *ShiftRows()*, *MixColumns()*, and *AddRoundKey()*. In this way, the corresponding four reverse transformations are signified as *InvSubBytes()*, *InvShiftRows()*, *InvMixColumns()*, and *AddRoundKey()*. Then, the description of encryption is shown in the Algorithm 1 and the description of decryption in the Algorithm 2 [29].

Thanks to the requirements of high speed and low RAM, AES works well on various hardware. Therefore, AES is widely used in many different fields and organizations.

2.1.5 PRESENT Algorithm

Although AES is a prominent block cipher which is applied on a wide variety of areas and reduce the requirement of new block ciphers, it is still not applicable for some conditions. One of today's trends of IT landscape is to extensively develop tiny computer devices. Such developments may result in some particular security risks. However, not only AES but also the some other cryptographic primitives at hand

Algorithm 1 The encryption process of AES

```
1: procedure ENCIPHER($input[4*Nb]$, $output[4*Nb]$, $word[Nb*(Nr+1)]$)
2: begin
3:     byte $state[4*Nb]$
4:     $state \leftarrow input$
5:     $AddRoundKey(state, word[0, Nb-1])$
6:     for $round = 1$ to $(Nr-1)$ do
7:         $SubBytes(state)$
8:         $ShiftRows(state)$
9:         $MixColumns(state)$
10:        $AddRoundKey(state, word[round*Nb, (round+1)*Nb-1])$
11:    $SubBytes(state)$
12:    $ShiftRows(state)$
13:    $AddRoundKey(state, word[Nr*Nb, (Nr+1)*Nb-1])$
14:    $output \leftarrow state$
15: end
```

Algorithm 2 The decryption process of AES

```
1: procedure DECIPHER($input[4*Nb]$, $output[4*Nb]$, $word[Nb*(Nr+1)]$)
2: begin
3:     byte $state[4*Nb]$
4:     $state \leftarrow input$
5:     $AddRoundKey(state, word[Nr*Nb, (Nr+1)*Nb-1])$
6:     for $round = (Nr-1)$ step $-1$ downto 1 do
7:         $InvShiftRows(state)$
8:         $InvSubBytes(state)$
9:         $AddRoundKey(state, word[round*Nb, (round+1)*Nb-1])$
10:        $InvMixColumns(state)$
11:    $InvShiftRows(state)$
12:    $InvSubBytes(state)$
13:    $AddRoundKey(state, word[0, Nb-1])$
14:    $output \leftarrow state$
15: end
```

are not suitable for extremely resource-limited environments, for example, sensor networks, RFID tags, etc. In this context, PRESENT, an ultra-lightweight block cipher, is proposed in [5]. PRESENT takes both security and hardware efficiency into consideration so that it can avoid a compromise in security and realize a good performance in hardware at the same time.

PRESENT is an instance of SP-network (substitution-permutation network) and a hardware-optimized block cipher which may take keys ($\mathcal{K}$) of length of either 80 or 128 bits. In this way, two versions of PRESENT with individual keys of different lengths are derived and they are expressed as PRESENT-80 and PRESENT-128. Besides, both versions of PRESENT takes data blocks of 64 bits as input ($\mathcal{P}$) and generate new data blocks which are also 64 bits as output ($\mathcal{C}$). The whole algorithm of PRESENT is composed of 31 rounds. In contrast to AES, the number of rounds of PRESENT is independent with the key size, which means for both versions with secret key of either 80 or 128 bits, the number of rounds is fixed to be 31. For each round of the 31 rounds, a round function is employed. The function is comprised of three transformations which are displayed as follows:

- addition of round key
- substitution through a substitution table (S-Box)
- permutation

These transformations are carried out in sequence and denoted as *addRoundKey()*, *sBoxLayer()*, and *pLayer()*, respectively [5]. For the first transformation, the *key schedule*, generating a round key for each round, is denoted as a function *generateRoundKeys()* Furthermore, MASCA is applied to PRESENT-80 in this master project. Let K be the key register storing the user-provided key and K is represented to be a sequence $k_{79}k_{78}\ldots k_0$. In addition, let K_i describes the round key at the i-th round. K_i is expressed as a 64-bit sequence $\mathsf{k}_{63}\mathsf{k}_{62}\ldots\mathsf{k}_0$ where $1 \leq i \leq 32$. In this way, that the round key K_i is comprised of the 64 leftmost bits of the updated key register K at i-th round may lead to the following equation:

$$K_i = \mathsf{k}_{63}\mathsf{k}_{62}\ldots\mathsf{k}_0 = k_{79}k_{78}\ldots k_{16}.$$

When it is finished to generate the round key K_i, the current key register $K = k_{79}k_{78}\ldots k_0$ is updated through three steps:

- the key register shifts 61 bits to the left
- S-Box is employed to the four leftmost bits
- the five bits $k_{19}k_{18}k_{17}k_{16}k_{15}$ of the current key register K is exclusive-ored with the $round\_counter$ whose value is the round number i

These three steps are signified as follows:

$$[k_{79}k_{78}\dots k_1k_0] = [k_{18}k_{17}\dots k_{20}k_{19}]$$
$$[k_{79}k_{78}k_{77}k_{76}] = SBox[k_{79}k_{78}k_{77}k_{76}]$$
$$[k_{19}k_{18}k_{17}k_{16}k_{15}] = [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus round\_counter$$

An illustration of the PRESENT algorithm is shown in the Figure 3 and the corresponding description in the Algorithm 3 [5].

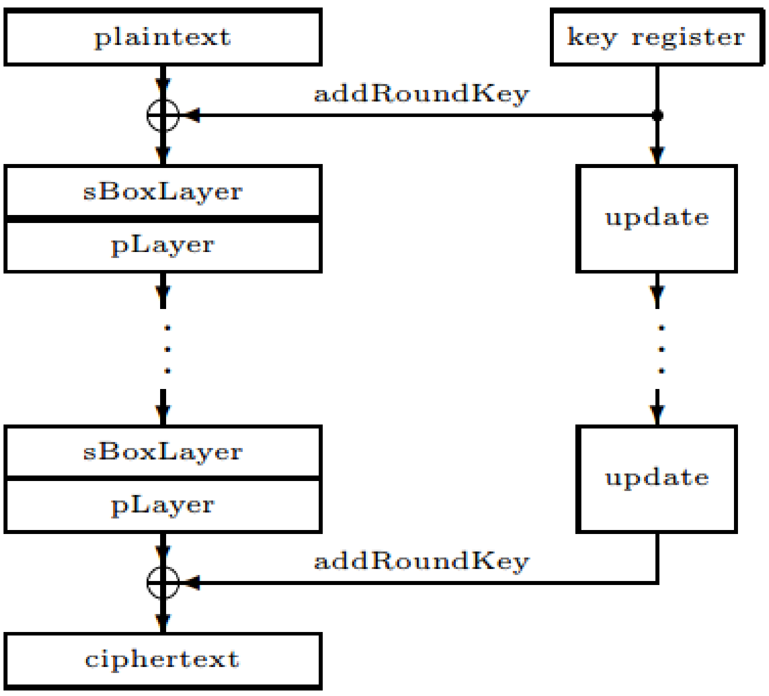

Figure 3: An illustration of PRESENT algorithm

The goal of PRESENT is to meet some specific needs, especially in the extremely resource-constrained environments. Such requirements are usually not able to indulged by AES and some other cryptographic primitives. However, PRESENT can achieve tradeoff between the security level and hardware efficiency in the constrained environments instead of compromising in any one aspect.

Algorithm 3 The encryption process of PRESENT

```
1: procedure ENCIPHER($input[64]$, $output[64]$, $K[80]$)
2: begin
3:     $state \leftarrow input$
4:     $generateRoundKeys(K)$
5:     for $i = 1$ to $31$ do
6:         $addRoundKey(state, K_i)$
7:         $sBoxLayer(state)$
8:         $pLayer(state)$
9:     $addRoundKey(state, K_{32})$
10:    $output \leftarrow state$
11: end
```

2.2 Algebraic Cryptanalysis

A new cryptanalytic method against block ciphers which is called *algebraic cryptanalysis* has been proposed for years. In contrast to the two most common cryptanalytic methods — linear cryptanalysis and differential cryptanalysis [25, 22, 21], algebraic cryptanalysis attempts to explore the algebraic structure of block cipehrs. The most common form of algebraic cryptanalysis is that the adversaries utilizes a large set of low-degree (usually quadratic) multivariate polynomial equations to describe the encryption transformation [4]. After building the system of algebraic equations, solving the system [6, 12] to recover the secret key is the next step. In order to solve such systems, several algorithms have been proposed and widely employed.

2.2.1 Specification

The essential idea of algebraic cryptanalysis is presented through two moves. By exploiting the algebraic structure of block ciphers, the first move is to build a model for a targeting cryptographic primitive by constructing a system of algebraic equations over a finite field, usually over $GF(2)$ where only two elements, 0 and 1, exist. Of course, the other finite fields can also be adoptable, e.g. $GF(2^3)$, $GF(2^7)$, etc., but not proper in this work. Furthermore, it is assumed that the S-Box of a block cipher is able to be expressed through an overdefined system of algebraic equations [14].

The second move is to solve this system in order that the secret key of the cryptographic primitive can be retrieved. Because the algebraic equations are so constructed that the solutions have a corresponding relationship with the secret key of this cryptographic primitive [8]. However, finding solutions for such a system is not trivial because the system contains a great number of variables and multivariate equations. To effectively tackle the problem, Gröbner base is a great option earlier and several techniques were developed for it, such as Buchberger algorithm, the F_4 and F_5 algorithm, XL, etc. [28, 14, 11, 10, 18]. Besides, an alternative way to solve the system has been developed afterwards. This later proposed way is to translate the algebraic system into a satisfiability(SAT) instance which is equivalent to the system and in *conjunctive normal form* (CNF) and then feed the instance to SAT-solvers, e.g. CryptoMinisat, [36, 17].

Note that the performance of algebraic cryptanalysis relies on encryption algorithms to a considerable extent when SAT solvers are utilized. In this work, we focus on two versions of the selected block ciphers — PRESENT-80 [5] and AES-128 [29].

2.2.2 SAT Problems

To solve a SAT problem is to verify whether a given logical formula (a set of Boolean clauses) is satisfiable or not which is achieved by finding out an assignment for the variables in order to evaluate the given logical formula to be true or proving such an assignment does not exist [19, 20]. Assuming the logical formula is in conjunctive normal form (CNF), it consists of a set of clauses. Any two of these clauses are associated through a conjunction (AND) and each clause is composed of literals which are variables (x)

or variable negations ($\neg x$). These literals are combined by disjunctions (OR) [34]. An example is given to illustrate a formula in conjunctive normal form (CNF)

$$(\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee x_3).$$

The above formula is evaluated to be true, when x_1 =false, x_2 = true, and x_3 =true. However, the following formula

$$(\neg x_1) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$$

may not be satisfied, which means this formula cannot be evaluated to be true for all possible assignments of variables.

The boolean expressions in CNF are usually encoded into another most general format — *DIMACS* — before they are fed to SAT solvers. For example, the corresponding *DIMACS* expressions of the above examples are displayed as follows:

```
p cnf 3 3       p cnf 3 3
−1 2 0          −1 0
1 2 0           1 3 0
1 −2 3 0        −2 −3 0
```

The first line indicates the form of the two logical formulas which is CNF. Besides, the last two integers of the first line demonstrate the number of variables and clauses, respectively. The variables x_1, x_2, x_3 are encoded into numbers $1, 2, 3$. Meanwhile, negations (e.g. $\neg x_1$) are encoded into negative numbers (e.g. -1). The following three lines are the clauses which put 0 at the end to declare a clause is finished.

In addition, the boolean expressions in CNF of the *dimacs* format for the example explained in **??** are presented in the Appendix A.

2.2.3 SAT Solvers

In this master project, SAT solving is mainly concentrated which is one of the most efficient ways for algebraic cryptanalysis and this technique has already drawn much attention in the past decades. The inputs of most SAT solvers, e.g. CryptoMiniSat [36], are in conjunctive normal form (CNF) [34]. In this condition, it is necessary to translate the algebraic system into a set of CNF clauses which are the equivalent forms of the corresponding algebraic equations. This translating course is exactly the conversion from an algebraic system to a SAT problem which was proved to be NP-complete [16, 15]. Then, these CNF clauses are fed to SAT solvers. In addition, the results obtained from SAT solvers can be directly translated to recover the secret key due to the mapping relationship between the results of SAT solving and the secret key.

The SAT solver employed in this work, CryptoMiniSat, aims to combine the advantages of several other SAT solvers in order to generate a formula which may be able to solve diverse types of problems in a reasonable time [36]. It is developed from MiniSat [17] and based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [16, 15]. This algorithm is a complete depth-first search algorithm which is backtracking-based. The idea behind the basic backtracking is to assign a truth value to one literal of a given formula which simplifies the given formula and to check repeatedly whether the given formula is satisfied. If the formula is satisfied, the assigned truth value is the expected solution. Otherwise, the opposite truth value is assigned and the same procedure as just described is repeated. The DPLL algorithm is improved over the basic backtracking algorithm through two steps. The first step is unit propagation in which a unit clause, a clause containing only one single unassigned literal, can be satisfied only if the necessary value is assigned to the single literal to make it true. The second step is pure literal elimination. If a variable in a formula only can exclusively be positive or negative literal, it's pure. This kind of variables can always be assigned in a way to make all clauses including them satisfied. Therefore, such clauses are not that helpful in solving the algebraic system and thus should be eliminated.

Another technique to solve a system of algebraic equations is Gröbner basis which is specified in [13]. However, this technique has been left behind by SAT solving in the aspect of efficiency.

2.3 Side-Channel Attacks

For the last decades, side-channel attacks have been proven to be a very effective and practical method to break cryptographic primitives. Such attacks are one kind of physical attacks which make use of the leaked physical information [37, 9, 1] to recover the secret parameters used in cryptographic operations.

2.3.1 Specification

Side-channel analyses are the attacks on cryptographic devices which exploit some characteristics specific to the implementation of cryptographic algorithms and try to take advantage of these characteristics (a.k.a. information leaked by devices) to recover the secret key of cryptographic primitives. An important point for such attacks is that they are more powerful and less general than the classical cryptanalysis because they are implementation-specific [37]. Besides, the leaked information is the so-called side-channel information and it could be electromagnetic radiation [2], timing information [23], and power consumption [21], etc. A definition which is important to understand and evolve the power consumption attacks is *trace*. A *trace* is the measured power consumption which is taken in the middle of cryptographic operations [37, 9, 1]. In addition, there are several common types for the power consumption attacks, such as simple power analysis (SPA) and differential power analysis (DPA) [24, 22, 21], template attacks [9, 1], etc.

In general, side-channel attacks associate all information gained from one single or multiple traces to recover the secret key of cryptographic primitives.

2.3.2 Power Analysis

The simple power analysis (SPA) is a method to make use of the leaked substantial side-channel information about noise and one power trace obtained in the course of encryption to generate the information which is about the secrets. If the leaked information with respect to noise is very less, the differential power analysis (DPA) is more suitable than the simple power analysis (SPA). In order to realize DPA, adversaries first need to carry out multiple encryptions which means a great deal of different plaintexts are enciphered into ciphertexts taking advantage of the same secret key and statistical analysis. Moreover, DPA intercepts many power traces and employs them to decide the correctness of a key block. Besides, it also needs to note that ciphertexts are known for DPA while the knowledge of plaintexts is not necessary. Although DPA is able to attack almost any symmetric or asymmetric cryptographic algorithms, yet it is still not optimal [24, 22, 21]. Because DPA cannot extract all information existing in each side-channel sample, while the template attacks may realize this [9, 1].

2.3.3 Template Attacks

In contrast to SPA and DPA, the template attacks as the most powerful side-channel attack dedicate themselves to model noise precisely rather than attempting to remove it. Basically, the template attacks are divided into two phases, a training phase and an attack phase. The first phase serves for collecting the power traces in the middle of encrypting random plaintexts with random keys. The second phase serves for attaining power traces during very few encryptions with the sceret key and matching the acquired power traces with one subkey of the secret key. Moreover, since the template attacks only take the key schedule as the target rather than S-Boxes, a new attack model is introduced, e.g. template attacks with a Hamming Weight (HW) model. Through this model, the acquired hamming weights of intermediate values (e.g. the hamming weights of inputs/outputs of S-Boxes) are utilized to retrieve the secret key [27, 9, 1].

2.4 Algebraic Side-Channel Attacks

The main idea of algebraic cryptanalysis is to build a system of algebraic equations for the targeted cryptographic algorithm. However, it is hard to find solutions for this algebraic system because of its great size — a large number of variables and equations. In this way, such attacks are not suitable for the block ciphers which may result in an algebraic system with great size [34, 8]. While side-channel attacks tries to exploit and make use of the leaked physical information to break block ciphers. Usually, block ciphers and their implementations are secure when only a limited number of side-channel traces (e.g. one trace) are acquired and the required effort to capture enough traces is great. In addition, such attacks cannot dig some weaknesses of block ciphers [34, 8, 9, 37]. In this context, an idea of integrating algebraic cryptanalysis with side-channel attacks and attempting to make the most of their advantages is come up with. This integration brings a more powerful cryptanalysis against block ciphers which is the algebraic side-channel attacks (ASCA).

ASCA builds a system of algebraic equations consisting of two parts. One part represents the algorithm of a block cipher itself and the other part describes the information leaked in the course of implementing the algorithm. In this way, though the number of traces is greatly limited (e.g. one trace), attackers may still be able to provide adquate information to solve the algebraic system. The reason for this is that attackers in such circumstances exploit as many leakages from all the cipher rounds as possible rather than only capturing side-channel information from one round [8, 34, 33].

3 MASCA: Mutant Algebraic Side-Channel Attacks

3.1 Motivation

The feasibility of algebraic side-channel attacks (ASCA) has already been proven by Renauld *et al.* However, their advantages and potentials might not be exploited to the greatest extent. In this section, we make some changes to the algebraic representation of cryptographic algorithms to achieve better performance, such as the improvement in solving time of SAT solvers, the reduction of the required side-channel information, etc. Such changes lead to short and simple clauses which are the so-called *mutants*. Taking advantage of mutants can decrease the large size of an algebraic system constructed by ASCA, simplify the structure of the system, etc., which results in that SAT solvers are capable of finding solutions for the system with a great speed. Eventually, the proposed MASCA can recover the secret key more efficiently.

3.2 Main Ideas

In order to make the work in [34] better in the aspect of the speed of solving SAT problems, the algebraic representation of SAT problems is tweaked. While at the very beginning, it is not very clear, what kind of algebraic representation might be considered as "good". Empirically, the size of the problem seems to be an meaningful characteristic of SAT instances. The size mentioned here indicates the number of not only variables but also equations of an algebraic system which more precisely refers to the quantity of literals and clauses of a SAT instance. In addition, thinking of the way how SAT solvers work - constructing a tree, searching it by depth-first backtracking, and attempting to prune branches efficiently when conflict clauses are discovered [3], short clauses may lead to a solution sooner than those long clauses. Therefore, the average length of clauses is a proper heuristic measure as well. Thus, MASCA imposes a constraint on the length of clauses which is determined to be 4 in this work and obtaining these short clauses is the first optimization step. So far, not all the obtained clauses are mutants. Some of them are redundant and not helpful to solve an algebraic system. Hence, distinguishing mutants and redundant clauses is the second optimization step which is realized through two filters explained later.

For PRESENT-80, the recovered Hamming weights of each round consist of 8 HW from addition of round key and 8 HW from substitution. In this way, there are at most 496 correct Hamming weights for 31 rounds. While for AES-128, there are 16 HW from addition of round key, 16 HW from substitution, and 4×13 HW from mixing columns for each round. These figures of 10 rounds are summed up and the corresponding total can reach a maximum of 788 correct Hamming weights [33]. In order to effectively represent the S-box of AES-128, Renauld *et al.* utilize a set of clauses listing all possible values of input and output of S-box, which generates 2048 clauses and their length is 9. The same method is applied to PRESENT-80, leading to 64 clauses of length 5.

Moreover, we attempted to find out the limitations that the S-Box owns on the Hamming weights of the S-Box input/output pairs. Particularly, when the input and output of a S-Box are known, the short clauses for this case are going to be included.

3.3 Specification

In this work, the unit of leakages is one byte (8 bits). Therefore, the range of both inputs/outputs of S-Box is from 0 to 255 and it is denoted as a set $\mathcal{IO} = \{io \in \mathbb{N} | io \leq 255\}$. The scope of the corresponding Hamming weights is from 0 to 8 and it is expressed as $\mathcal{W} = \{\omega \in \mathbb{N} | \omega \leq 8\}$. Let x, y describe the input and output of S-Box and ω_x, ω_y the Hamming weights of the input and output. Meanwhile, $x, y \in \mathcal{IO}$ and $\omega_x, \omega_y \in \mathcal{W}$. In this way, Hamming weight pairs are in the form (ω_x, ω_y).

In order to calculate the Hamming weight of an input/output, an important function $HW(\cdot)$ is introduced. For a given byte x, $HW(x) = \omega_x$ holds, if and only if every subset of the bits of $x = (x_1, \ldots, x_8)$, with size of $(\omega_x + 1)$, includes at least one 0 and every subset with size of $(8 - \omega_x + 1)$ includes at least one 1. In addition, the Hamming weight of a byte x can be expressed as a set of equations over $GF(2)$ with the variables indicating the bits of x, where $x = (x_1, \ldots, x_8)$.

3.3.1 The Weights of Hamming Weight Pairs

The weight of a Hamming weight pair is the count of input/output pairs of S-Box mapping to this certain Hamming weight pair and it is denoted as $\mathcal{W}_{HWP}$ in this work. Such weights of all possible Hamming weight pairs of PRESENT-80 and AES-128 are computed and presented in the Table 2 and 3.

As shown in the Table 2, there exist 37 Hamming weight pairs of PRESENT-80 whose weights are not 0. While the Table 3 demonstrates that AES-128 has 47 Hamming weight pairs with non-zero weights. For example, an input/output pair of PRESENT-80 S-Box, $x = (0,0,0,0,1,1,1,1)$ and $y = (1,0,1,0,0,0,1,0)$, is mapped to the Hamming weight pair $(4,3)$. Accordingly, one of the input/output pairs of AES-128 corresponding to the Hamming weight pair $(4,3)$ is $x = (0,0,0,1,1,1,0,1)$ and $y = (1,0,1,0,0,1,0,0)$.

in \ out	0	1	2	3	4	5	6	7	8
0	0	0	0	0	1	0	0	0	0
1	0	0	0	0	8	0	0	0	0
2	0	0	2	2	18	4	2	0	0
3	0	0	8	12	8	20	8	0	0
4	1	2	3	24	7	22	6	4	1
5	0	4	4	16	12	8	8	4	0
6	0	2	6	2	12	2	4	0	0
7	0	0	4	0	4	0	0	0	0
8	0	0	1	0	0	0	0	0	0

Table 2: Weights of Hamming weight pairs of PRESENT

3.3.2 Generating Clauses

As specified above, $HW(\cdot)$ denotes the function calculating the Hamming weight of the parameter and z denotes an eight-bit vector where $z \in \mathbb{N}$. Besides, let $\mathcal{W} = \{\omega \in \mathbb{N} | \omega \leq 8\}$ express the set of all possible

in \ out	0	1	2	3	4	5	6	7	8
0	0	0	0	0	1	0	0	0	0
1	0	0	2	0	1	3	2	0	0
2	0	2	3	8	5	4	4	2	0
3	1	1	4	17	16	10	5	2	0
4	0	3	9	11	21	16	9	1	0
5	0	1	7	10	19	14	3	2	0
6	0	0	3	7	5	8	4	0	1
7	0	1	0	2	2	1	1	1	0
8	0	0	0	1	0	0	0	0	0

Table 3: Weights of Hamming weight pairs of AES

values of Hamming weights of eight-bit vectors. Then, there exists the equation $HW(z) = \omega$ with $\omega \in \mathcal{W}$. This equation is in fact comprised of two inequalities — $HW(z) \leq \omega$ and $HW(z) \geq \omega$ — each of which is able to be expressed equivalently through a set of clauses. These clauses are the *classical* HW clauses.

Besides, the clauses are also generated for possible Hamming weight pairs of cryptographic algorihtms. As presented in the table 2 and 3, the weights of some Hamming weigh t pairs are not 0, which means there indeed exist input/output pairs of S-Box for these Hamming weight pairs, individually. For each of these Hamming weight pairs, a set of equations are built to describe the Hamming weight pair and define the S-Box itself. In [27], a set of short equations were extracted from this set makeing use of Gröbner bases and they were converted to CNF clauses through the PolyBoRi's CNF converter. However, this method works well only for those Hamming weight pairs with very small weights. For the Hamming weight pairs with high weights, i.e. the weight is greater than 7 which is expressed as $\mathcal{W}_{HWP} \geq 7$ (e.g. $\mathcal{W}_{HWP_{(4,3)}} = 24$ for PRESENT-80), long clauses come out as a result. Finding solutions for a set of long clauses is hard and time-consuming. Therefore, it should spare no efforts to avoid long clauses. In order to bypass the long clauses, another approach rather than Gröbner bases was employed. The reasonably short clauses of length ℓ $(1 \leq \ell \leq 4)$ satisfied by all input/output pairs mapping to one certain high count Hamming weight pair were added to the algebraic system, which was indeed helpful for solving the system. As demonstrated in [27], it can be concluded that the reasonably short clauses are generated through three steps. The first step is to find all possible clauses of length ℓ $(1 \leq \ell \leq 4)$ taking advantage of the *exhaustive search*. Since both input and output of S-Box are 8 bits, there exist C_{ℓ}^{16} possible clauses for each length ℓ $(1 \leq \ell \leq 4)$. The second step is to compute all possible input/output pairs corresponding to the individual Hamming weight pairs with non-zero weights. The last step is to check the created clauses for each Hamming weight pair, see if they are satisfied by the corresponding input/output pairs, and keep the satisfied clauses as the results of the individual Hamming weight pairs. So far, searching for all possible clauses for the Hamming weight pairs with non-zero weights has been finished. The Table 4 and 5 report the number of resulting clauses after performing exhaustive search for the Hamming weight pairs with high weights $(\mathcal{W}_{HWP} \geq 7)$ of PRESENT-80 and AES-128, respectively.

Pair	1	2	3	4
(1,4)	0	112	2162	19749
(2,4)	0	34	1096	13312
(3,2)	0	106	2098	19482
(3,3)	0	40	1294	15205
(3,4)	0	106	2098	19482
(3,5)	0	13	678	10430
(3,6)	0	106	2098	19482
(4,3)	0	2	252	6862
(4,4)	0	53	1710	18486
(4,5)	0	1	374	8326
(5,3)	0	19	806	11760
(5,4)	0	23	1084	14417
(5,5)	0	55	1618	17651
(5,6)	2	117	2088	19280
(6,4)	0	43	1274	14988

Table 4: Number of unfiltered clauses for Hamming weight pairs of PRESENT-80

Pair	1	2	3	4
(2,3)	2	98	1877	18374
(3,3)	0	20	788	11349
(3,4)	0	7	604	10695
(3,5)	0	36	1270	15422
(4,2)	0	57	1550	16923
(4,3)	0	24	1114	14613
(4,4)	0	0	212	7184
(4,5)	0	12	673	11027
(4,6)	2	100	1886	18178
(5,2)	1	99	2011	19271
(5,3)	0	23	1157	15252
(5,4)	0	4	499	9536
(5,5)	0	12	838	12427
(6,3)	1	107	2077	19516
(6,5)	1	100	1997	19028

Table 5: Number of unfiltered clauses for Hamming weight pairs of AES-128

Pair	1	2	3	4
(1,4)	0	112	8	9
(2,4)	0	34	52	4
(3,2)	0	106	0	0
(3,3)	0	40	378	0
(3,4)	0	106	8	9
(3,5)	0	13	326	0
(3,6)	0	106	0	0
(4,3)	0	2	10	117
(4,4)	0	53	464	90
(4,5)	0	1	18	71
(5,3)	0	19	324	0
(5,4)	0	23	52	61
(5,5)	0	55	408	0
(5,6)	2	58	0	0
(6,4)	0	43	52	61

Table 6: Number of filtered clauses for Hamming weight pairs of PRESENT-80

Pair	1	2	3	4
(2,3)	2	39	39	0
(3,3)	0	20	297	0
(3,4)	0	7	55	33
(3,5)	0	36	427	0
(4,2)	0	57	53	32
(4,3)	0	24	50	17
(4,4)	0	0	212	2754
(4,5)	0	12	26	47
(4,6)	2	41	24	15
(5,2)	1	69	28	0
(5,3)	0	23	565	0
(5,4)	0	4	38	53
(5,5)	0	12	516	0
(6,3)	1	77	16	0
(6,5)	1	70	36	0

Table 7: Number of filtered clauses for Hamming weight pairs of AES-128

3.3.3 Optimizing Clauses

The generation of clauses has been accomplished as specified in 3.3.2. However, not all of these clauses are useful and only mutants can help. These unnecessary (a.k.a. redundant) clauses raise the size of algebraic systems and thus make it more difficult for SAT solvers to find solutions for SAT instances. Accordingly, the solving time is getting longer. Therefore, extracting mutants (or eliminating redundant clauses) from the resulting clauses obtained in 3.3.2 is a very significant action. For this purpose, some measures should be taken to further process the clauses before they are fed to SAT solvers. Considering the way SAT solvers work, two filters in this master project are proposed to minimize the algebraic system, reduce redundancy, and choose clauses more carefully so that the clauses can be optimized to a as great extent as MASCA can. Thus, the optimized clauses (a.k.a. mutants) are able to improve the efficiency of solving systems and better assist the attackers to recover the secret key of cryptographic primitives.

The function of the first filter, expressed as **FilterOne** here, is to select the clauses of a certain length ℓ ($1 \leq \ell \leq 4$) which only contain the variables of inputs or outputs — we signify the bits of an input/output pair as $x = (x_1, x_2, \ldots, x_8)$ and $y = (y_1, y_2, \ldots, y_8)$ — and totally discard the other clauses of the same length ℓ. This operation has a close relationship with the power of each Hamming weight pair. Assuming we have the following notations:

- a Hamming weight pair (ω_x, ω_y) and
- all the clauses of a fixed length ℓ ($1 \leq \ell \leq 4$) comprise a corresponding set which is denoted as CLAUSE_ℓ. Then, the conjunction of all such clause sets is expressed as $\mathsf{CLAUSE} = \bigcup_{\ell=1}^{4} \mathsf{CLAUSE}_\ell$.

In this way, the core algorithm of **FilterOne** is demonstrated in the Algorithm 4 and the java implementation is shown in the Appendix C.

As shown in the Algorithm 4, **FilterOne** is not applied to the clauses of length ℓ where $\ell \leq \omega_{min}$ and the clauses of length ℓ where $\ell > \omega_{max}$ are directly removed from the corresponding set CLAUSE_ℓ. Only the clauses of length ℓ where $\omega_{min} < \ell \leq \omega_{max}$ are the target to which the **FilterOne** is employed. To keep the clauses only including either the input variables or the output variables is dependent upon the power of their corresponding Hamming weight pair. In this master project, the variables relating to the bigger Hamming weight are taken. After the usage of **FilterOne**, a great deal of unnecessary clauses are eliminated, which means the redundancy of clauses is reduced greatly.

The second filter which is denoted as **FilterTwo** is aiming at removing the *inclusion* relation between any two clause sets of different lengths ℓ where $1 \leq \ell \leq 4$. The *inclusion* relation is defined to be that each literal existing in a clause of length $< \ell$ (from the set $\mathsf{CLAUSE}_1 \cup \mathsf{CLAUSE}_2 \cup \cdots \cup \mathsf{CLAUSE}_{\ell-1}$[1]) is also included by a clause of length ℓ. Such a clause of length ℓ should be removed from the relating clause

[1] The same notations as specified for the first filter are also used for the second filter. CLAUSE_ℓ is on behalf of a set consisting of the clauses of length ℓ where $1 \leq \ell \leq 4$.

Algorithm 4 The algorithm of the **FilterOne**

```
1:  procedure FilterOne(CLAUSE)
2:  begin
3:      $\omega_{min} = min(\omega_x, \omega_y)$
4:      $\omega_{max} = max(\omega_x, \omega_y)$
5:      for $\ell = 1$ to 4 do
6:          if $\ell \leq \omega_{min}$ then
7:              leave CLAUSE$_\ell$as it is
8:          else if $\ell > \omega_{max}$ then
9:              discard CLAUSE$_\ell$
10:         else if $\omega_{min} < \ell \leq \omega_{max}$ then
11:             if $\omega_x > \omega_y$ then
12:                 remove each clause including any $y_i$, where $1 \leq i \leq 8$
13:             else if $\omega_x < \omega_y$ then
14:                 remove each clause including any $x_i$, where $1 \leq i \leq 8$
15: end
```

set CLAUSE$_\ell$. An example is given here to specify this filter. We have the clause sets of certain lengths as follows:

$$\begin{aligned}
\mathsf{CLAUSE}_1 &= \{\{x_1\}, \{x_3\}, \{x_5\}\} \\
\mathsf{CLAUSE}_2 &= \{\{x_1, x_2\}, \{x_2, x_4\}, \{x_5, x_6\}\} \\
\mathsf{CLAUSE}_3 &= \{\{x_1, x_2, x_7\}, \{x_2, x_4, x_8\}, \{x_2, x_6, x_7\}\} \\
\mathsf{CLAUSE}_4 &= \{\{x_2, x_6, x_7, x_8\}\}.
\end{aligned}$$

The clause set of length 1 CLAUSE$_1$ is kept as it is. The second filter **FilterTwo** firstly deals with the clause set of length 2, i.e. CLAUSE$_2$, and eliminates the clauses $\{x_1, x_2\}$ and $\{x_5, x_6\}$ from CLAUSE$_2$ because both clauses contain the shorter clauses $\{x_1\}$ and $\{x_5\}$ which are belonging to CLAUSE$_1$. The same operation is performed to CLAUSE$_3$ and CLAUSE$_4$, too. Then, the first round of **FilterTwo** is finished. For the second round, **FilterTwo** starts working on the clause set of length 3 and repeats the same elimination operation as carried out in the previous round — remove any clause of length 3 and 4 containing any clause of length 2. This course is repeated until the removal of the clauses of the maximum length ℓ_{max} ($\ell_{max} = 4$ here) including any clause of length ℓ_{max-1} is accomplished. The resulting clause sets of the above example are

$$
\begin{aligned}
&\mathsf{CLAUSE}_1 = \{\{x_1\},\{x_3\},\{x_5\}\}\\
&\mathsf{CLAUSE}_2 = \{\{x_2,x_4\}\}\\
&\mathsf{CLAUSE}_3 = \{\{x_2,x_6,x_7\}\}\\
&\mathsf{CLAUSE}_4 = \{\}.
\end{aligned}
$$

It is obvious that the resulting sets are much simpler compared with the previous clause sets. Since the filter is designed on a basis of the working scheme of SAT solvers and the amount of clauses does matter for SAT solvers in the course of solving SAT problems, the further simplification can ease the burden of SAT solvers. In general, the algorithm of **FilterTwo** is demonstrated in the Algorithm 5 and the corresponding java implementation is specified in the Appendix C.

Algorithm 5 The algorithm of the **FilterTwo**

```
 1: procedure FilterTwo(CLAUSE)
 2: begin
 3:     $\omega_{max} = max(\omega_x, \omega_y)$
 4:     for $\ell = 1$ to $\omega_{max}$ do
 5:         $diff = \omega_{max} - \ell$
 6:         for $j = 1$ $diff$ do
 7:             $\forall clause_{\ell+j} \in \mathsf{CLAUSE}_{\ell+j}$
 8:             $\forall clause_{\ell} \in \mathsf{CLAUSE}_{\ell}$
 9:             if $clause_{\ell+j}$ contains $clause_{\ell}$ then
10:                 remove $clause_{\ell+j}$
11: end
```

After applying the two filters to the originally generated clauses, the mutants are acquired and inserted into the algebraic system to better assist SAT solvers to find a solution. The Table 6 and 7 present the number of resulting clauses (mutants) for the Hamming weight pairs with high weights ($\mathcal{W}_{HWP} \geq 7$) of PRESENT-80 and AES-128, respectively.

4 Experiments

Through the optimization for CNF representation of Hamming weight leakages, the mutants are obtained and injected into the algebraic system, which makes the inputs of SAT solvers being of more simple and optimized structure. Therefore, the SAT solving by MASCA is able to be accelerated. Furthermore, some equations of the algebraic system may become superfluous and should be eliminated to further improve the inputs, leading to the reduction of required Hamming weight leakages.

The conducted experiments mainly serves two purposes. One purpose is to give evidence of that the solving time is greatly shortened by using MASCA compared to ASCA, based on the same "standard" amount of Hamming weight information required by ASCA. The so-called "standard" amount for PRESENT are Hamming weights of four consecutive internal rounds (64 HW) and for AES of three consecutive internal rounds (252 HW) [34, 33, 27]. The other purpose is to report that MASCA improves the quantity of Hamming weights demanded by ASCA. For this purpose, it is necessary to compare the amount of known Hamming weight information needed by MASCA with that needed by ASCA (and IASCA in [27] in the case of AES).

4.1 Experimental Settings

For the presented experiments in this section, an assumption that all given Hamming weights are correct is made. To generate and optimize CNF clauses which are the input of SAT solvers, the Java implementation of two parts are utilized. One part is the ASCA introduced by Renauld [32] and the other is the MASCA which generates the clauses corresponding to known Hamming weight information through exhaustive searach and optimizes these clauses. To solve the SAT instances generated by both ASCA and MASCA, the SAT solver Cryptominisat [36] is employed. The version of the employed Cryptominisat is 2.9.0. In order to get the expected results by processing the outcomes of Cryptominisat, another Java implementation is applied. In addition, two significant criteria are set to increase the reliability of the experiments. First of all, a time threshold is set for all experiments because the solving process of some SAT problems might take very long time which is impractical for actual attacks. In this work, the time threshold is set to be 3,600 seconds. More precisely, no matter whether the SAT problems have solutions or not, attacks are reckoned to be failed when no solution has been found in less than 3,600 seconds. Secondly, the success rate of the experiments is set on a convincing level which is determined to be higher than 90% within the time threshold. Because it would also be impractical if the success rate were too low. Furthermore, all experiments in this work are performed on a Sun X4440 server which was equipped with RAM of 128 GB and CPUs with Quad-Core AMD Opteron^{TM} Processor 8356. Each CPU is running at 2.3 GHz.

In order to make the experimental results persuasive, 100 experiments with 100 distinct plaintext/ciphertext pairs for each case in each attack scenario — consecutive and randomly distributed Hamming weights in known and unknown plaintext/ciphertext attack scenarios — are performed and a corresponding average of the solving time is calculated.

4.2 Experimental Steps

The steps of conducting the experiments are specified as follows:

a) generating CNF clauses and writting them to the files with the names in a certain form like $cnf\_\omega_x\_\omega_y.txt$ where ω_x and ω_y are the Hamming weight pair to which the CNF clauses are mapped.

b) reading clauses from the files created in the step a), adapting them according to the variables used in the algebraic system, and inserting the adapted clauses into the system to yield the SAT instance.

c) feeding the SAT instance to SAT solvers.

d) Once a solution has been found, the second Java implementation is put to use to compare the result after SAT solving and the original secret key and calculate the average solving time and the success rate if they are equal.

Note that these four steps are divided into three phases:

- clause generation — a).
- SAT instance generation — b).
- SAT solving — c) and d).

The clause generation only needs to be executed once at the very beginning of experiments. Then, experiments with different plaintext/ciphertext pairs are created during SAT instance generation. The number of plaintext/ciphertext pairs is set as a parameter. At last, the SAT solving is performed. To carry out the steps c) and d) for 100 experiments automatically, a shell script which is illustrated in the Listing 1 is executed.

Subsequently, the experiments conducted for PRESENT-80 and AES-128 and the results are individually demonstrated in 4.3 and 4.4 to give the evidence of the improvement in the performance of mutant algebraic side-channel attacks (MASCA) in this master project.

4.3 Experiments for PRESENT Algorithm

The experiments conducted for PRESENT-80 aims at proving the improvement not only in the solving time but also in the required amount of Hamming weight information. Since no great enhancement to the original PRESENT algorithm has been proposed, only the experimental results of ASCA and MASCA are compared in this subsection.

4.3.1 Improving Solving Time

At first, the experimental results supporting the improvement in solving time of MASCA using the same quantity of Hamming weight information required by ASCA are reported. As shown in the Figure 4[2]

[2] The conditions of the experiements: PRESENT-80, Hamming weights of 4 consecutive rounds (64 HW), known plaintext/ciphertext attack scenario.

```
#!/bin/bash
for i in $(seq 0 99);
do
       sat="sat_KL_" $i "_n0.txt"
       result="result_KL_" $i "_n0.txt"
       if [ -f $sat ]; then
       ./cryptominisat $sat > $result
       fi
       if [ -f $result ]; then
       java Comparison $sat $result >> tmp.txt
       echo $(date +"%T") " : sat "$i" done !"
       else
       echo $result "does not exist !"
       fi
done
java Average tmp.txt > final.txt
echo "All finished !"
```

Listing 1: Shell script for automatic run of 100 experiments

and 5[3], the improvement has been realized in both known and unknown plaintext/ciphertext attack scenarios.

In the Figure 4, it is easy to see that the solving time of MASCA and ASCA in the case of $R2-R5$ and $R26-R29$ makes no big difference. In contrast to the cases of $R2-R5$ and $R26-R29$, MASCA cuts off the solving time to a great extent in the case of the middle rounds. In fact, MASCA consumes roughly the same time to find solutions for all cases in the known plaintext/ciphertext attack scenario.

The Figure 5 suggests that compared to ASCA, MASCA reduces the solving time for all cases in the unknown plaintext/ciphertext attack scenario in the almost same level.

Moreover, to further prove the improved performance of MASCA, the results of 100 experiments with 100 different plaintext/ciphertext pairs of MASCA and ASCA using the Hamming weights of $R10$, $R11$, $R12$, and $R13$ are individually compared and directly presented in the Figure 6 (the known plaintext/ciphertext attack scenario) and Figure 7 (the known plaintext/ciphertext attack scenario).

[3] The conditions of the experiements: PRESENT-80, Hamming weights of 4 consecutive rounds (64 HW), unknown plaintext/ciphertext attack scenario.

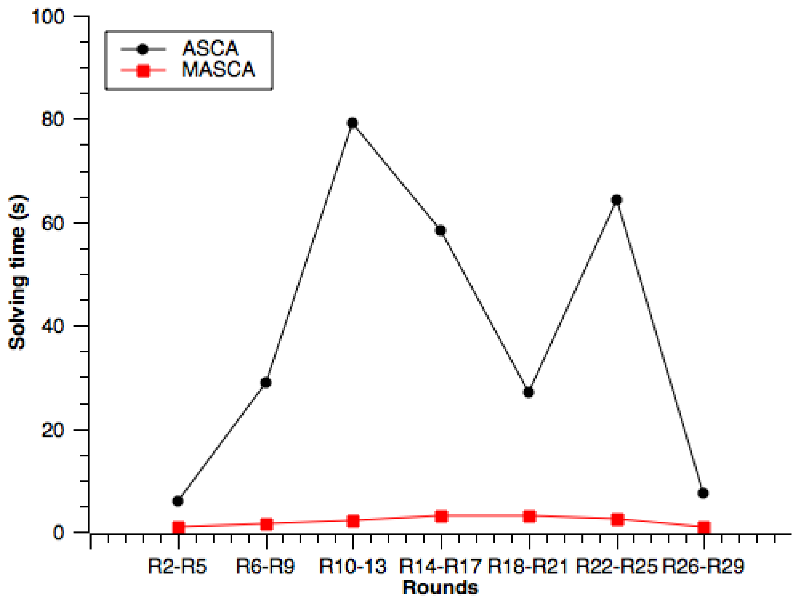

Figure 4: Solving time of MASCA and ASCA in a known plaintext/ciphertext attack scenario for PRESENT-80

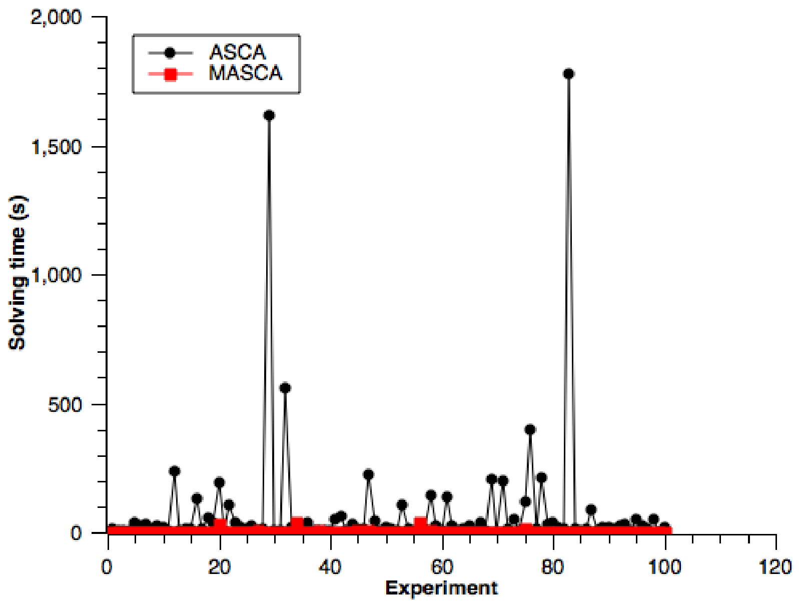

Figure 6: Solving time of MASCA and ASCA in a known plaintext/ciphertext attack scenario for PRESENT-80 using Hamming weights of $R10$, $R11$, $R12$, and $R13$

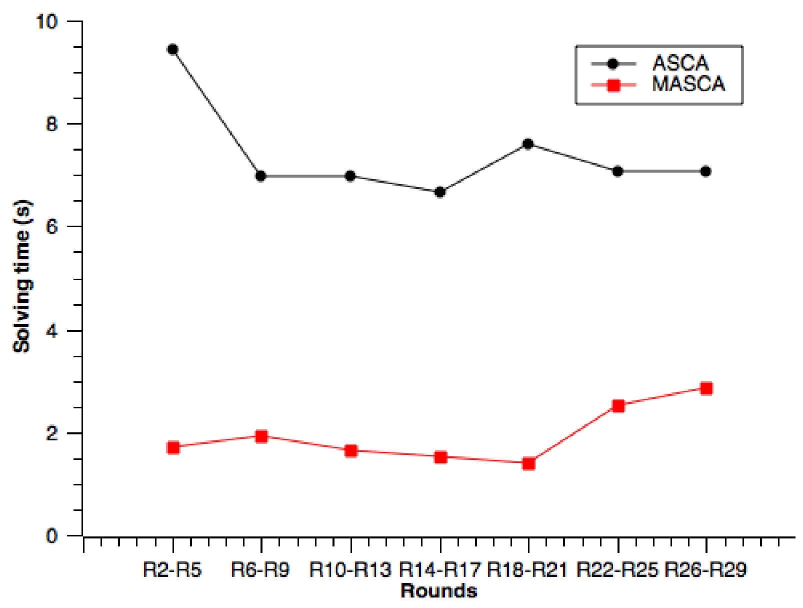

Figure 5: Solving time of MASCA and ASCA in an unknown plaintext/ciphertext attack scenario for PRESENT-80

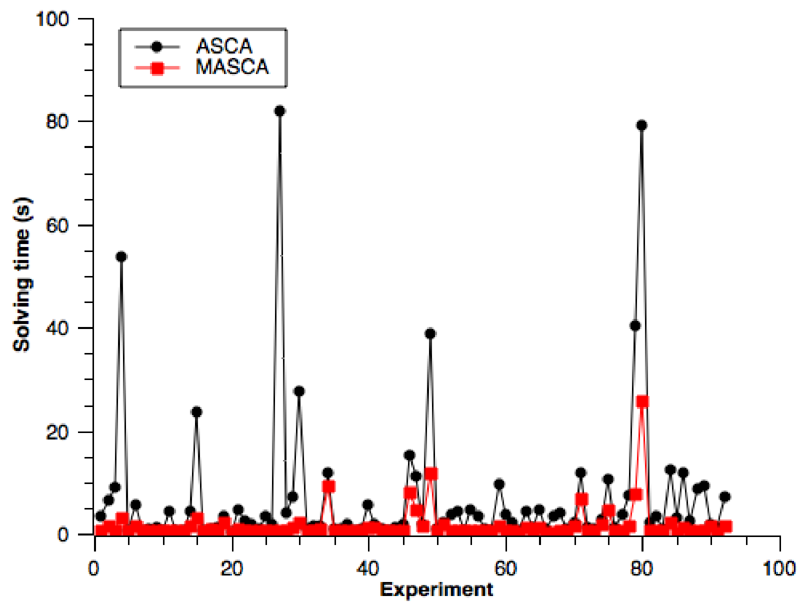

Figure 7: Solving time of MASCA and ASCA in an unknown plaintext/ciphertext attack scenario for PRESENT-80 using Hamming weights of $R10$, $R11$, $R12$, and $R13$

4.3.2 Reducing Hamming Weight Leakages

The following Table 8 and Table 9 illustrate the reduction of the required amount of Hamming weights needed by MASCA in both known and unknown plaintext/ciphertext attack scenario.

From the Table 8, it is easy to tell that merely 48 Hamming weights (3 consecutive rounds) can already supply adequate information to the system to recover the secret key in the known plaintext/ciphertext attack scenario, whereas ASCA is in need of 64 Hamming weights (4 consecutive rounds). Besides, if known Hamming weights are distributed at random in the same attack scenario, 190 Hamming weights suffice for MASCA, while ASCA demands more than 240 Hamming weights.

Table 9 indicates that MASCA utilizes Hamming weights of less than four consecutive rounds to retrieve the secret key in the unknown plaintext/ciphertext attack scenario. This quantity is not improved very well. But the solving time in this condition is decreased as shown in Figure 5. In addition, ASCA demands at least 400 Hamming weights when known Hamming weights are distributed randomly. However, MASCA requires at most 280 Hamming weight in the same circumstances.

Attack	ASCA	MASCA
PRESENT consecutive	4 rounds 64 HW	3 rounds 48 HW
PRESENT random	>240 HW	190 HW

Table 8: Quantity of Hamming weighs required by MASCA and ASCA for PRESENT-80 in a known plaintext/ciphertext attack scenario

Attack	ASCA	MASCA
PRESENT consecutive	4 rounds 64 HW	$<$ 4 rounds 62 HW
PRESENT random	>400 HW	280 HW

Table 9: Quantity of Hamming weighs required by MASCA and ASCA for PRESENT-80 in an unknown plaintext/ciphertext attack scenario

4.4 Experimental Results for AES

The experiments for AES-128 are contributed to provide evidence for the better performance of MASCA in this work. But some approaches of improving algebraic side-channel attacks have been developed in the past years, therefore, the experimental results of MASCA are compared not only with ASCA but also with IASCA in [27].

4.4.1 Improving Solving Time

The first part of experiments for AES-128 suggests that MASCA consumes less time to get the secret key taking advantage of the same amount of Hamming weights as ASCA. As shown in the Figure 8[4]

[4] The conditions of the experiments: AES-128, Hamming weights of 3 consecutive rounds (252 HW), known plaintext/ciphertext attack scenario.

and 9[5], the improvement in solving time for AES-128 has been realized in both known and unknown plaintext/ciphertext attack scenarios.

Figure 8 shows that MASCA indeed improves the solving time for AES-128 in the known plaintext/ciphertext attack scenarios for all cases — especially for the middle rounds. As a matter of fact, both Figure 4 and Figure 8 suggest that it takes more time for adversaries to attack the middle rounds than to attack the rounds being closer to the head or the end. The reason for this may be that SAT solvers are in need of more time to reach the Hamming weights from the intermediate rounds [27]. Figure 4 and Figure 8 also indicate that the solving time of ASCA has a big difference when attacking the intermediate rounds and the rounds being closer to the head or the end. IASCA in [27] cuts the solving time of ASCA in half and MASCA shortens this once cut solving time to a greater extent, in other words, reduces the solving time of MASCA in [27] by half.

Figure 9 illustrates the reduction of the solving time of MASCA for AES-128 in the unknown plaintext/ciphertext attack scenarios. Compared with ASCA, IASCA in [27] makes a slight improvement, while MASCA cuts down the solving time obviously. Besides, there is no big difference of the solving time of MASCA for any consecutive rounds, which is similar to PRESENT-80 shown in the Figure 5.

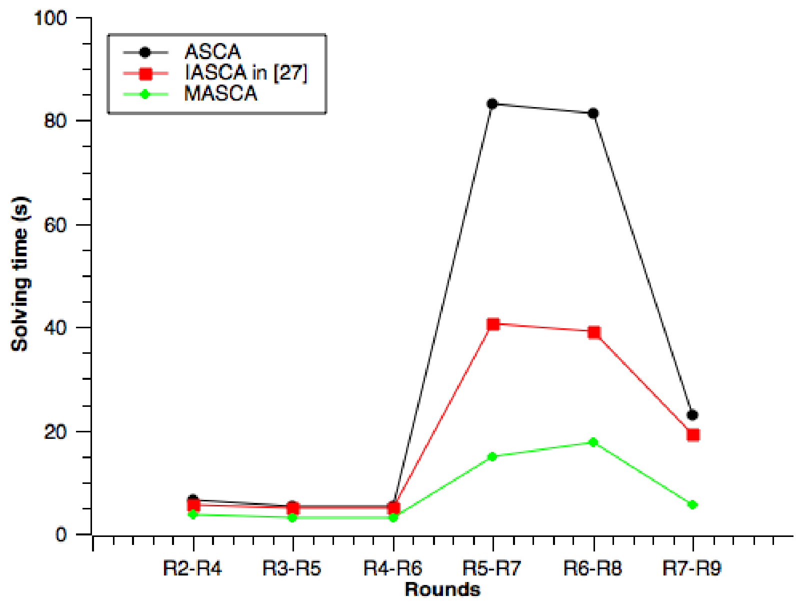

Figure 8: Solving time of MASCA, IASCA in [27], and ASCA in a known plaintext/ciphertext attack scenario for AES-128

[5] The conditions of the experiments: AES-128, Hamming weights of 3 consecutive rounds (252 HW), unknown plaintext/ciphertext attack scenario.

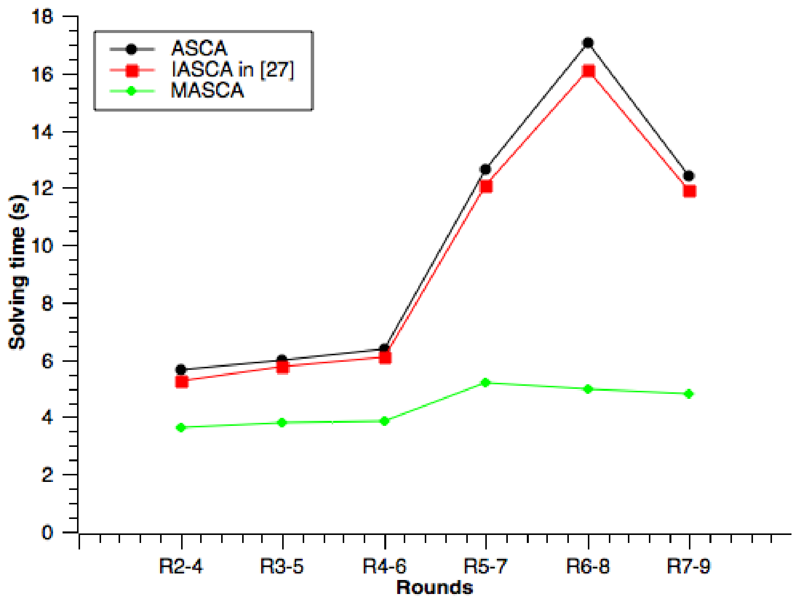

Figure 9: Solving time of MASCA, IASCA in [27], and ASCA in an unknown plaintext/ciphertext attack scenario for AES-128

To further demonstrate the performance of MASCA is better than ASCA and IASCA in [27], 100 experiments with 100 different plaintext/ciphertext pairs utilizing the Hamming weights of $R5$, $R6$, and $R7$ are carried out. The comparison of the experimental results are displayed in the Figure 10 and Figure 11.

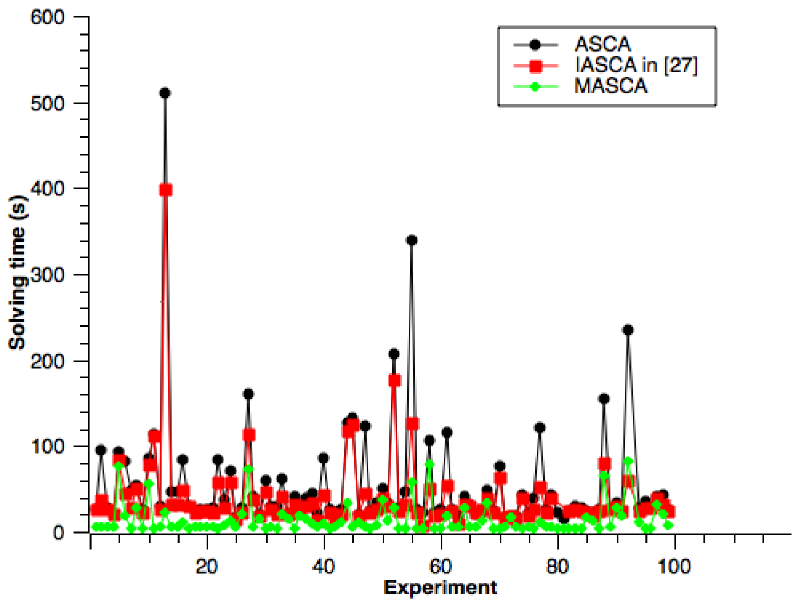

Figure 10: Solving time of MASCA, IASCA in [27], and ASCA in a known plaintext/ciphertext attack scenario for AES-128 using Hamming weights of $R5$, $R6$, and $R7$

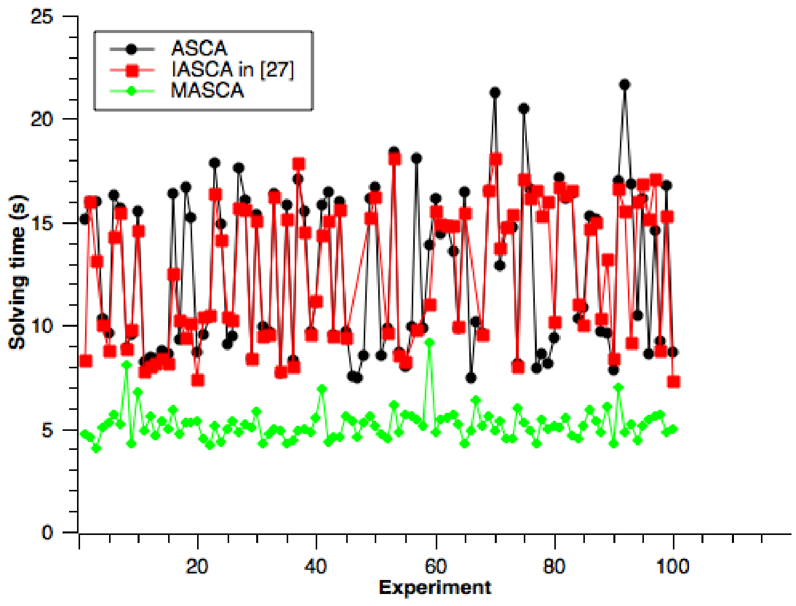

Figure 11: Solving time of MASCA, IASCA in [27], and ASCA in an unknown plaintext/ciphertext attack scenario for AES-128 using Hamming weights of $R5$, $R6$, and $R7$

4.4.2 Reducing Hamming Weight Leakages

The purpose of the second part of experiments for AES-128 is to support that MASCA requires less Hamming weight information than ASCA and IASCA in [27] to break cryptographic primitives. The Table 10 and 11 present the amount of required Hamming weights of ASCA, IASCA in [27], and MASCA to support the improvement in the quantity of Hamming weight information of MASCA.

It is obvious from Table 10 that MASCA demands only 152 Hamming weights (less than two consecutive rounds) to break cryptographic primitives when known Hamming weights are consecutive in the known plaintext/ciphertext attack scenario. This amount is over one round less than that needed by ASCA and even less than the required quantity of IASCA in [27]. Meanwhile, merely 230 Hamming weights are already sufficient for MASCA to better help solve the algebraic system when known Hamming weights are distributed at random, while ASCA is in need of 551 Hamming weights and IASCA in [27] 394 Hamming weights.

Table 11 shows the comparison of the quantity of Hamming weights needed by ASCA, IASCA in [27], and MASCA in the unknown plaintext/ciphertext attack scenarios. In the case of consecutive Hamming weights, MASCA demands Hamming weight information of only a little bit more than 2 rounds to get the secret key. It means precisely that MASCA needs all the Hamming weights of two consecutive rounds R_i and R_{i+1} (168 HW) and one more leakage of the round R_{i+2}. Besides, in the case of randomly distributed Hamming weights, 460 Hamming weights for MASCA are good enough to supply adequate information to the system compared with 551 Hamming weights for ASCA and 472 Hamming weights for IASCA in [27].

Attack	ASCA	IASCA in [27]	MASCA
AES consecutive	3 rounds 252 HW	2 rounds 168 HW	< 2 rounds 152 HW
AES random	551 HW	394 HW	230 HW

Table 10: Quantity of Hamming weighs required by ASCA, IASCA in [27], and MASCA for AES-128 in a known plaintext/ciphertext attack scenario

Attack	ASCA	IASCA in [27]	MASCA
AES consecutive	3 rounds 252 HW	< 3 rounds 184 HW	< 3 rounds 169 HW
AES random	551 HW	472 HW	460 HW

Table 11: Quantity of Hamming weighs required by ASCA, IASCA in [27], and MASCA for AES-128 in an unknown plaintext/ciphertext attack scenario

5 Error Tolerance

The previous improvement in this work is based on an assumption that all known Hamming weights are correct. However, there exist some physical effects in the real attacks which have influences on the power values in the case of the power consumption side-channel and they are denoted as noise (such as electronic noise, quantization noise, and switching noise). Because of the noise, the emitted side-channel information may result in incorrect Hamming weights [30, 31, 35, 38]. If the equations built on these erroneous Hamming weights are inserted into the algebraic system, it would lead to incorrect solutions or even make SAT problems unsatisfiable. Therefore, the capability of MASCA is extended in this section so that MASCA can deal with the errors occurring in the real attacks.

5.1 Specification

As specified in 3.3.2, the equation $HW(z) = \omega$ can be expressed taking advantage of two inequalities $HW(z) \leq \omega$ and $HW(z) \geq \omega$. In this way, the varying range of the Hamming weight is $[\omega, \omega]$. However, if erroneous Hamming weights occur or to say that the values of correct Hamming weights are uncertain, this range can be gradually enlarged so that the correct Hamming weights can be included in this range with a high probability. For example, assuming that the correct Hamming weight is in the interval $[\omega, \omega + 1]$ which leads to two inequalities $HW(z) \geq \omega$ and $HW(z) \leq \omega + 1$. The clauses which describe these two inequalities are inserted into the algebraic system. Analogously, if the values of Hamming weights are more uncertain, the interval containing the correct Hamming weight is extended to $[\omega, \omega+2]$. In this way, the clauses describing $HW(z) \geq \omega$ and $HW(z) \leq \omega + 2$ are combined and inserted into the system. In general, assume there is an interval of Hamming weights $[\omega_1, \omega_2]$, then let $\mathbb{EC}_i$ (error classes introduced in [27]) describe the corresponding set of all possible intervals of length i where $i = \omega_2 - \omega_1$. More precisely,

$$\text{if } \omega_2 - \omega_1 = 0, \text{ then error class } \mathbb{EC}_0 = \{[\omega, \omega]\};$$
$$\text{if } \omega_2 - \omega_1 = 1, \text{ then error class } \mathbb{EC}_1 = \{[\omega - 1, \omega], [\omega, \omega + 1]\};$$
$$\text{if } \omega_2 - \omega_1 = 2, \text{ then error class } \mathbb{EC}_2 = \{[\omega - 2, \omega], [\omega, \omega + 2], [\omega - 1, \omega + 1]\};$$

and so on. In this master project, five error classes ($\mathbb{EC}_0$, $\mathbb{EC}_1$, $\mathbb{EC}_2$, $\mathbb{EC}_3$, and $\mathbb{EC}_4$) are considered and for each error class, only one interval is taken into account, which is presented in Table 12. But we only focus on the error classes $\mathbb{EC}_0$, $\mathbb{EC}_1$, and $\mathbb{EC}_2$. Therefore, not only $(\mathbb{EC}_0, \mathbb{EC}_0)$, $(\mathbb{EC}_1, \mathbb{EC}_1)$, and $(\mathbb{EC}_2, \mathbb{EC}_2)$ but also $(\mathbb{EC}_0, \mathbb{EC}_1)$, $(\mathbb{EC}_1, \mathbb{EC}_0)$, $(\mathbb{EC}_0, \mathbb{EC}_2)$, $(\mathbb{EC}_2, \mathbb{EC}_0)$, $(\mathbb{EC}_1, \mathbb{EC}_2)$ as well as $(\mathbb{EC}_2, \mathbb{EC}_1)$ are taken into consideration.

As specified in the subsection 3.3.2, the clauses mapping to Hamming weight pairs of PRESENT-80 and AES-128 are generated by applying the exhaustive search. Table 4 and 5 show the corresponding number of clauses for Hamming weight pairs with high count ($\mathcal{W}_{HWP} \geq 7$) in the case that Hamming weights are

Error Class	Interval
$\mathbb{EC}_0$	$[\omega, \omega]$
$\mathbb{EC}_1$	$[\omega, \omega+1]$
$\mathbb{EC}_2$	$[\omega-1, \omega+1]$
$\mathbb{EC}_3$	$[\omega-2, \omega+1]$
$\mathbb{EC}_4$	$[\omega-2, \omega+2]$

Table 12: The targeted intervals for error classes

error-free which means the error class $\mathbb{EC}_0$. Using the same method — exhaustive search — to the error classes $\mathbb{EC}_1$ and $\mathbb{EC}_2$ to generate clauses, a slight change is made.

Suppose there is a Hamming weight pair $(2,3)$, the corresponding set of all possible Hamming weight pairs based on $(2,3)$ for the error class $\mathbb{EC}_1$ is

$$\mathsf{HWP}_{\mathbb{EC}_1,(2,3)} = \{(2,3),(2,4),(3,3),(3,4)\}.$$

Then, check the weight in the Table 2 (using PRESENT-80 as an example here) for each Hamming weight pair of $\mathsf{HWP}_{\mathbb{EC}_1,(2,3)}$. For example, the weight of $(2,3)$ is 2 which is signified as $\mathcal{W}_{HWP_{(2,3)}} = 2$. Similarly, $\mathcal{W}_{HWP_{(2,4)}} = 18$, $\mathcal{W}_{HWP_{(3,3)}} = 12$, and $\mathcal{W}_{HWP_{(3,4)}} = 8$. What should be noted here is that a Hamming weight pair is not taken into consideration when the weight of the Hamming weight pair is equal to 0, which means no valid input/output pair for this Hamming weight pair. Assuming $\mathcal{W}_{HWP_{(3,4)}} = 0$, then

$$\mathsf{HWP}_{\mathbb{EC}_1,(2,3)} = \{(2,3),(2,4),(3,3)\}.$$

The next step is to create all possible clauses of length ℓ where $1 \le \ell \le 4$ and compute all possible input/output pairs for each Hamming weight pair ($\mathcal{W}_{HWP} > 0$) of $\mathsf{HWP}_{\mathbb{EC}_1,(2,3)}$. Subsequently, the tasks are to check whether the clauses satisfy all the input/output pairs and keep the valid clauses. For an input/output pair (x, y), these valid clauses are describing the inequalities

$$2 \le HW(x) \le 3$$
$$3 \le HW(y) \le 4$$

at the same time. For the error class $\mathbb{EC}_2$, the corresponding set of all possible Hamming weight pairs based on $(2,3)$ is

$$\mathsf{HWP}_{\mathbb{EC}_2,(2,3)} = \{(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,2),(3,3),(3,4)\}.$$

From the Table 2, it is easy to check the individual weights and find that $\mathcal{W}_{HWP_{(1,2)}} = 0$ and $\mathcal{W}_{HWP_{(1,3)}} = 0$ which lead to

$$\text{HWP}_{\mathbb{EC}_2,(2,3)} = \{(1,4),(2,2),(2,3),(2,4),(3,2),(3,3),(3,4)\}.$$

In the following, the similar procedures are performed as specified for the error class $\mathbb{EC}_1$ and the resulting clauses describes the inequalities

$$1 \leq HW(x) \leq 3$$
$$2 \leq HW(y) \leq 4$$

at the same time. The number of clauses for high count Hamming weight pairs of PRESENT-80 and AES-128 for the error classes $\mathbb{EC}_1$ and $\mathbb{EC}_2$ are displayed in the Table 13, 14, 15, and 16.

Pair	1	2	3	4
(1,4)	0	14	740	10491
(2,4)	0	4	296	5739
(3,2)	0	1	120	4208
(3,3)	0	0	42	2744
(3,4)	0	0	74	2858
(3,5)	0	0	122	3858
(3,6)	0	53	1366	14880
(4,3)	0	0	12	2073
(4,4)	0	0	28	2568
(4,5)	0	0	140	4586
(5,3)	0	0	140	4797
(5,4)	0	8	368	7033
(5,5)	0	16	756	10955
(5,6)	0	68	1566	16104
(6,4)	0	31	1076	13375

Table 13: Number of unfiltered clauses for Hamming weight pairs of PRESENT-80 in the case of $\mathbb{EC}_1$

Pair	1	2	3	4
(2,3)	0	0	94	3515
(3,3)	0	0	2	739
(3,4)	0	0	0	715
(3,5)	0	0	155	4577
(4,2)	0	0	164	4825
(4,3)	0	0	7	898
(4,4)	0	0	2	640
(4,5)	0	0	127	4031
(4,6)	0	43	1199	13987
(5,2)	0	6	464	8210
(5,3)	0	0	116	3932
(5,4)	0	0	98	3472
(5,5)	0	0	342	7154
(6,3)	0	36	1139	13656
(6,5)	0	51	1370	15146

Table 14: Number of unfiltered clauses for Hamming weight pairs of AES-128 in the case of $\mathbb{EC}_1$

Pair	1	2	3	4
(1,4)	0	6	618	9682
(2,4)	0	0	84	3390
(3,2)	0	1	118	4120
(3,3)	0	0	38	2054
(3,4)	0	0	4	727
(3,5)	0	0	64	2258
(3,6)	0	0	118	3728
(4,3)	0	0	4	1233
(4,4)	0	0	4	506
(4,5)	0	0	6	1285
(5,3)	0	0	12	1779
(5,4)	0	0	4	646
(5,5)	0	0	26	2026
(5,6)	0	0	128	4293
(6,4)	0	0	84	3665

Table 15: Number of unfiltered clauses for Hamming weight pairs of PRESENT-80 in the case of $\mathbb{EC}_2$

Pair	1	2	3	4
(2,3)	0	0	72	2907
(3,3)	0	0	2	507
(3,4)	0	0	0	129
(3,5)	0	0	0	437
(4,2)	0	0	35	2037
(4,3)	0	0	0	76
(4,4)	0	0	0	0
(4,5)	0	0	0	114
(4,6)	0	0	48	2260
(5,2)	0	0	107	3614
(5,3)	0	0	3	526
(5,4)	0	0	0	157
(5,5)	0	0	0	350
(6,3)	0	0	70	2921
(6,5)	0	0	69	2835

Table 16: Number of unfiltered clauses for Hamming weight pairs of AES-128 in the case of $\mathbb{EC}_2$

Pair	1	2	3	4
(1,4)	0	14	28	1
(2,4)	0	4	8	31
(3,2)	0	1	0	0
(3,3)	0	0	42	0
(3,4)	0	0	0	65
(3,5)	0	0	122	0
(3,6)	0	53	20	0
(4,3)	0	0	0	54
(4,4)	0	0	28	1876
(4,5)	0	0	4	79
(5,3)	0	0	140	0
(5,4)	0	8	0	118
(5,5)	0	16	332	0
(5,6)	0	68	8	0
(6,4)	0	31	38	49

Table 17: Number of filtered clauses for Hamming weight pairs of PRESENT-80 in the case of $\mathbb{EC}_1$

Pair	1	2	3	4
(2,3)	0	0	6	0
(3,3)	0	0	2	0
(3,4)	0	0	0	40
(3,5)	0	0	155	0
(4,2)	0	0	11	59
(4,3)	0	0	3	29
(4,4)	0	0	2	589
(4,5)	0	0	9	43
(4,6)	0	43	42	43
(5,2)	0	6	33	0
(5,3)	0	0	116	0
(5,4)	0	0	5	41
(5,5)	0	0	342	0
(6,3)	0	36	56	0
(6,5)	0	51	28	0

Table 18: Number of filtered clauses for Hamming weight pairs of AES-128 in the case of $\mathbb{EC}_1$

Pair	1	2	3	4
(1,4)	0	6	26	8
(2,4)	0	0	2	38
(3,2)	0	1	0	0
(3,3)	0	0	38	0
(3,4)	0	0	0	2
(3,5)	0	0	64	0
(3,6)	0	0	4	0
(4,3)	0	0	0	10
(4,4)	0	0	4	402
(4,5)	0	0	0	11
(5,3)	0	0	12	0
(5,4)	0	0	0	2
(5,5)	0	0	26	0
(5,6)	0	0	4	0
(6,4)	0	0	0	64

Table 19: Number of filtered clauses for Hamming weight pairs of PRESENT-80 in the case of $\mathbb{EC}_2$

Pair	1	2	3	4
(2,3)	0	0	6	0
(3,3)	0	0	2	0
(3,4)	0	0	0	1
(3,5)	0	0	0	0
(4,2)	0	0	0	28
(4,3)	0	0	0	0
(4,4)	0	0	0	0
(4,5)	0	0	0	4
(4,6)	0	0	0	26
(5,2)	0	0	5	0
(5,3)	0	0	3	0
(5,4)	0	0	0	3
(5,5)	0	0	0	0
(6,3)	0	0	6	0
(6,5)	0	0	4	0

Table 20: Number of filtered clauses for Hamming weight pairs of AES-128 in the case of $\mathbb{EC}_2$

Of course, there also exists redundancy in the clauses for the error classes $\mathbb{EC}_1$ and $\mathbb{EC}_2$. To remove the redundancy and get the mutants, **FilterOne** and **FilterTwo** are adopted. The quantities of the resulting clauses (mutants) of PRESENT-80 and AES-128 for $\mathbb{EC}_1$ and $\mathbb{EC}_2$ are presented in the Table 17, 18, 19, and 20. In addition, the number of filtered clauses (mutants) for mix error classes of PRESENT-80 and AES-128 are demonstrated in the Appendix B.

5.2 Experiments for Error Tolerance

In order to prove that MASCA is capable of dealing with erroneous Hamming weights, some experiments have been conducted. To specify the experimental environment, some measures are introduced [27]. *Correctness* describes the probability that the selected interval contains the correct Hamming weight. Assume there is an interval in which the correct Hamming weight is contained, the correctness is increased accordingly when the bounds of the interval are increased, . *Descriptiveness* indicates the quantity of side-channel information which is supplied through the equations describing the interval of Hamming weights.

In the experiments presented here, 496,000 Hamming weight predictions for PRESENT-80 and 788,000 for AES-128 are obtained through 1,000 measurements. For each Hamming weight prediction, a likelihood vector $\mathcal{L}$ is acquired. The likelihood vectors for PRESENT-80 $\mathcal{L}_{P-80}$ and AES-128 $\mathcal{L}_{A-128}$ are shown in the Table 21 and 22. The certainty that the interval includes the correct Hamming weight is measured by summing up the likelihood of the Hamming weight predictions and the Hamming weight is included in the interval. In order to specify a convincing level of certainty, a certainty threshold $\mathcal{T}$ is introduced, cf. [27]. The summed-up likelihood of Hamming weight predictions must exceed the certainty threshold $\mathcal{T}$ so that the interval can be expressed as a set of clauses which is inserted into the algebraic system as the input of SAT solvers.

HW	Likelihood $\mathcal{L}_{P-80}$
0	0.37%
1	3.13%
2	10.91%
3	21.93%
4	27.23%
5	21.97%
6	10.95%
7	3.12%
8	0.39%

Table 21: A certainty vector for PRESENT-80

HW	Likelihood $\mathcal{L}_{A-128}$
0	0.41%
1	3.11%
2	10.90%
3	21.89%
4	27.37%
5	21.91%
6	10.92%
7	3.11%
8	0.38%

Table 22: A certainty vector for AES-128

In order to determine the error tolerance of MASCA, two facts are evaluated. First of all, the distribution of error classes using distinct certainty thresholds from $\mathcal{T}_{all} = \{80\%, 85\%, 90\%, 94\%, 95\%, 98\%, 99\%\}$ needs to be determined to fix the minimum certainty threshold which may lead to the correctness 100%. Under this consideration, the error classes $\mathbb{EC}_3$ and $\mathbb{EC}_4$ are also taken into account. The distribution of the five error classes is presented in the Table 23 [27]. As specified above, the higher a certainty threshold is, the higher correctness can be reached. However, the descriptiveness is decreased. When the certainty threshold is equal to or greater than 95% ($\mathcal{T} \geq 95\%$), the correctness reaches 100%. In addition, the probability of $\mathbb{EC}_0$ is 0 for $\mathcal{T} = 99\%$.

$\mathcal{T}$(%)	$\mathbb{EC}_0$(%)	$\mathbb{EC}_1$(%)	$\mathbb{EC}_2$(%)	$\mathbb{EC}_3$(%)	$\mathbb{EC}_4$(%)	Correctness (%)
80	35	47	18	0	0	82
85	23	64	13	1	0	94
90	14	45	36	5	0	99
94	11	38	38	13	0	99
95	9	29	44	18	0	100
98	4	18	31	43	4	100
99	0	10	23	47	20	100

Table 23: The distribution of error classes

Secondly, the minimum amount of Hamming weight information (of consecutive rounds) as well as the corresponding solving time are analyzed when the clauses which express the identified intervals of Hamming weights are inserted into the algebraic system. The experiments for PRESENT-80 and AES-128 are individually conducted 100 times with 100 distinct plaintext/ciphertext pairs and a time threshold for the experiments in this section is set to be 100 seconds. The experimental results are depicted in the Table 24 and 25.

$\mathcal{T}$(%)	HW Rounds	solving time (s)
80	$< R1 - R3$ (40 HW)	2.08
85	$< R1 - R3$ (40 HW)	3.96
90	$R1 - R3$ (48 HW)	5.81
94	$< R1 - R4$ (56 HW)	10.37
95	$R1 - R5$ (80 HW)	12.12
98	$R1 - R13$ (208 HW)	47.29
99	$R1 - R31$ (496 HW)	608.01 (13%)
80	$R2 - R10$ (144 HW)	1.32
85	$R2 - R10$ (144 HW)	6.14
90	$R2 - R15$ (224 HW)	41.52
94	$R2 - R22$ (336 HW)	83.8
95	$R2 - R29$ (448 HW)	145.39 (79%)
98	$R2 - R30$ (464 HW)	323.97 (23%)
99	$R1 - R31$ (496 HW)	608.01 (13%)

Table 24: Experimental results of PRESENT-80 with error tolerance

$\mathcal{T}$(%)	HW Rounds	solving time (s)
80	$R1$ (84 HW)	3.73
85	$R1$ (84 HW)	3.95
90	$< R1 - R2$ (116 HW)	5.39
94	$< R1 - R2$ (116 HW)	6.94
95	$R1 - R2$ (168 HW)	10.61
98	$R1 - R6$ (504 HW)	23.50 (79%)
99	$R1 - R10$ (788 HW)	992.87 (17%)
80	$R3 - R7$ (420 HW)	5.46
85	$R3 - R7$ (420 HW)	6.38
90	$R3 - R7$ (420 HW)	7.83
94	$R3 - R7$ (420 HW)	43.49
95	$R3 - R7$ (420 HW)	51.35(83%)
98	$R3 - R9$ (588 HW)	867.46 (41%)
99	$R1 - R10$ (788 HW)	992.87 (17%)

Table 25: Experimental results of AES-128 with error tolerance

As shown in the Table 24, the Hamming weight rounds of PRESENT-80 are selected from two positions. First of all, the Hamming weight rounds containing the first round $R1$ are chosen. When the certainty threshold $\mathcal{T} = 80\%$ and 85%, the required information are 40 Hamming weights of more than two rounds (the first two rounds and the input of S-Box of the third round). For $\mathcal{T} = 90\%$, 48 Hamming weights of three rounds are sufficient and 56 Hamming weights of more than three rounds (the first three rounds and the input of S-Box of the fourth round) for $\mathcal{T} = 94\%$. In addition, the correctness can be 100% when $\mathcal{T}$ reaches 95%, 98%, or 99%. For $\mathcal{T} = 95\%$ and 98%, the first 5 rounds and the first 13 rounds are demanded to make the experiments 100% solved. However, although all Hamming weights of 31 rounds of PRESENT-80 are put to use when $\mathcal{T} = 99\%$, still only 13% of the experiments can be successfully solved. Meanwhile, it also means it is impossible to choose the Hamming weights of the intermediate rounds of PRESENT-80 to get more experiments solved when $\mathcal{T} = 99\%$. The intermediate rounds are just the second selected position. When $\mathcal{T} = 95\%$, 98%, and 99%, solutions cannot be found for all experiments. For $\mathcal{T} = 95\%$, $R2 - R29$ are selected to get 79% of experiments solved with the average solving time 145.39 seconds and $R2$–$R30$ for $\mathcal{T} = 98\%$ to get 23% of experiments solved with the average solving time 323.97 seconds.

The Table 25 presents two groups of experimental results for AES-128. For the first group, the Hamming weights are selected from the first round $R1$. In this situation, the first round $R1$ is good enough for $\mathcal{T} = 80\%$ and 85% while Hamming weights of more than one round (the first round $R1$ and the input/output of S-Box of the second round $R2$) are needed when $\mathcal{T} = 90\%$ and 94%. Furthermore, the correctness reaches 100% when $\mathcal{T} = 95\%$ and only the first two rounds are already sufficient. Besides, the correctness can also be 100% when $\mathcal{T} = 98\%$ and 99% for which not all experiments are able to be solved, even though all Hamming weights of 10 rounds are used for $\mathcal{T} = 99\%$. For the second group, the Hamming weights of the intermediate rounds are selected. Five internal rounds $R3 - R7$ suffice for $\mathcal{T} = 80\%$, 85%, 90%, and 94%. When $\mathcal{T} = 95\%$, still $R3 - R7$ are chosen and they get 83% of experiments solved, meanwhile, the correctness is 100%.

6 Conclusion

In this master project, an improved algebraic side-channel attack which is named as *Mutant algebraic side-channel attack* (MASCA) is proposed. MASCA has two main features, improvement in the performance and error-tolerance. The first feature is realized by employing mutants which are attained by optimizing the representation of algebraic systems through exhaustive search and two filters. The proposed filters bring a significant decrease in the solving time based on the same amount of Hamming weights required by ASCA as well as the reduction of the Hamming weight information needed to solve algebraic systems. In order to support the good performance of MASCA, some experiments have been conducted and the experimental results have been compared and analyzed. Since there already exist some algorithms of improving algebraic side-channel attacks applied to AES and almost none for PRESENT, the experimental results of ASCA, IASCA in [27], and MASCA are compared for AES-128 to demonstrate the further enhancement of MASCA, while for PRESENT-80, only the comparison of the results of ASCA and MASCA is carried out. When the known Hamming weights are consecutive in known plaintext/ciphertext attack scenarios, the Hamming weight amount required by MASCA for PRESENT-80 is one round less than ASCA. Even though the demanded quantity for PRESENT-80 in unknown plaintext/ciphertext attack scenarios has not been reduced greatly, the corresponding solving time has indeed been shortened. Furthermore, in the case of randomly distributed Hamming weights, MASCA makes a good progress in both attack scenarios for PRESENT-80. For AES-128, MASCA makes an obvious improvement not only in the solving time but also in the needed number of Hamming weight leakages in both attack scenarios no matter the known Hamming weights are consecutive or distributed at random.

The so-called error tolerance is the second feature of MASCA. The reason why this feature is necessary and some basics have already been explained. Besides, also some experiments, for which the distribution of error classes and the minimum required amount of Hamming weight information are two important aspects, have been performed to demonstrate that MASCA is capable of dealing with incorrect Hamming weights.

In future work, we will endeavor to fit more features in MASCA and make further improvement. First of all, redundant clauses could be further diminished. Although the proposed algorithms have attempted to eliminate the unnecessary clauses, there still exists redundancy in the clauses. Secondly, the distribution of error classes could be analyzed more precisely so that MASCA would be more practical. Last but not least, we try to develop a new technique to increase the success rate which might be more than 95% or even 100% in some certain conditions more effectively so that the persuasiveness of experiments would be increased accordingly. In this way, the practicability of MASCA can be further enhanced.

References

[1] M.A.E. Aabid, S. Guilley, and P. Hoogvorst. Template attacks with a power model. 2007.

[2] D. Agrawal, B. Archambeault, J.R. Rao, and P. Rohatgi. The em side-channel(s). In *Secure Integrated Circuits and Systems*, volume 2523, pages 29–45. Springer Berlin Heidelberg, 2003.

[3] G.V. Bard. *Algebraic Cryptanalysis*. Springer, 2009.

[4] A. Biryukov and C. D. Cannière. Block ciphers and systems of quadratic equations. In *Fast Software Encryption*, volume 2887, pages 274–289. Springer Berlin Heidelberg, 2003.

[5] 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*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[6] S. Bulygin and M. Brickenstein. Obtaining and solving systems of equations in key variables only for the samll variants of aes. *Mathematics in Computer Science*, 3:185–200, 2010.

[7] S. Bulygin and J. Buchmann. Algebraic cryptanalysis of the round-reduced and side channel analysis of the full printcipher-48. In *Cryptology and Network Security*, volume 7092, pages 54–75, 2011.

[8] C. Carlet, J.C. Faugère, C. Goyet, and G. Renault. Analysis of the algebraic side channel attack. *Journal of Cryptographic Engineering*, 2(1):45–62, 2012.

[9] S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In *CHES*, pages 13–28, 2002.

[10] Jean charles Faugère. A new efficient algorithm for computing gröbner bases (f4). In *IN: ISSAC '02: PROCEEDINGS OF THE 2002 INTERNATIONAL SYMPOSIUM ON SYMBOLIC AND ALGEBRAIC COMPUTATION*, pages 75–83, 2002.

[11] C. Cid, S. Murphy, and M. Robshaw. *Algebraic Aspects of the Advanced Encryption Standard*. Springer US, 2006.

[12] C. Cid, S. Murphy, and M.J.B. Robshaw. Computational and algebraic aspects of the advanced encryption standard. In *Proceedings of the Seventh International Workshop on Computer Algebra in Scientific Computing, CASC 2004*, pages 93–103, 2004.

[13] C. Cid and R.P. Weinmann. Block cipher: Algebraic cryptanalysis and gröbner bases. In *Gröbner Bases, Coding, and Cryptography*, pages 307–327. Springer, 2009.

[14] N.T. Courtois and J. Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In *Advances in Cryptology - CRYPTO 2002*, volume 2501, pages 267–287, 2002.

[15] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[16] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[17] N. Eén and N. Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[18] Jean-Charles Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (f5), 2004.

[19] O. Fourdrinoy, É. Grégoire, B. Mazure, and L. Saïs. Eliminating redundant clauses in sat instances. In *Proceedings of the 4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR '07, pages 71–83, Berlin, Heidelberg, 2007. Springer-Verlag.

[20] M. Heule, M. Järvisalo, F. Lonsing, M. Seidl, and A. Biere. Clause elimination for sat and qsat. *Journal of Artificial Intelligence Research*, pages 127–168, 2015.

[21] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. pages 388–397. Springer-Verlag, 1999.

[22] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1:5–27, 2011.

[23] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. pages 104–113. Springer-Verlag, 1996.

[24] S. Mangard. A simple power-analysis (spa) attack on implementation of the aes key expansion. In *Information Security and Cryptology - ICISC 2002*, volume 2587, pages 343–358, 2003.

[25] M. Matsui. Linear cryptanalysis method for des cipher. In *Advances in Cryptology — EUROCRYPT '93*, pages 386–397. Springer Berlin Heidelberg, 2001.

[26] Alfred J. Menezes, Paul C. van Oorschot, and Scoot A. Vanstone. *Handbook of Applied Cryptography*. Springer, 1996.

[27] M.S.E. Mohamed, S. Bulygin, M. Zohner, A. Heuser, M. Walter, and J. Buchmann. Improved algebraic side-channel attack on aes. *Journal of Cryptographic Engineering*, 3(3):139–156, 2013.

[28] S. Murphy and M.J.B. Robshaw. Essential algebraic structure within the aes. In *Advances in Cryptology - CRYPTO 2002*, volume 2442, pages 1–16, 2002.

[29] NIST. *Advanced Encryption Standard (AES) (FIPS PUB 197)*. National Institute of Standards and Technology, November 2001.

[30] Y. Oren, M. Kirschbaum, T. Popp, and A. Wool. Algebraic side-channel analysis in the presence of errors. In Stefan Mangard and François-Xavier Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 428–442. Springer, 2010.

[31] Yossef Oren and Avishai Wool. Tolerant algebraic side-channel analysis of AES. Cryptology ePrint Archive, Report 2012/092, 2012.

[32] M. Renauld. Simulating algebraic side channel attacks ascatocnf converter. http://www.ecrypt.eu.org/tools/ascatocnf.

[33] M. Renauld, F. X. Standaert, and N. V. Charvillon. Algebraic side-channel attacks on the AES: Why time also matters in DPA. In *CHES*, pages 97–111, 2009.

[34] Mathieu Renauld and François-Xavier Standaert. Algebraic side-channel attacks. In *Inscrypt*, pages 393–410, 2009.

[35] L. Song, L. Hu, S.Sun, Z. Zhang, D. Shi, and R. Hao. Error-tolerant algebraic side-channel attacks using bee. In *Information and Communications Security*, volume 8958, pages 1–15. Springer-Verlag, 2015.

[36] M. Soos. Cryptominisat 2.5.0. In *SAT Race competitive event booklet*, July 2010.

[37] F.-X. Standaert. Introduction to side-channel attacks. In *Secure Integrated Circuits and Systems*, pages 27–42. Springer US, 2009.

[38] X. Zhao, F. Zhang, S. Guo, T. Wang, Z. Shi, H. Liu, and K. Ji. Mdasca: An enhanced algebraic side-channel attack for error tolerance and new leakage model exploitation. In *Constructive Side-Channel Analysis and Secure Design*, volume 7275, pages 231–248. Springer Berlin Heidelberg, 2012.

A An Example of Boolean Expressions in CNF

The following is an example of boolean expressions in conjunctive normal form (CNF) of the *dimacs* format which describes the S-Box $\mathbf{S} = \{3, 0, 2, 5, 7, 1, 6, 4\}$ that is introduced in **??**.

```
p cnf 6 24
1 2 3 -4 0
1 2 3 5 0
1 2 3 6 0
1 2 -3 -4 0
1 2 -3 -5 0
1 2 -3 -6 0
1 -2 3 -4 0
1 -2 3 5 0
1 -2 3 -6 0
1 -2 -3 4 0
1 -2 -3 -5 0
1 -2 -3 6 0
-1 2 3 4 0
-1 2 3 5 0
-1 2 3 6 0
-1 2 -3 -4 0
-1 2 -3 -5 0
-1 2 -3 6 0
-1 -2 3 4 0
-1 -2 3 5 0
-1 -2 3 -6 0
-1 -2 -3 4 0
-1 -2 -3 -5 0
-1 -2 -3 -6 0
```

B Number of Clauses for Mix Error Classes

Pair	1	2	3	4
(1,4)	0	112	8	9
(2,4)	0	14	36	16
(3,2)	0	19	42	0
(3,3)	0	8	366	0
(3,4)	0	5	26	37
(3,5)	0	10	256	0
(3,6)	0	106	0	0
(4,3)	0	0	2	104
(4,4)	0	1	160	1940
(4,5)	0	0	14	60
(5,3)	0	0	184	0
(5,4)	0	8	20	136
(5,5)	0	22	336	0
(5,6)	0	77	8	0
(6,4)	0	31	50	57

Table 26: Number of filtered clauses for Hamming weight pairs of PRESENT-80 in the case of $(\mathbb{EC}_0, \mathbb{EC}_1)$

Pair	1	2	3	4
(1,4)	0	34	32	4
(2,4)	0	34	32	4
(3,2)	0	75	8	0
(3,3)	0	1	120	0
(3,4)	0	11	26	39
(3,5)	0	0	168	0
(3,6)	0	53	24	0
(4,3)	0	1	8	72
(4,4)	0	2	428	1595
(4,5)	0	0	12	75
(5,3)	0	19	322	0
(5,4)	0	20	4	136
(5,5)	0	53	334	0
(5,6)	2	49	0	0
(6,4)	0	43	38	51

Table 27: Number of filtered clauses for Hamming weight pairs of PRESENT-80 in the case of $(\mathbb{EC}_1, \mathbb{EC}_0)$

Pair	1	2	3	4
(1,4)	0	112	8	9
(2,4)	0	6	26	44
(3,2)	0	19	42	0
(3,3)	0	3	300	0
(3,4)	0	0	2	80
(3,5)	0	3	230	0
(3,6)	0	10	26	0
(4,3)	0	0	0	98
(4,4)	0	0	4	1088
(4,5)	0	0	8	51
(5,3)	0	0	158	0
(5,4)	0	0	0	117
(5,5)	0	8	184	0
(5,6)	0	22	14	0
(6,4)	0	25	50	44

Table 28: Number of filtered clauses for Hamming weight pairs of PRESENT-80 in the case of $(\mathbb{EC}_0, \mathbb{EC}_2)$

Pair	1	2	3	4
(1,4)	0	34	32	0
(2,4)	0	34	16	8
(3,2)	0	71	8	0
(3,3)	0	1	118	0
(3,4)	0	2	6	71
(3,5)	0	0	164	0
(3,6)	0	51	24	0
(4,3)	0	0	4	61
(4,4)	0	0	122	3027
(4,5)	0	0	6	54
(5,3)	0	1	80	0
(5,4)	0	2	2	99
(5,5)	0	0	172	0
(5,6)	0	42	36	0
(6,4)	0	20	2	107

Table 29: Number of filtered clauses for Hamming weight pairs of PRESENT-80 in the case of $(\mathbb{EC}_2, \mathbb{EC}_0)$

Pair	1	2	3	4
(1,4)	0	6	26	8
(2,4)	0	0	2	38
(3,2)	0	1	0	0
(3,3)	0	0	38	0
(3,4)	0	0	0	2
(3,5)	0	0	64	0
(3,6)	0	0	4	0
(4,3)	0	0	0	45
(4,4)	0	0	4	552
(4,5)	0	0	0	48
(5,3)	0	0	128	0
(5,4)	0	0	0	64
(5,5)	0	8	114	0
(5,6)	0	16	14	0
(6,4)	0	25	38	38

Table 30: Number of filtered clauses for Hamming weight pairs of PRESENT-80 in the case of ($\mathbb{EC}_1$, $\mathbb{EC}_2$)

Pair	1	2	3	4
(1,4)	0	14	28	0
(2,4)	0	4	8	23
(3,2)	0	1	8	0
(3,3)	0	0	42	0
(3,4)	0	0	0	43
(3,5)	0	0	118	0
(3,6)	0	51	20	0
(4,3)	0	0	0	13
(4,4)	0	2	428	1595
(4,5)	0	0	6	1175
(5,3)	0	0	12	0
(5,4)	0	0	0	42
(5,5)	0	0	128	0
(5,6)	0	34	24	0
(6,4)	0	8	0	105

Table 31: Number of filtered clauses for Hamming weight pairs of PRESENT-80 in the case of ($\mathbb{EC}_2$, $\mathbb{EC}_1$)

Pair	1	2	3	4
(2,3)	0	38	71	0
(3,3)	0	0	136	0
(3,4)	0	0	21	73
(3,5)	0	24	352	0
(4,2)	0	5	30	76
(4,3)	0	0	11	59
(4,4)	0	0	26	1914
(4,5)	0	1	27	57
(4,6)	2	33	35	15
(5,2)	0	12	26	0
(5,3)	0	0	164	0
(5,4)	0	0	11	59
(5,5)	0	6	469	0
(6,3)	0	54	30	0
(6,5)	0	54	32	0

Table 32: Number of filtered clauses for Hamming weight pairs of AES-128 in the case of ($\mathbb{EC}_0$, $\mathbb{EC}_1$)

Pair	1	2	3	4
(2,3)	0	11	31	0
(3,3)	0	2	184	0
(3,4)	0	0	2	73
(3,5)	0	2	210	0
(4,2)	0	22	40	86
(4,3)	0	1	29	68
(4,4)	0	0	31	1773
(4,5)	0	0	19	53
(4,6)	1	34	39	33
(5,2)	0	54	30	0
(5,3)	0	13	406	0
(5,4)	0	2	25	55
(5,5)	0	3	397	0
(6,3)	1	58	42	0
(6,5)	0	81	27	0

Table 33: Number of filtered clauses for Hamming weight pairs of AES-128 in the case of ($\mathbb{EC}_1$, $\mathbb{EC}_0$)

Pair	1	2	3	4
(2,3)	0	25	52	0
(3,3)	0	0	107	0
(3,4)	0	0	2	64
(3,5)	0	0	162	0
(4,2)	0	4	18	97
(4,3)	0	0	7	45
(4,4)	0	0	3	1154
(4,5)	0	0	0	50
(4,6)	0	1	23	64
(5,2)	0	11	23	0
(5,3)	0	0	103	0
(5,4)	0	0	3	44
(5,5)	0	0	123	0
(6,3)	0	37	28	0
(6,5)	0	24	39	0

Table 34: Number of filtered clauses for Hamming weight pairs of AES-128 in the case of $(\mathbb{EC}_0, \mathbb{EC}_2)$

Pair	1	2	3	4
(2,3)	0	11	31	0
(3,3)	0	1	141	0
(3,4)	0	0	1	63
(3,5)	0	2	175	0
(4,2)	0	12	30	89
(4,3)	0	0	1	86
(4,4)	0	0	0	629
(4,5)	0	0	8	59
(4,6)	0	20	28	108
(5,2)	0	14	31	0
(5,3)	0	1	258	0
(5,4)	0	0	0	57
(5,5)	0	0	116	0
(6,3)	0	12	32	0
(6,5)	0	3	40	0

Table 35: Number of filtered clauses for Hamming weight pairs of AES-128 in the case of $(\mathbb{EC}_2, \mathbb{EC}_0)$

Pair	1	2	3	4
(2,3)	0	0	6	0
(3,3)	0	0	2	0
(3,4)	0	0	0	2
(3,5)	0	0	0	0
(4,2)	0	0	5	67
(4,3)	0	0	2	20
(4,4)	0	0	0	178
(4,5)	0	0	0	26
(4,6)	0	0	6	47
(5,2)	0	6	31	0
(5,3)	0	0	73	0
(5,4)	0	0	0	17
(5,5)	0	0	72	0
(6,3)	0	28	39	0
(6,5)	0	21	21	0

Table 36: Number of filtered clauses for Hamming weight pairs of AES-128 in the case of $(\mathbb{EC}_1, \mathbb{EC}_2)$

Pair	1	2	3	4
(2,3)	0	0	6	0
(3,3)	0	0	2	0
(3,4)	0	0	0	33
(3,5)	0	0	117	0
(4,2)	0	0	0	35
(4,3)	0	0	0	2
(4,4)	0	0	0	134
(4,5)	0	0	2	30
(4,6)	0	16	12	113
(5,2)	0	0	9	0
(5,3)	0	0	6	0
(5,4)	0	0	0	25
(5,5)	0	0	81	0
(6,3)	0	0	6	0
(6,5)	0	0	38	0

Table 37: Number of filtered clauses for Hamming weight pairs of AES-128 in the case of $(\mathbb{EC}_2, \mathbb{EC}_1)$

C Java Implementation of the Two Proposed Filter

```java
private void filter1ClausesForSingleHWPair(String key, List<int[]> clauses, int err){
    String[] tmp = key.split(" ");
    int[] hwPair = new int[2];
    hwPair[0] = Integer.valueOf(tmp[0]);
    hwPair[1] = Integer.valueOf(tmp[1]);

    // compare the power of the current HW pair in order to
    // decide which group of variables should be contained in clauses
    int p0 = hwPair[0];
    int p1 = hwPair[1];
    if(p0 > 4){
      p0 = 8 - p0;
    }
    if(p1 > 4){
      p1 = 8 - p1;
    }
    int pMax = Math.max(p0, p1);
    int pMin = Math.min(p0, p1);

    HWCC hwcc = hwInfo0.get(key);
    if(err == 1){
      hwcc = hwInfo1.get(key);
    }
    else if(err == 2){
      hwcc = hwInfo2.get(key);
    }

    for(int[] c0 : clauses){
      if(c0.length < pMin){
        hwcc.getcLengthChosen().get(c0.length).add(this.deepcopy(c0));
      }
      else if(c0.length <= pMax && c0.length >= pMin){
        if(c0.length == 2){
          if((hwcc.getcLengthOriginal().get(1).size() == 0)
              || (hwcc.getcLengthOriginal().get(1).size() > 0 && pMin > 1)){
            hwcc.getcLengthChosen().get(c0.length).add(this.deepcopy(c0));
            continue;
          }
        }
        boolean mark = true;
        if(p0 > p1){
          // filter the clauses containing only input variables (because : p0 >= p1)
          for(int i = 0; i < c0.length; i++){

            if(Math.abs(c0[i]) >= 9 ){ // contains only input variable
              mark = false;
              break;
            }
          }

        }
        else if(p0 < p1){
          // filter the clauses containing only output variables (because : p0 < p1)
          for(int i = 0; i < c0.length; i++){

            if(Math.abs(c0[i]) < 9 ){ // contains only output variable
              mark = false;
              break;
            }
          }
        }

        if(mark){
          hwcc.getcLengthChosen().get(c0.length).add(this.deepcopy(c0));
        }
      }

  } // end check all clauses
}
```

Listing 2: The java implementation of the first filter

```java
// the clauses of length 1
List<int[]> cLen1 = hwcc0.getcLengthOriginal().get(1);

if(cLen1.size() == 16){

  hwcc0.getCounter2AfterFilters()[1] = 16;
  hwcc0.getcLengthChosen().put(1, this.deepcopy(cLen1));
  hwcc0.setChosenClauses(this.deepcopy(cLen1));
  for(int i = 2; i <= this.neededCnfLength; i++){
    List<int[]> newC = new ArrayList<int[]>();
    hwcc0.getcLengthChosen().put(i, newC);
  }
}
else if(cLen1.size() < 16){
  //*************** part 1 of filter 2 *******************

  // keep the clauses of length 1 to be chosen clauses
  hwcc0.getcLengthChosen().put(1, this.deepcopy(cLen1));
  hwcc0.getChosenClauses().addAll(this.deepcopy(cLen1));
  hwcc0.getCounter2AfterFilters()[1] = cLen1.size();

  // set the variables in the clauses of length to be solutions
  // and create a set for them
  int[] solutions = new int[cLen1.size()];
  for(int i = 0; i < solutions.length; i++){
    solutions[i] = Math.abs(cLen1.get(i)[0]);
  }
  for(int i = 2; i <= hwMax; i++){
    List<int[]> target = hwcc0.getcLengthChosen().get(i);
    if(err == 3 || err == 4){
      target = hwcc0.getcLengthOriginal().get(i);
    }
    List<int[]> cF2P1 = this.filter2Part1(solutions, target);
    hwcc0.getcLengthChosen().put(i, this.deepcopy(cF2P1));
  }
}

//*************** part 2 of filter 2 *******************
for(int i = 2; i <= hwMax; i++){
  int diff = hwMax - i;
  if(diff == 0){
    break;
  }
  for(int j = 1; j <= diff; j++){
    List<int[]> cF2P2 = this.filter2Part2(hwcc0.getcLengthChosen().get(i),
    hwcc0.getcLengthChosen().get(i + j));
    hwcc0.getcLengthChosen().put(i + j, cF2P2);
  }
}
// sum up the results
List<int[]> chosenC = new ArrayList<int[]>();
int[] counter2 = new int[this.neededCnfLength+1];
for(Map.Entry<Integer, List<int[]>> entry1 : hwcc0.getcLengthChosen().entrySet()){
  chosenC.addAll(entry1.getValue());
  counter2[entry1.getKey()] = entry1.getValue().size();
}
hwcc0.setChosenClauses(this.deepcopy(chosenC));
hwcc0.setCounter2AfterFilters(counter2);
```

Listing 3: The java implementation of applying the second filter

```java
private List<int[]> filter2Part1(int[] solutions, List<int[]> clauses){

    // remove the clauses(length 2) containing
    // the variables which are from the solution set
    List<int[]> result = new ArrayList<int[]>();
    for(int[] c : clauses){

      boolean containFlag = false;
      for(int i = 0; i < c.length; i++){
        for(int j = 0; j < solutions.length; j++){
          if(Math.abs(c[i]) == solutions[j]){
            containFlag = true;
            break;
          }
        }
      }

      if(!containFlag){
        if(!result.contains(c)){
          result.add(c);
        }
      }

    }
    return result;
}
```

Listing 4: The java implementation of the part 1 of the second filter

```java
private List<int[]> filter2Part2(List<int[]> srcSet, List<int[]> targetSet){
    List<int[]> tmpSet = this.deepcopy(targetSet);

    for(int[] srcClause : srcSet){
      Iterator<int[]> tmp = tmpSet.iterator();
      while(tmp.hasNext()){
        int[] targetClause = tmp.next();
        boolean isSubset = this.isSubset(srcClause, targetClause);
        if(isSubset){
          tmp.remove();
        }
      }
    }
    return tmpSet;
}
```

Listing 5: The java implementation of the part 2 of the second filter