Reducing Overdefined Systems of Polynomial Equations Derived from Small Scale Variants of the AES via Data Mining Methods

Jana Berušková[1], Martin Jureček[1*], Olha Jurečková[1]

[1*]Faculty of Information Technology, Czech Technical University in Prague, Prague, Czechia.

*Corresponding author(s). E-mail(s): martin.jurecek@fit.cvut.cz;
Contributing authors: berusjan@fit.cvut.cz; jurecolh@fit.cvut.cz;

Abstract

This paper deals with reducing the secret key computation time of small scale variants of the AES cipher using algebraic cryptanalysis, which is accelerated by data mining methods. This work is based on the known plaintext attack and aims to speed up the calculation of the secret key by processing the polynomial equations extracted from plaintext-ciphertext pairs. Specifically, we propose to transform the overdefined system of polynomial equations over GF(2) into a new system so that the computation of the Gröbner basis using the F4 algorithm takes less time than in the case of the original system. The main idea is to group similar polynomials into clusters, and for each cluster, sum the two most similar polynomials, resulting in simpler polynomials. We compare different data mining techniques for finding similar polynomials, such as clustering or locality-sensitive hashing (LSH). Experimental results show that using the LSH technique, we get a system of equations for which we can calculate the Gröbner basis the fastest compared to the other methods that we consider in this work. Experimental results also show that the time to calculate the Gröbner basis for the transformed system of equations is significantly reduced compared to the case when the Gröbner basis was calculated from the original non-transformed system. This paper demonstrates that reducing an overdefined system of equations reduces the computation time for finding a secret key.

Keywords: AES, Algebraic Cryptoanalysis, Gröbner Bases, Data Mining, Partitioning Around Medoids, Locality Sensitive Hashing

1 Introduction

Algebraic cryptanalysis is gaining more and more attention and has been successfully applied in many cryptosystems. It is especially relevant for the AES cipher [1], which was designed to be mainly resistant to standard techniques such as linear and differential cryptanalysis. Algebraic cryptanalysis consists of two main parts. First, a cryptosystem is transformed into a polynomial equation system over some finite field. Then, a secret key is revealed by solving this system using existing methods, such as SAT solvers or algorithms for computing Gröbner basis. Note that the problem of solving a system of multivariate polynomial equations over a finite field is an NP-complete problem, even if the system consists of quadratic equations and the field is GF(2) [2].

The AES cipher can be considered a map from a finite set to a finite set. According to the Universal Mapping Theorem [3], any such map can be written as a polynomial system over GF(p), where p is some prime number. Following this theorem, it is possible to represent the AES as a system of polynomial equations over GF(p), no matter how complex the structure of the AES is. However, the design of the AES can be described using standard algebraic operations, which allows the relatively easy construction of a system of polynomial equations, as described in Section 4. This work considers this system of polynomial equations over GF(2), and the unknown variables correspond to secret key bits.

Also, this work deals with small scale variants of the AES [4] since it is computationally unfeasible to apply the algebraic cryptanalysis proposed in this paper using our working machine. Algebraic cryptanalysis is based on a known plaintext attack in our work, i.e., we assume that we have several pairs of plaintexts and the corresponding ciphertexts. From each such pair, we create a system of polynomial equations over GF(2), where the number of equations equals the length of a secret key. For more than one pair, we obtain an overdefined system, i.e., the number of equations is greater than the number of unknown variables. In this work, we conducted experiments with relatively large systems extracted from several hundred plaintext-ciphertext pairs. Our experimental results indicate that the more such systems, the lower the entropy of the secret key.

The contribution of this paper is an approach for mining the data about the secret key from big data that are in the form of large polynomial equations over GF(2). Specifically, we proposed an approach based on data mining methods to process the large polynomial equations in a more appropriate form (w.r.t. time complexity) for the algorithm F4 [5] for computing Gröbner bases. The main idea behind using data mining methods is to use these approaches to cluster the equations in groups of similar equations. Then, for each cluster, we find the two most similar polynomials (w.r.t. some similarity function) and sum them. The resulting polynomials are given as input to the F4 algorithm to compute Gröbner basis. According to the Elimination theorem (see Section 3), the secret key is trivial to extract using polynomials from reduced Gröbner basis. Our experimental results demonstrate that finding the secret key of some small scale variants of the AES is significantly faster for the F4 algorithm than when using the conventional way, i.e., the F4 algorithm is directly applied to the polynomial equations without the preprocessing part.

The rest of this paper is organized as follows. Section 2 provides an overview of the related works focused on algebraic cryptanalysis. Section 3 introduces technical terms to understand the mathematical background, specifically the theory of Gröbner bases. Rather than presenting the AES, a well-known cipher described in many papers, we describe the structure of small scale variants of the AES cipher in this section. In Section 4, we present the construction of the system of polynomial equations that can be used to model these variants. Section 5 presents three existing methods, two of them from the data mining domain, which we used for reducing overdefined systems of polynomial equations, making the computation of Gröbner bases using the F4 algorithm more efficient and thereby speeding up the finding of the secret key. Section 6 presents the experimental results of the three methods and compares them in terms of computational times, memory requirements, and other metrics. Section 7 concludes the work and provides directions for future work.

2 Related Work

Algebraic cryptanalysis of small scale variants of AES has been the subject of numerous prior studies. The authors of [6] defined two sets of small variations of AES that differ in the form of the final round. In addition, they provided instructions for constructing systems of equations corresponding to these small scale variants of the AES. The idea of using the zero-dimensional Gröbner-representation for AES was proposed in [7]. Based on their research results, the authors demonstrated the possibility of an attack using a single plaintext-ciphertext pair for the full AES-128.

The interpretation of AES as a system of equations over GF(2^8) is presented in [8]. The authors also introduced a new technique that further simplifies the analysis of the AES. Instead of describing the AES encryption process in terms of operations on an array of bytes, they represented the data as column vectors. The authors also introduced a new iterated block cipher, the

Big Encryption System (BES), which operates on 128-byte blocks with a 16-byte key.

The authors of [9] presented some algebraic aspects of the representation of AES as a system of polynomial equations according to the BES approach. Using stepwise substitutions, they were able to get rid of all intermediate variables and obtain two systems whose solution exactly corresponds to revealing the secret key.

In [10], researchers conducted specific studies on the linear transformation and the system of multivariate polynomial equations of AES-256. Then, they proposed a zero-dimensional Gröbner basis construction method for the system by selecting the appropriate term order and variable order. In addition, a Gröbner basis attack scheme was proposed, which has lower complexity than a brute force attack.

The authors of [11] described AES as a system of overdefined sparse quadratic equations over $GF(2)$. They also suggested a new technique called an XSL attack for the family of XSL-ciphers that includes AES. In [12], the authors explained the rationale behind various claims for AES key recovery attacks based on the XSL technique. To better understand the XSL algorithm and its behavior, the authors of [13] examined it more thoroughly and explained it more succinctly. However, the authors have also demonstrated that the XSL algorithm cannot solve the system arising from the AES, as presented in [11]. After discussing some of the XSL algorithm's alternatives, the authors concluded that it is unlikely for the algorithm to solve the AES system of equations efficiently in its current state.

The authors of [14] investigated methods for obtaining and solving key variable-only equations that arise during the cryptanalysis of small AES variants. Unlike many other attempts, the method described in this paper is unusual in that the researchers focused on obtaining equations for only the key variables and used a zero-dimensional Gröbner representation for the AES but worked over $GF(2)$.

The authors of [15] investigated algebraic attacks on the AES. They began with a brief history of AES, followed by a description of the AES algorithm, and then discussed various techniques for solving systems of multivariate quadratic equations over arbitrary fields, such as relinearization and XL algorithm.

In [16], the authors worked with the small scale variants of AES. They also represented the ciphers as systems of polynomial equations over GF(2) that only included the variables of the initial key and found the secret keys using Gröbner bases. The authors compared the performance of Gröbner bases to that of an SAT solver, demonstrating the supremacy of Gröbner bases in solving polynomial systems for larger variants of the AES.

3 Background

This chapter presents the basic definitions of all mathematical terms used in this work. We will mainly focus on the area of Gröbner bases and the definition of small scale variants of the AES.

3.1 Gröbner Bases

In this section, we introduce the concept of the Gröbner basis of an ideal and explain their purpose in the computations of a secret key from a system of polynomial equations over GF(2). Most definitions listed here are from [5].

Definition 3.1. Let $I \subseteq k[x_1, ..., x_n]$ be an ideal (other than $\{0\}$) and let have some monomial order on $k[x_1, ..., x_n]$. Then:

(i) A set of leading terms of nonzero elements from I is denoted by $\mathrm{LT}(I)$, and is defined by

$$\mathrm{LT}(I) = \{cx^\alpha \mid \exists f \in I \setminus \{0\}, \mathrm{LT}(f) = cx^\alpha\}.$$

(ii) An ideal generated by $\mathrm{LT}(I)$ is denoted by $\langle \mathrm{LT}(I) \rangle$.

Theorem 3.2. *Let $I \subseteq k[x_1, ..., x_n]$ be an ideal other than $\{0\}$.*

(i) $\langle \mathrm{LT}(I) \rangle$ is a monomial ideal.

(ii) There are $g_1, ..., g_t \in k[x_1, ..., x_n]$, such that $\langle \mathrm{LT}(I) \rangle = \langle \mathrm{LT}(g_1), ..., \mathrm{LT}(g_t) \rangle$.

The proof of this theorem is given in [5, p. 77].

Theorem 3.3. ***Hilbert Basis Theorem.*** *Every ideal $I \subseteq k[x_1, ..., x_n]$ has a finite generating set, i.e., $I = \langle g_1, ..., g_t \rangle$ for some $g_1, ..., g_t \in I$.*

The proof of this theorem is given in [5, p. 77]. Recall that a basis is a set of polynomials in $k[x_1, ..., x_n]$ that generates an ideal I. Hilbert basis theorem then states that every ideal I in $k[x_1, ..., x_n]$ is generated by finitely many polynomials.

Definition 3.4. Let's have a monomial order on the polynomial ring $I \subseteq k[x_1, ..., x_n]$. A finite subset $G = \{g_1, ..., g_t\}$ of a nonzero ideal $I \subseteq k[x_1, ..., x_n]$ is a **Gröbner basis** if

$$\langle \mathrm{LT}(g_1), ..., \mathrm{LT}(g_t)\rangle = \langle \mathrm{LT}(I)\rangle.$$

This definition states that a set $\{g_1, ..., g_t\} \subseteq I$ is a Gröbner basis for I if and only if the leading term of any element of I is divisible by some of the $\mathrm{LT}(g_i)$.

Corollary 3.5. *Let's have a monomial order on $k[x_1, ..., x_n]$. Then every ideal $I \subseteq k[x_1, ..., x_n]$ has a Gröbner basis.*

The proof of this theorem is given in [5, p. 78].

Definition 3.6. Let $I \subseteq k[x_1, ..., x_n]$ be an ideal. Then $\mathbf{V}(I)$ denotes a set

$$\{(a_1, ..., a_n) \in k^n \mid \forall f \in I : f(a_1, ..., a_n) = 0\}.$$

Theorem 3.7. *$\boldsymbol{V}(I)$ is an affine variety. Specifically, if $I = \langle f_1, ..., f_s\rangle$ then $\boldsymbol{V}(I) = \boldsymbol{V}(f_1, ..., f_s)$.*

An important consequence of this theorem is that ideals determine varieties.

Definition 3.8. Let G be a Gröbner basis for polynomial ideal I. A **Reduced Gröbner basis** for I is a Gröbner basis for I such that:

(i) $\mathrm{LC}(p) = 1$ for all $p \in G$, where LC denotes leading coefficient.
(ii) For all $p \in G$, no monomial p belongs to $\langle \mathrm{LT}(G \setminus \{p\})\rangle$.

Any polynomial ideal has its reduced Gröbner basis, and this basis is unique. Several computer algebra systems, such as Magma [17], compute reduced Gröbner basis.

Definition 3.9. Let $\mathbb{F}_q[x_1, \dots, x_n]$ be a polynomial ring over the finite field $\mathbb{F}_q$, where $q = p^m$, p is a prime number and m is a positive integer. For every $i = 1, \dots, n$, the polynomials $x_i^q - x_i$ are called **field equations** of $\mathbb{F}_q$.

Theorem 3.10. ***Finiteness Theorem.*** *Let $f_1, \dots, f_m \in \mathbb{F}_q[x_1, \dots, x_n]$ be polynomials over $\mathbb{F}_q$. If we have $\langle f_1, \dots, f_m\rangle \cap \mathbb{F}_q[x_i] \neq 0$ for all x_i, then $V(\langle f_1, \dots, f_m\rangle) \subseteq \mathbb{F}_q^n$ is finite.*

According to the Finiteness Theorem, adding the field equations into a polynomial system extracted from a cipher ensures that the system will have finitely many solutions. If E is an extension of a finite field F, then all elements of the field F satisfy the field equations over F, but no element of the set $E \setminus F$ satisfies these equations.

Definition 3.11. Let $I = \langle f_1, \dots, f_m\rangle \subseteq \mathbb{F}[x_1, \dots, x_n]$ be an ideal. The l-th **elimination ideal** I_l is the ideal of $\mathbb{F}[x_{l+1}, \dots, x_n]$ given by $I_l = I \cap \mathbb{F}[x_{l+1}, \dots, x_n]$.

Theorem 3.12. ***The Elimination Theorem.*** *Let $G \subseteq I \subseteq \mathbb{F}[x_1, \dots, x_n]$ be a Gröbner basis of I so that $x_1 \succeq_{lex} x_2 \succeq_{lex} \cdots \succeq_{lex} x_n$, where $\succeq_{lex}$ is the lexicographic monomial order. Then, for every $0 \le l < n$, the set $G_l = G \cap \mathbb{F}[x_{l+1}, \dots, x_n]$ is a Gröbner basis of the l-th elimination ideal I_l.*

The Elimination Theorem provides a method for eliminating unknown variables in polynomials belonging to Gröbner basis. The eliminated system is considerably easier to solve, and the eliminated variables can be found using the substitution method.

In the experimental part of this work, we used the F4 algorithm [18], proposed by Faugère in 1999, to compute Gröbner bases.

3.2 Small Scale Variants of the AES

Due to the computational complexity of solving the problem of finding the secret key of the AES cipher, it was necessary to develop reduced variants of the AES, which would have similar algebraic properties to AES, but its analysis would take significantly less time and memory to analyze.

Carlos Cid et al. [4] proposed a whole family of such ciphers, which we will call *small scale variants of the AES*. Each of these variants can be characterized by four parameters *n, r, c,* and *e* which are defined as follows

- n - the number of rounds performed by the encryption algorithm, $1 \le n \le 10$
- r - number of rows r of the variable *state*, $r = 1, 2, 4$
- c - number of columns c of the variable *state*, $c = 1, 2, 4$
- e - the number of bits e indicating the size of the *word* (an element of the state), $e = 4, 8$

For the small scale variant of the AES, we adopted the notation $\mathrm{SR}(n, r, c, e)$ from [4].

The size of the variable *state* for the particular variants of the cipher can then be calculated from the parameters as $r{\cdot}c{\cdot}e$ bits, which are represented by an array of $r \cdot c$ words.

Note that, for the e parameter, the individual ciphers work with elements from the field $\mathrm{GF}(2^e)$, and hence, an irreducible polynomial must be defined for each e. The variants of $\mathrm{SR}(n,r,c,8)$ work with polynomials from the field defined as $\mathrm{GF}(2)[x] \setminus \langle x^8 + x^4 + x^3 + x + 1\rangle$. On the other hand, the variants $\mathrm{SR}(n,r,c,4)$ work with the irreducible polynomial $p(x) = x^4 + x + 1$, i.e., the field $\mathrm{GF}(2)[x] \setminus \langle x^4 + x + 1\rangle$.

For the original AES algorithm, each round consists of the *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey* functions, except for the last round where *MixColumns* is missing. The same holds for small scale variants of the $\mathrm{SR}^*(n,r,c,e)$ cipher, which use simplified variants.

The same simplified functions are also used by the $\mathrm{SR}(n,r,c,e)$ variants. However, the last round is not different from the previous ones and contains all functions, including *MixColumns*.

From the description of all the above parameters, it now follows that the full version of the AES algorithm corresponds with the $\mathrm{SR}^*(10,4,4,8)$ variant. Since, when encrypting the plaintext with the same secret key, the ciphertexts from the versions $\mathrm{SR}^*(n,r,c,e)$ and $\mathrm{SR}(n,r,c,e)$ can be converted to each other using an affine mapping, we can only use $\mathrm{SR}(n,r,c,e)$ versions in our work.

We will describe the differences in individual transformation functions for their simplified variants. The *SubBytes* function replaces individual bytes of the *state* variable using the *S-box* substitution table. For $e = 8$, the *SubBytes* function is identical to the one used for full AES.

For $e = 4$, the *SubBytes* function is created using the following transformations.

i) Take the multiplicative inversion in $\mathrm{GF}(2^4)$ (the element 0_{16} is mapped onto itself).
ii) Apply the following affine transformation over $\mathrm{GF}(2^4)$:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad (1)$$

The function *ShiftRows* works independently of the number of rows and columns so that we can define it in the same way as for the original AES. Each row is cyclically rotated to the left by the number of bytes determined by the index of the given row if we index from 0.

The function *MixColumns* for $c = 4$ works the same as for the original AES. For $c = 2$, the linear transformation using matrix multiplication is as follows:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \end{bmatrix} = \begin{bmatrix} 03 & 02 \\ 02 & 03 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \end{bmatrix} \text{ pro } 0 \le c < 2. \qquad (2)$$

Note that for $e = 4$ and $e = 8$, the respective field $\mathrm{GF}(2^e)$ and its irreducible polynomial are used.

The *AddRoundKey* function works the same for all variants. At the beginning of the algorithm and at each round, it simply xor the corresponding part of the expanded key to the *state* variable.

However, the difference between the variants occurs in the *KeyExpantion* function when creating an expanded key. The length of the original key itself is no longer fixed at 128 bits but is calculated from the specified parameters as $(r \cdot c) \cdot e$. The length of the expanded key is then calculated as $c \cdot (n+1) \cdot e$.

4 Polynomial Equations Deriving from AES

The system of equations for the AES cipher consists of two parts. One part is obtained from the transformations during the key expansion, and the other during the encryption. While the system for key expansion depends only on the variables of this secret key, part of the system for encryption already includes the variables of individual plaintexts and ciphertexts. As a result, we can generate a different polynomial system for each pair of plaintext and ciphertext, even when using the same secret key [19].

There are several ways in which the polynomial system of equations for the AES cipher can be constructed, which are described, for example, in [4], [19] or [20]. In this work, we use the construction of a system that is exhaustively described in our previous work [16].

Systems of equations for all small scale versions of AES are modeled in the same way. However, for the sake of illustration, we will provide in next two subsections a concrete example of the construction of a polynomial system corresponding to the $\mathrm{SR}(n,2,2,4)$ cipher.

4.1 Quadratic Equations

First, we will focus on the single non-linear transformation of the entire algorithm, which occurs when applying the reduced S-box table described in Section 3.2.

If we take the polynomial $b \in \mathrm{GF}(2^e)$ as the input and $c \in \mathrm{GF}(2^e)$ as the output of the S-box, we know that $bc = 1$, if $b \neq 0$. However, the probability of the situation $b = 0$ is minimal [19], and if it were to occur, we would avoid this problem by choosing other input data.

We can now define the first four equations that describe the result of the polynomial multiplication operation in $\mathrm{GF}(2)[x] \setminus \langle x^4+x+1\rangle$ for $bc = 1$, where the coefficients b_i and c_i are from $\mathrm{GF}(2)$

$$\begin{aligned}
1 &= b_0c_0 \oplus b_3c_1 \oplus b_2c_2 \oplus b_1c_3\\
0 &= b_1c_0 \oplus b_0c_1 \oplus b_3c_2 \oplus b_2c_3 \oplus b_3c_1 \oplus b_2c_2 \oplus b_1c_3\\
0 &= b_2c_0 \oplus b_1c_1 \oplus b_0c_2 \oplus b_3c_3 \oplus b_3c_2 \oplus b_2c_3\\
0 &= b_3c_0 \oplus b_2c_1 \oplus b_1c_2 \oplus b_0c_3 \oplus b_3c_3
\end{aligned} \tag{3}$$

For the variant $e = 8$, we would get a total of eight equations in a similar way.

The relation $bc = 1$ can also provide other equations. For example, we can multiply the entire formula by the polynomial b or c and thus get the relations $bc^2 = c$ and $b^2c = b$ and from them, the equations $bc^2 + c = 0$ and $b^2c + b$ =0.

To obtain quadratic equations from the relation $bc^2 + c = 0$, we must first calculate c^2. Since in (3) we have already received the result of the bc operation, we can now replace b_i with c_i in it and thus obtain the following coefficients

$$\begin{aligned}
c_0^2 &= c_0 \oplus c_2\\
c_1^2 &= c_2\\
c_2^2 &= c_1 \oplus c_3\\
c_3^2 &= c_3
\end{aligned} \tag{4}$$

We can then substitute these coefficients into the relation $bc^2 + c = 0$ and the result will be the following four quadratic equations

$$\begin{aligned}
0 &= b_0c_0 \oplus b_0c_2 \oplus b_3c_2 \oplus b_2c_1 \oplus b_2c_3 \oplus b_1c_3 \oplus c_1\\
0 &= b_1c_0 \oplus b_1c_2 \oplus b_0c_2 \oplus b_3c_1 \oplus b_3c_3 \oplus b_3c_2 \oplus b_2c_1\\
&\quad \oplus b_1c_3 \oplus c_1\\
0 &= b_2c_0 \oplus b_2c_2 \oplus b_1c_2 \oplus b_0c_1 \oplus b_0c_3 \oplus b_3c_1 \oplus b_2c_3\\
&\quad \oplus c_2\\
0 &= b_3c_0 \oplus b_3c_2 \oplus b_2c_2 \oplus b_1c_1 \oplus b_1c_3 \oplus b_0c_3 \oplus b_3c_3\\
&\quad \oplus c_3
\end{aligned}$$

Using the same procedure, we can obtain the equations for the relation $b^2c + b = 0$, and as a total, we obtain eight additional equations for our system. We can point out here that for the variant $e = 8$, we would get 16 equations this way.

In the next step, we can multiply the original formulas by a specific power of one of its variables. The most advantageous for us will be the third power since when using the square power, the result would contain cubic equations and not just quadratic ones.

We take the relation $bc^4 = c^3$ and adjust it again to $bc^4 + c^3 = 0$. We get the coefficients for c^4 as a result of multiplying c^2c^2, where c^2 is calculated in (4). Using substitution into the equations (3), then after all adjustments, we get the equations

$$\begin{aligned}
0 &= b_3c_3 \oplus b_3c_1 \oplus b_2c_3 \oplus b_2c_2 \oplus b_1c_3 \oplus b_0c_3 \oplus b_0c_2\\
&\quad \oplus b_0c_1 \oplus b_0c_0 \oplus c_3c_1 \oplus c_2c_1 \oplus c_2c_0 \oplus c_0\\
0 &= b_3c_2 \oplus b_3c_1 \oplus b_2c_2 \oplus b_1c_2 \oplus b_1c_1 \oplus b_1c_0 \oplus b_0c_3\\
&\quad \oplus b_0c_1 \oplus c_3c_2 \oplus c_2c_0 \oplus c_1c_0 \oplus c_3\\
0 &= b_3c_2 \oplus b_2c_2 \oplus b_2c_1 \oplus b_2c_0 \oplus b_1c_3 \oplus b_1c_1 \oplus b_0c_3\\
&\quad \oplus b_0c_2 \oplus c_3c_2 \oplus c_3c_1 \oplus c_3c_0 \oplus c_2c_1 \oplus c_2c_0\\
&\quad \oplus c_1c_0 \oplus c_2\\
0 &= b_3c_2 \oplus b_3c_1 \oplus b_3c_0 \oplus b_2c_3 \oplus b_2c_1 \oplus b_1c_3 \oplus b_1c_2\\
&\quad \oplus b_0c_3 \oplus c_3c_2 \oplus c_3c_2 \oplus c_3c_1 \oplus c_3 \oplus c_2 \oplus c_1
\end{aligned}$$

Using the same procedure, we also get the equations for the formula $b^4c+b^3 = 0$, and in total, we get eight more equations for our system. We end up with twenty quadratic equations for the variant $e = 4$ and forty for the variant $e = 8$. As stated in [19], we will not need all 20 equations for our system, but the first twelve will suffice.

4.2 Linear Equations

We will construct the next equations only from linear transformations, and hence, these equations will also be linear. We begin by introducing the equations related to the S-box table.

These equations are based on the second step of the reduced S-box table construction given in

Section 3.2 and can be directly expressed using the expression from (1). As an input to this expression, we take the polynomial $c \in \mathrm{GF}(2^e)$, which is the output of the transformation described in Section 4.1. If we combine the four linear equations created in this way with the twelve quadratic equations from the previous section, we get sixteen equations that accurately describe the table for the S-box.

Next, we will focus on modeling the individual transformations of the entire AES algorithm. For simplicity, we work with data that describes the entire two-dimensional array of the *state* variable.

For the function *SubBytes*, we introduce the matrix L, which is based on the matrix used to produce the S-box in the expression (1), as follows

$$L_b = \begin{pmatrix} 1&0&1&1\\ 1&1&0&1\\ 1&1&1&0\\ 0&1&1&1 \end{pmatrix} \longrightarrow L = \begin{pmatrix} L_b&0&0&0\\ 0&L_b&0&0\\ 0&0&L_b&0\\ 0&0&0&L_b \end{pmatrix}$$

At the same time, we also expand the constant vector $(0,1,1,0)^T = 6_{16}$ from (1) to $\mathbf{v} = (6_{16}, 6_{16}, 6_{16}, 6_{16})$ so that we can work with the entire *state* and not only with individual bytes. Furthermore, we denote the input vector of the function *SubBytes* as $\mathbf{b}$ and the output vector as $\mathbf{b^{-1}}$ (since it contains inverse elements from $\mathbf{b}$). Each component of these vectors consists of four coefficients of the b and c polynomials defined in Section 4.1, and hence each element brings twelve quadratic equations to our system.

For the *ShiftRows* function, we will then introduce the matrix R, which will represent the rotation of individual rows

$$R = \begin{pmatrix} I_4&0&0&0\\ 0&0&0&I_4\\ 0&0&I_4&0\\ 0&I_4&0&0 \end{pmatrix}$$

where I_4 is the identity matrix of size 4×4.

To illustrate the *MixColumns* function, we first rewrite the polynomial multiplication operation bc from (3) into the matrix form

$$\begin{pmatrix} b_0 & b_3 & b_2 & b_1\\ b_1 & b_0 \oplus b_3 & b_3 \oplus b_2 & b_2 \oplus b_1\\ b_2 & b_1 & b_0 \oplus b_3 & b_3 \oplus b_2\\ b_3 & b_2 & b_1 & b_0 \oplus b_3 \end{pmatrix} \begin{pmatrix} c_0\\ c_1\\ c_2\\ c_3 \end{pmatrix} \tag{5}$$

If we now substitute the binary values of coefficients 03_{16} and 02_{16} from (2) for the individual bits b_i in the matrix from (5), we get the following two matrices

$$M_{03} = \begin{pmatrix} 1&0&0&1\\ 1&1&0&1\\ 0&1&1&0\\ 0&0&1&1 \end{pmatrix} \text{ a } M_{02} = \begin{pmatrix} 0&0&0&1\\ 1&0&0&1\\ 0&1&0&0\\ 0&0&1&0 \end{pmatrix}$$

The matrix M that describes the entire *MixColumns* function is then

$$M = \begin{pmatrix} M_{03}&M_{02}&0&0\\ M_{02}&M_{03}&0&0\\ 0&0&M_{03}&M_{02}\\ 0&0&M_{02}&M_{03} \end{pmatrix}$$

We can now express one round $0 < i \leq n$ of the $\mathrm{SR}(n,2,2,4)$ cipher using this expression:

$$\mathbf{b_i} = MRL\mathbf{b_{i-1}^{-1}} + \mathbf{k_i} + \mathbf{v},$$

where $\mathbf{k_i}$ is a vector containing 16 bits representing variables in the given part of the expanded key for round i.

From this expression, we get sixteen linear equations for each round. In addition, we add twelve quadratic equations for each element from $\mathbf{b_{i-1}^{-1}}$, so that in total, we have sixty-four equations in our system for each round of the algorithm.

In the beginning, we get $\mathbf{b_0}$ by taking the plaintext values and adding to them the variables of the initialization part of the expanded key, which are directly the variables of the secret key. As a result, sixteen more equations are added to the system before the first round.

4.3 Key Expansion Equations

Let $k_i = (k_{i,0}, k_{i,1}, k_{i,2}, k_{i,3})^T \in \mathrm{GF}(2^4)^4$ denote the given part of the secret key for round i. According to [4], this part is then defined for each round i as:

$$\begin{aligned} \begin{pmatrix} k_{i,2q}\\ k_{i,2q+1} \end{pmatrix} &= \begin{pmatrix} L(s_0)\\ L(s_1) \end{pmatrix} + \begin{pmatrix} 6_{16}\\ 6_{16} \end{pmatrix} + \begin{pmatrix} \kappa_i\\ 0 \end{pmatrix} \\ &+ \sum_{t=0}^{q} \begin{pmatrix} k_{i-1,2t}\\ k_{i-1,2t+1} \end{pmatrix} \end{aligned} \tag{6}$$

for $0 \leq q < 2$, where $s_0 = k_{i-1,3}^{-1}, s_1 = k_{i-1,2}^{-1}$, and κ_i is the round constant from $GF(2^4)$. From this expression, we get sixteen linear equations for each round.

When generating parts of the key for each round, the S-box is always applied twice. For each round, we add another $2 \cdot 12$ quadratic equations to our system, which we defined in Section 4.1. Thus, for each round, we add a total of forty linear and quadratic equations to our system that describe the expanded secret key.

5 Reducing Overdefined Systems

This section presents techniques based on data mining used for transforming the original systems of polynomial equations into a new system for which the F4 algorithm can calculate the Groebner base faster than the direct application of F4 to the original system. This kind of attack is a known plaintext attack where we assume that we have several different plaintexts and corresponding ciphertexts. For each plaintext-ciphertext pair, we obtain as many polynomials as the number of secret key bits, and as a result, we create an overdefined system of polynomial equations. We will then reduce the system using various data mining algorithms to accelerate the calculation of the Gröbner base and, as a result, the secret key.

More precisely, the problem to be solved via data mining methods is defined as follows:

Let P be a system of m polynomial equations $\{p_i(k_1, \ldots, k_n) = 0 \mid i = 1, \ldots, m\}$ over GF(2), where m depends on the number of plaintext-ciphertext pairs (denoted by *num*) according to the formula $m = num \cdot n$. The secret key $K = (k_1, \ldots, k_n)$ is a solution of P. The goal is to transform the original system P to a new system of polynomial equations, the system $Q = \{q_j(k_1, \ldots, k_n) = 0 \mid j = 1, \ldots, m'\}$ such that:

- K belongs to an affine variety $V(J)$, where J is an ideal generated by m' polynomials from Q, where $m \geq m' \geq n$,
- the F4 algorithm computes the Gröbner basis faster for the preprocessed system Q than for the original system P.

To deals with this problem, we suggest the following approach:

1. group the polynomials $p_i, i = 1, \ldots, m$, to m' clusters according some similarity function,
2. for each cluster $j, j = 1, \ldots, m'$, find two the most similar polynomials, and sum them to obtain the polynomials $q_j(k_1, \ldots, k_n)$,
3. run the F4 algorithm for the system Q to compute Gröbner basis.

Let I be an ideal generated by the polynomials of the system P, i.e., $I = \langle p_1, \ldots, p_m \rangle$. It is trivial to prove that $V(I) \subseteq V(J)$, and hence the secret key belongs also to $V(J)$. Experimental results in Section 6 demonstrate that the F4 algorithm computes the Gröbner basis faster for the preprocessed system Q than for the original system P. We considered only the secret key calculation method based on the Gröbner base, which is calculated using the F4 algorithm implemented in the software Magma [17].

In the rest of this section, we present three techniques for reducing the original system of polynomial equations: Partitioning Around Medoids (PAM), Locality-Sensitive Hashing (LSH), and Removing the Most Significant Monomial (RMSM), where PAM and LSH are data mining methods. We implemented each of these techniques and provided implementation details.

5.1 Cluster Analysis

We first explored the application of cluster analysis, or clustering [21]. Its principle consists of dividing relevant data into groups so that in each group, there are data similar to each other with respect to some similarity function. Since we work with polynomials over GF(2), we used a similarity function based on the Hamming distance.

In our case, the relevant data consists of all individual polynomials, and the Hamming distance is defined as the number of different monomials of the given two polynomials. Since we work in GF(2), the Hamming distance can be calculated as the number of monomials of the polynomial formed by the mutual xor of the given two polynomials.

We will show that such Hamming distance modified for working with polynomials satisfies all the axioms of the distance definition:

i) $d(p, q) = 0$ if and only if $p = q$: Since the polynomials are defined in GF(2), all monomials contained in both p and q polynomials cancel each other during xoration. As a result, only the different monomials remain. Therefore, for the resulting polynomial to be zero, p must contain the same monomials as q and hence, $p = q$.

ii) $d(p, q) = d(q, p)$: It follows directly from the definition of the xor operation that it is commutative. [22]

iii) $d(p, q) \leq d(p, r) + d(r, q)$: Proof of this property follows from the definition of the xor operation. It is stated in [21] that this axiom does not have to be fulfilled for the purposes of clustering.

5.1.1 Partitioning Around Medoids

As a suitable clustering algorithm for our data, we chose the Partitioning Around Medoids (PAM) algorithm [23] since it is based on the search for k representative centers, so-called *medoids*, which belong to a given data set. In other words, clusters (groups of data) are formed around medoids in this case, which are always selected only from the specified set. This differs from the more frequently used *K-means* algorithm, which also divides the data into groups around certain centers called centroids. However, these centroids are calculated as the optimal centers of all data of the given group and may not occur in the ideal generated by the polynomials from the system P.

Another reason why we used the PAM algorithm is that it allows clustering with respect to any distance metric, i.e., it is not limited to an Euclidean distance, which is not suitable for GF(2). The pseudocode of the PAM algorithm is described in Algorithm 1.

Algorithm 1 PAM algorithm

```
Input: Number of clusters $k$, set of data points $P$
Output: $k$ clusters
 1: Initialize: randomly select $k$ data points from $P$ to become the medoids
 2: Assign each data point to its closest medoid
 3: for all cluster do
 4:     identify the observation that would yield the lowest average distance if it were to be re-assigned as the medoid
 5:     if the observation is not current medoid then
 6:         make this observation the new medoid
 7:     end if
 8: end for
 9: if at least one medoid has changed then
10:     go to step 2
11: else
12:     end the algorithm.
13: end if
```

At the beginning of the algorithm, the medoids are chosen randomly, and then the distance of all elements from the set P to each of the medoids is calculated. Individual elements are then assigned to the closest medoids. Then the so-called *total cost* of this assignment is calculated, equal to the sum of the distances of all data to their respective medoids.

The algorithm works in a cycle and executes the following commands for each medoid. For each element that is assigned to the closest medoid, it recalculates the distances as if it were its center and then calculates a new total price. If the total cost is lower than the previous one, the given medoid is updated to this element.

The for cycle in step three continues as long as the medoids are updated. And when it runs out, we get the required number of clusters around the respective medoids, which terminates the PAM algorithm.

In our implementation, the calculation continues since the required output of this polynomial processing method is not entire clusters but only one representative from each of them. Instead of choosing one random polynomial, we optimized the resulting system as follows.

We select two polynomials from each cluster that are most similar to each other and then take the result of their mutual xor as a representative of this cluster. As a result, the selected polynomials will be shorter (in terms of number of monomials) than all the originally generated polynomials.

5.2 Locality-Sensitive Hashing

As another type of polynomial system processing, we experimented with a group of algorithms called *Locality Sensitive Hashing* (LSH) [25]. The goal of LSH is for a given element p to find such $x \in \{x_1, ..., x_n\}$ for which $d(p, x)$ is the smallest of all $\{x_1, \ldots, x_n\}$, where $d(p, x)$ is their mutual distance. There are effective algorithms for solving this problem for data with a lower number of dimensions. However, since we work in a multidimensional space, we used *approximate nearest neighbor* search algorithms [24]. In our case of searching for the most similar polynomials, we defined the distance d between two polynomials as the number of monomials of their xor.

If we find the most similar polynomials from the entire generated set, then by xoring them, we will again obtain simpler polynomials (in terms of a number of monomials) which we can use to calculate Gröbner bases. Our implementation of LSH consists of the following parts: Shinling, MinHashing, and Bucketing.

5.2.1 Shinling

Initially, it is necessary to create sets of all monomials that individual polynomials contain. A superset is constructed from these sets, which includes monomials from all polynomials. A dictionary is then generated from this superset, where a number is assigned for each monomial. We then convert individual polynomials into so-called *sparse binary vectors*. Each element of this vector now represents exactly one monomial from the dictionary of all monomials. If the given polynomial contains a certain monomial, one is assigned at the corresponding place in the binary vector; otherwise, there is zero.

5.2.2 MinHashing

In the next part, the first hashing process converts sparse vectors into dense ones, which we call *signatures*. These signatures are of the same size for all polynomials. The size depends on the size of the dictionary, but it is smaller in order of magnitude.

For each polynomial, one signature is created such that a random permutation of dictionary-sized numbers is created for each element of this signature. In that permutation, the ordinal place where the number 1 is located is then searched, and it is determined whether 1 is located in the same place in the corresponding sparse vector. If so, the number 1 from the permutation is stored as one element of the signature. Otherwise, it is searched at which place in the permutation the number 2 is, and analogically, this place is compared with the sparse vector. If there is a 1, it is written as signature element number 2, and if not, this procedure is repeated for 3, 4, and so on.

5.2.3 Bucketing

In the last step of the algorithm, similar signatures are searched for, which will determine the so-called *pairs of candidates* for similar polynomials. However, if we were looking for the similarity of the entire signatures, it would mean that we are only looking for almost identical polynomials, and very few candidates would be found.

Therefore, the algorithm first divides the signatures into several subparts, which are hashed again, each one separately. Then, if the same hash appears for two different polynomials, at least for some part, these two polynomials are marked as candidates for a similar pair.

For each pair of candidates found, their mutual distance and the polynomial formed by their xor are then calculated. These pairs of candidate polynomials are then sorted by their mutual distance from smallest to largest, and the corresponding number of the smallest ones is then used to calculate the Gröbner basis.

5.3 Algorithm for Removing the Most Significant Monomial

As the last type of polynomial system processing, we experimented with a method in which, from a large number of generated polynomials, we select for the calculation of the Gröbner basis precisely those whose leading monomial, using the inverted graded lexicographic order, has the smallest possible degree. With such polynomials, the F4 algorithm works faster since in the *ComputeM* function (described in chapter 10 of [5]), all monomials from all polynomials are processed in a cycle from the largest to the smallest. If we take polynomials with the smallest possible leading monomials, fewer of them would be processed. This strategy focusing on leading terms was mentioned in [26].

If two different polynomials have the same leading monomial, we can sum them, and the resulting polynomial will have an even smaller degree of the leading monomial, making it more convenient to calculate the Gröbner basis.

In the rest of this section, we provide the implementation details. First, we find the leading monomial and calculate its degree for each polynomial. Our scripts implement polynomials using structures from the *PolyBoRi* package [27]. The functions *lead()* and *deg()* are directly implemented here, which provide this functionality.

The problem is, however, that these functions operate with monomials ordered using lexicographic ordering, whereas, for the F4 algorithm, it is more advantageous to order monomials in an inverted graded lexicographic order, as recommended in [18]. It is, therefore, necessary to find the leading monomial in this order.

All polynomials are arranged in an array according to the degree of their leading monomial from the smallest to the largest. Another polynomial with the same leading monomial as a given polynomial is successively searched for all polynomials. When such a pair is found, its xor is selected into the set for calculating the Gröbner basis, and both polynomials are discarded from the original pair search array. This procedure is repeated until we get the required number of polynomials to calculate the Gröbner basis.

6 Experimental Results

All experiments were performed on the machine with the operating system *Ubuntu 20.04.4 LTS* running on two *Intel Xeon Gold 6136* processors containing 12 cores each. In total, the machine has 24 cores and 768 GB of RAM.

The following experiments deal with reducing overdefined systems of polynomial equations derived from small scale variants of the AES. For each plaintext-ciphertext pair, we get one subsystem with as many equations as unknown bits of the secret key. By uniting these subsystems, we get an overdefined system to which we gradually apply the three algorithms described in Section 5 to reduce the system. For such a reduced system, we compute the Gröbner basis using the F4 algorithm to obtain the secret key. We compared the computation times of this approach with the standard method, i.e., direct application of the F4 algorithm without reducing the equations.

For each of the selected simplified variants of the AES cipher, we experimented with the parameter *num*, which indicates the number of plaintext-ciphertext pairs from which individual polynomial subsystems are generated. Since the secret key computation duration depends on this input data, we repeated each measurement ten to twenty times for different data sets (i.e., systems of polynomial equations). We computed the average values and the deviations from the computational times from all the experiments and present them in this section.

For each cipher, we then generated graphs of the average time of polynomials generation and secret key calculation depending on the value of the *num* parameter. We selected the best result from each combination and present them for each cipher in a table with the following records

- Cipher - Selected variant of the AES cipher
- KB - Number of key bits
- *num* - Number of generated polynomial subsystems
- POL - Number of polynomials chosen for computation of the Gröbner basis
- MON - Average number of monomials in polynomials
- $\text{TIME}_{\text{PREP}}$ - Average preprocessing time of polynomials
- TIME_{KEY} - Average time to compute the secret key
- TIME_{ALL} - Average time of the entire computation, i.e., $\text{TIME}_{\text{PREP}}+\text{TIME}_{\text{KEY}}$
- MEM - Average used memory [MB]

The computational times are given in seconds. The missing values were caused by the computation time exceeding the maximum set limit (four hours) or some internal error in Magma scripts that occurred during the calculation.

6.1 Reference Solution

This section presents the experiments for the so-called *reference solution*, defined as follows. We generated random data consisting of the binary values of the plaintexts and the secret key, from which the ciphertext and the required polynomials were computed. We measured how long it takes to compute the secret key for different numbers of generated polynomial subsystems (i.e., for different *num* values) without using any polynomial system reduction.

As was demonstrated in our previous work [16], the key computation time decreased when we considered more polynomials than the secret key

Table 1: Measured runtimes without using reduction - the reference solution.

Cipher	KB $[bits]$	num	POL	MON	$\text{TIME}_{\text{PREP}}$ $[s]$	TIME_{KEY} $[s]$	TIME_{ALL} $[s]$	MEM $[MB]$
SR(1,2,2,4)	16	2	32	20	< 1	< 1	1	33
SR(2,2,2,4)	16	2	32	2451	2	3	5	68
SR(3,2,2,4)	16	2	32	32762	7	793	800	24464
SR(1,4,2,4)	32	2	64	36	1	< 1	1	33
SR(2,4,2,4)	32	2	64	33142	5	—	—	—
SR(1,2,4,4)	32	2	64	23	1	< 1	1	33
SR(2,2,4,4)	32	2	64	6689	3	—	—	—
SR(1,4,4,4)	64	2	128	40	3	< 1	3	33
SR(1,2,2,8)	32	2	64	316	7	< 1	7	33
SR(1,4,2,8)	64	2	128	566	14	2	16	33
SR(1,2,4,8)	64	4	256	347	14	3	17	33
SR(1,4,4,8)	128	8	1024	598	39	—	—	—

bits. Therefore, we started our experiments from $num = 2$.

Since there is no reduction of polynomials, the more we generate, the longer the total computation time since more polynomials must be processed. Table 1 lists the results for each variant of the cipher whose cryptanalysis took an acceptable time (i.e., less than four hours).

Table 1 shows that the higher the number of rounds, the more monomials in polynomials. However, this only applies until the third round since the number of monomials does not increase from the fourth round, which aligns with our previous work [16]. Therefore, we did not conduct experiments for ciphers with four or more rounds.

For example, for the SR(2,2,2,4) cipher, we present the following graphs based on Table 1. In Fig. 1, we can see the relation between the number of generated polynomial subsystems (i.e., num) and the following average computational times: the preparation time (i.e., the time of extraction of polynomials from the cipher), the computation of the secret key, and the total time, which is a sum of the previous two times. Fig. 2 shows the relation between the number of polynomial systems and the average number of monomials for the given polynomials.

6.2 PAM

This section presents the results of the reduction method based on clustering using the PAM algorithm. As input data, we used the same plaintexts

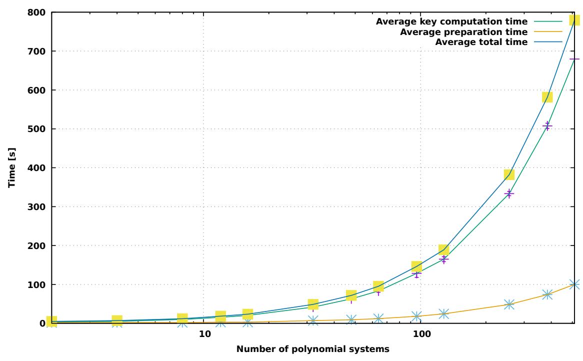

Fig. 1: The graph of the average computation time for different num for SR(2,2,2,4) in the reference solution.

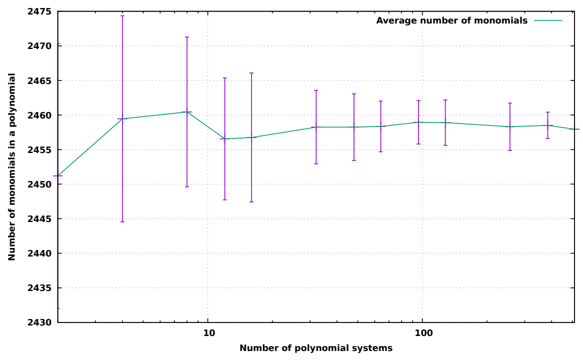

Fig. 2: The graph of the average number of monomials in polynomials for different num for SR(2,2,2,4) in the reference solution.

Table 2: Measured runtimes for the best configurations using the PAM method.

Cipher	KB $[bits]$	num	POL	MON	TIME$_{\text{PREP}}$ $[s]$	TIME$_{\text{KEY}}$ $[s]$	TIME$_{\text{ALL}}$ $[s]$	MEM $[MB]$
SR(1,2,2,4)	16	8	128	19	1	< 1	1	33
SR(2,2,2,4)	16	8	64	1847	12	2	14	73
SR(3,2,2,4)	16	2	16	32598	24	647	671	28465
SR(1,4,2,4)	32	8	256	14	2	< 1	2	33
SR(2,4,2,4)	32	2	32	29207	58	—	—	—
SR(1,2,4,4)	32	8	256	23	2	< 1	2	33
SR(2,2,4,4)	32	48	32	1148	5018	26	5044	406
SR(1,4,4,4)	64	8	512	40	11	< 1	11	33
SR(1,2,2,8)	32	8	256	315	17	< 1	17	50
SR(1,4,2,8)	64	8	512	566	99	4	103	245
SR(1,2,4,8)	64	8	512	364	58	46	104	720
SR(1,4,4,8)	128	8	256	412	438	7	445	215

and secret keys as were generated for the reference solution.

In Table 2, we present the best results from all experiments performed for this reduction method. Note that for all reduced variants of the cipher AES, we considered the following POL values: 1×KB, 2×KB, 4×KB, and 8×KB. For all reduction methods presented in this work, the corresponding tables, i.e., Table 2 for PAM, Table 3 for LSH, and Table 4 for RMSM, show only the fastest calculation times obtained from these four values of the POL parameter. We experimented with values of the parameter num from the set $\{2, 4, 8, 12, 16, 32, 48, 96, 128\}$.We started the experiments with $num = 2$ and the number of secret key bits equal to the number of polynomials from which the Gröbner basis was calculated. If the Gröbner basis could not be calculated, we increased the value of the num parameter. This is how we proceeded also with the experiments for LSH and RMSM.

For example, for SR(1,2,2,4), Fig. 3 demonstrates the relation between the number of polynomial subsystems that were reduced using the PAM algorithm to the one system and the average computational time of the entire computation of the secret key. The figure shows this relation for different values of the parameter POL. The option POL = 1×KB takes the highest computation time, while the other options have significantly better results. Note that the number of polynomials for computation of the Gröbner basis corresponds to the number of clusters created by the PAM algorithm.

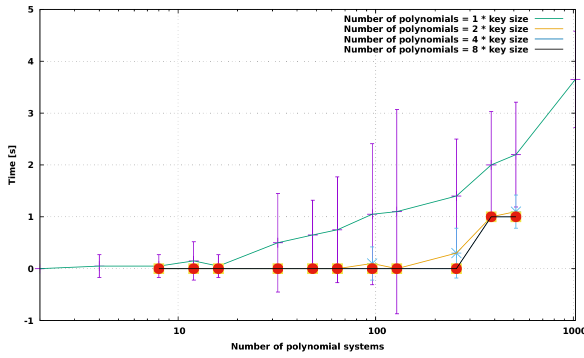

Fig. 3: The graph of the average key computation time for different num for SR(1,2,2,4) using the PAM method.

For the SR(2,2,2,4) and SR(3,2,2,4) ciphers, we did not achieve the best results with the option where we have the most polynomials for computing the Gröbner basis (i.e., POL=8×KB), as is the case with the other ciphers. For the SR(2,2,2,4), the best results were obtained if we chose four times the number of key bits, and for the SR(3,2,2,4) variant, we only need the same number of polynomials as the key has bits.

In addition, the cipher SR(2,2,2,4) is the only one where the time to compute the secret key decreases for the increasing parameter num. This dependence is presented in Fig. 4. We broke the

Table 3: Measured runtimes for the best configurations using the LSH method.

Cipher	KB [$bits$]	num	POL	MON	TIME$_{\text{PREP}}$ [s]	TIME$_{\text{KEY}}$ [s]	TIME$_{\text{ALL}}$ [s]	MEM [MB]
SR(1,2,2,4)	16	8	128	6	1	< 1	1	33
SR(2,2,2,4)	16	12	128	555	3	< 1	3	47
SR(3,2,2,4)	16	16	16	32335	174	589	763	24553
SR(1,4,2,4)	32	8	256	14	2	< 1	2	33
SR(2,4,2,4)	32	64	128	10240	2543	4490	7033	249128
SR(1,2,4,4)	32	8	256	6	2	< 1	2	36
SR(2,2,4,4)	32	8	64	2056	24	8	32	461
SR(1,4,4,4)	64	8	512	13	5	< 1	5	33
SR(1,2,2,8)	32	8	256	200	9	< 1	9	144
SR(1,4,2,8)	64	8	512	405	20	14	34	5199
SR(1,2,4,8)	64	8	512	195	17	1	18	434
SR(1,4,4,8)	128	48	512	288	144	3	147	5634

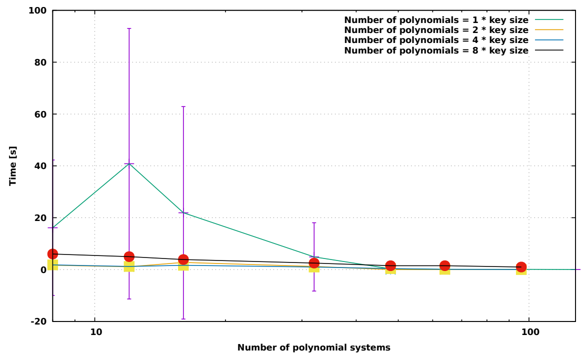

Fig. 4: The graph of the average key computation time for different num for SR(2,2,2,4) using the PAM method.

SR(2,2,4,4) cipher only once out of all attempts, where 48 subsystems were reduced. The resulting calculation is shown in Table 2.

Using the PAM method, we found the secret key even for the first round of the full AES cipher, i.e., SR(1,4,4,8). However, the reduction takes a relatively long time in this case, so only the computation for a maximum of sixteen generated polynomial systems (num=16) reached the set limit (three hours). In this case, the computation of the key took only four seconds. Although for the variant num=8, the key computation took three seconds longer, and the reduction was faster by half an hour.

6.3 LSH

This section presents the experimental results of the reduction method based on the LSH algorithm. We used the same plaintexts and secret keys as were generated for the reference solution. We performed four sets of experiments differing in the number of polynomials used to compute the Gröbner basis. We gradually set this number to one times the number of key bits, then two times, four times, and finally eight times.

In Table 3, we present the best results from all sets of experiments performed for the LSH reduction method. For most ciphers, we observed that the more polynomials chosen to calculate the Gröbner basis, the faster the basis was calculated. Hence, for most ciphers, the best results were achieved for the number of chosen polynomials corresponding to eight times the key bits.

For simpler ciphers, the LSH method was able to find such similar polynomials at higher num that the resulting xors of polynomials contained even less than three monomials, as can be seen in Fig. 5. For the SR(3,2,2,4) cipher, the computation of the Gröbner basis did not end at all with $num > 1$, so in Table 3, we only present the result where the number of polynomials equals number of key bits.

Using the LSH-based method, like using the PAM-based method, we found a secret key for the first round of the full version of the AES cipher. Among all sets of experiments, the secret key was

Table 4: Measured runtimes for the best configurations using the RMSM method.

Cipher	KB $[bits]$	num	POL	MON	TIME$_{\text{PREP}}$ $[s]$	TIME$_{\text{KEY}}$ $[s]$	TIME$_{\text{ALL}}$ $[s]$	MEM $[MB]$
SR(1,2,2,4)	16	2	16	22	< 1	< 1	1	33
SR(2,2,2,4)	16	4	16	771	2	6	8	376
SR(3,2,2,4)	16	8	32	32777	62	514	576	22700
SR(1,4,2,4)	32	2	32	35	1	< 1	2	35
SR(2,4,2,4)	32	96	32	32133	185	—	—	
SR(1,2,4,4)	32	2	32	26	1	< 1	2	33
SR(2,2,4,4)	32	8	256	6141	494	154	648	13024
SR(1,4,4,4)	64	8	128	41	14	< 1	15	33
SR(1,2,2,8)	32	8	256	326	12	< 1	12	104
SR(1,4,2,8)	64	8	64	579	< 1	—	—	
SR(1,2,4,8)	64	16	512	389	138	27	165	102153
SR(1,4,4,8)	128	96	256	634	11	—	—	

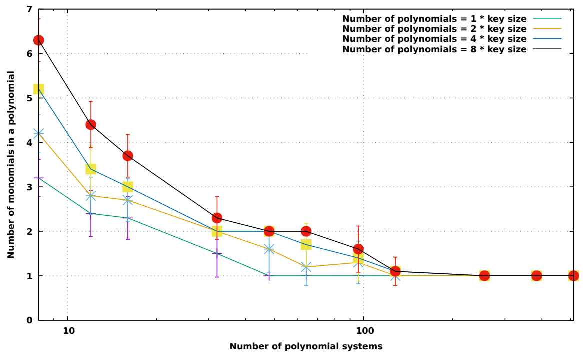

Fig. 5: The graph of the average number of monomials in polynomials for different num for SR(1,2,2,4) using the LSH method.

computed the fastest when 4·128 polynomials were chosen to compute the Gröbner basis.

Our experimental results indicate that we achieve better results for higher values of the num parameter. The reason is that the techniques for reducing overdefined systems of polynomial equations process the input set of polynomials and produce simpler polynomials (concerning the number of monomials) that are more suitable for computing Gröbner basis than the original, i.e., not reduced polynomials.

6.4 Removing the Most Significant Monomial

The last reduction method we considered in the experiments is Removing the Most Significant Monomial (RMSM). We used the same plaintexts and secret keys as were generated for the reference solution. In Table 4, we present the best results from all sets of experiments performed for the RMSM reduction method.

For SR(1,2,2,4), the results do not change as the number of polynomials increases. The preprocessing of polynomials for num up to 512 takes up to under a minute, and the key is always found within one second. For SR(2,2,2,4), there was a significant change in the average number of monomials per polynomial between two and four systems. Approximately 2000 polynomials have been reduced to just 771 in this case.

For SR(1,4,2,4), there is a significant increase in polynomial reduction time, but the key computation still fits within one second. Figures 6 and 7 show the time for preparing the polynomials, calculating the secret key, and the average number of monomials for the given polynomials for SR(1,4,2,4).

6.5 Comparison of Reduction Methods

When comparing all tested methods of reducing a system of polynomials, we first consider only the secret key computation time. For small values of the num parameter, the reduction methods did not show faster calculation times for finding the secret key than the reference solutions, which is demonstrated in Tables 1 to 4. However, at higher values of the num parameter, i.e., if we

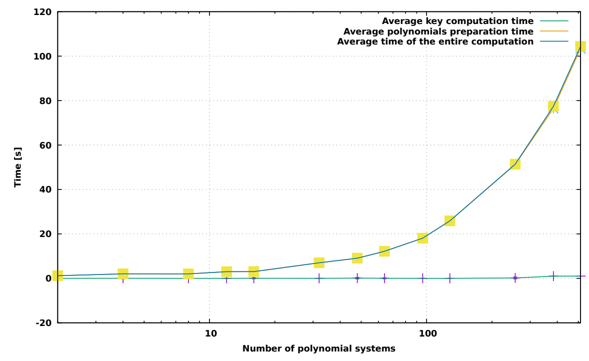

Fig. 6: The graph of the average computation time for different *num* for SR(1,4,2,4) using the method of removing the most significant monomial.

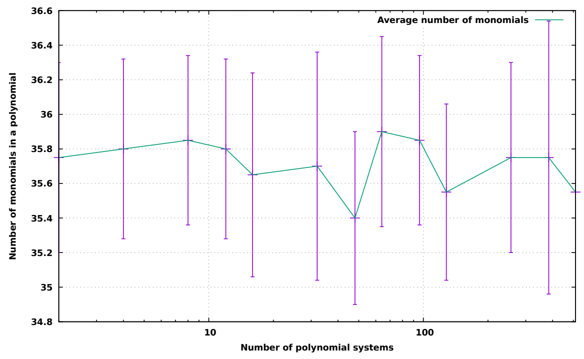

Fig. 7: The graph of the average number of monomials in polynomials for different *num* for SR(1,4,2,4) using the method of removing the most significant monomial.

use more equations for preprocessing, the reduction methods achieved a significantly shorter time for calculating the key. This is demonstrated for SR(1,4,2,4) in Fig. 8, where all reduction methods significantly outperformed the reference solution. According to this figure, the computation of the secret key from the already reduced polynomials takes almost the same short time for the PAM and LSH methods.

The following experiment focuses on the running time of the entire script, i.e., both the computation of the secret key and the time for generating and reducing all the polynomials. Figure 9 for SR(1,4,2,4) shows that the reduction of a large number of polynomials took the longest when clustering using the PAM algorithm. On the contrary,

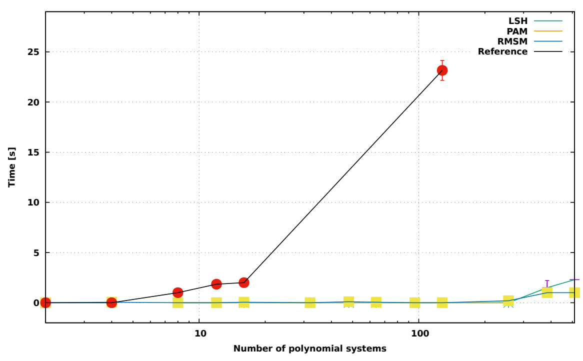

Fig. 8: The graph of the average key computation time for different *num* for SR(1,4,2,4) - comparison of all four methods.

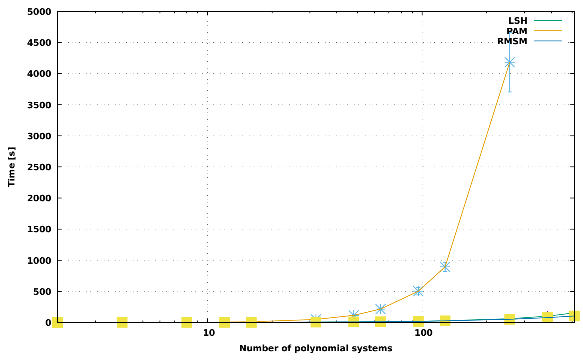

Fig. 9: The graph of the average time of the entire computation for different *num* for SR(1,4,2,4) - comparison of all four methods.

when using the LSH or the RMSM, the reduction took a relatively short time, even for a large number of polynomials. However, the PAM method is still more efficient than the RMSM when focusing only on a small number of polynomials, as can be seen, for example, in Fig. 10. In the figures 9 and 10, we can see that the entire computation time, in contrast to the PAM method, fluctuates for the RMSM.

The following comparison focuses on the SR(1,4,4,8) cipher, i.e., the full version of AES with only one round. Note that we only broke this cipher using PAM and LSH.

Figure 11 demonstrates for SR(1,4,4,8) that using the PAM method, the computation of the secret key took a shorter time than using LSH for the same *num* parameter. On the other hand, reduction using PAM took an order of magnitude

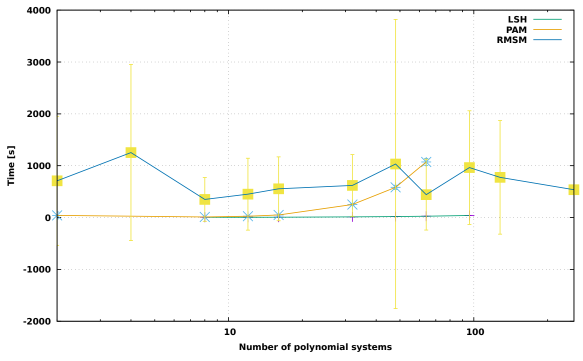

Fig. 10: The graph of the average time of the entire computation for different *num* for SR(1,4,4,4) - comparison of all four methods.

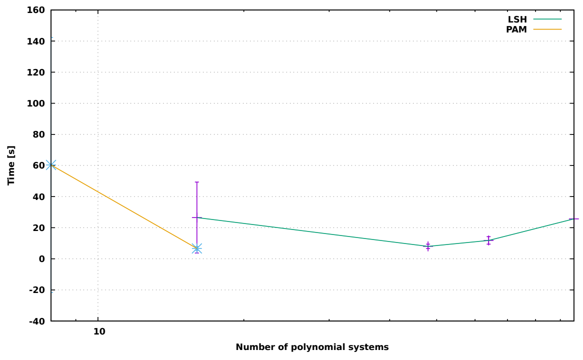

Fig. 11: The graph of the average key computation for different *num* for SR(1,4,4,8) - comparison of PAM and LSH methods.

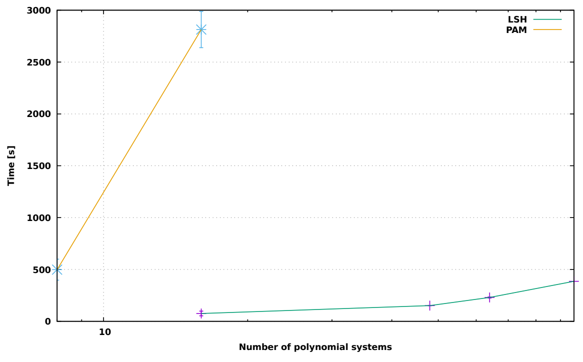

Fig. 12: The graph of the average time of the entire computation for different *num* for SR(1,4,4,8) - comparison of PAM and LSH methods.

longer than using LSH, even for a small number of generated polynomials. Hence, the entire computation took significantly longer for the PAM method, as shown in Fig. 12 for SR (1,4,4,8). Therefore, we consider the LSH method to be more effective than PAM.

Based on Tables 1 to 4, we can determine that, on average, the RMSM method needed the most memory and, comparable to it, the LSH method. The PAM method required significantly less memory, and the reference solution required the least memory compared to all reduction methods.

Finally, we can compare our results with those achieved in [16]. For most ciphers, the authors of [16] revealed the secret key after the first round in a few seconds using two polynomial systems without reduction. These results correspond to our reference solution, which we then further improved with the mentioned reduction methods.

For the small scale variants of the AES ciphers with two rounds, the authors in [16] used the reduction, which searches for the most similar polynomials in a set of polynomials from a few other systems of polynomials. Using this method, the authors managed to compute the secret key of the SR(2,2,4,4) cipher in 1 hour. Using the PAM method, the key for this cipher was calculated in 26 seconds using the same computer as used in [16]. However, the polynomial reduction took an inordinately long time, which resulted in the PAM-based method being slower. However, with the LSH method, the reduction took significantly less time, and the entire computation took 32 seconds, of which only 8 seconds were needed to reveal the key.

The only cipher with three rounds for which we revealed the secret key was SR(3,2,2,4). In [16], after the reduction, the key calculation took over 25 minutes; using the PAM and the LSH methods, it was only 10 minutes on average. In comparison, RMSM took 12.6 minutes for this cipher.

7 Conclusion

In this work, we applied three techniques for reducing overdefined systems of polynomial equations. We used a method for Removing the Most Significant Monomial and two data mining methods, Partitioning Around Medoids and Locality Sensitive Hashing, both used for finding pairs of the most similar polynomials. We compared these methods in terms of computational time and memory requirements. Using these methods, we reduced the original system of polynomial equations, in terms of the number and size of the equations, to a form more suitable for the F4 algorithm designed for calculating the Gröbner base. The experimental results showed that the preprocessing of the equations can significantly reduce the computation time for finding a Gröbner base and, as a result, the secret key.

In this work, we focused on one type of similarity of polynomials based on the number of common monomials. In future work, we plan to focus on several similarity functions, which will not only focus on the number of monomials but also their order in a given monomial order. Further experiments can be focused on higher numbers of equations in the system and quantification of the degree of simplification of the computation of the F4 algorithm based on the number of input equations.

ACKNOWLEDGEMENTS

This work was supported by the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics" and by the Grant Agency of the CTU in Prague, grant No. SGS23/211/OHK3/3T/18 funded by the MEYS of the Czech Republic.

Declarations

The authors have no relevant financial or non-financial interests to disclose.

References

[1] Rijmen, V., Daemen, J.: Advanced encryption standard. Proceedings of federal information processing standards publications, national institute of standards and technology **19**, 22 (2001)

[2] Courtois, N., Klimov, A., Patarin, J., Shamir, A.: Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In: International Conference on the Theory and Applications of Cryptographic Techniques, pp. 392–407 (2000). Springer

[3] Bard, G.: Algebraic cryptanalysis. Springer (2009)

[4] Cid, C., Murphy, S., Robshaw, M.J.B.: Small scale variants of the aes. In: Gilbert, H., Handschuh, H. (eds.) Fast Software Encryption, pp. 145–162. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/11502760_10

[5] Cox, D.A., Little, J., O'Shea, D.: Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 4th edn. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16721-3

[6] Cid, C., Murphy, S., Robshaw, M.J.: Small scale variants of the aes. In: International Workshop on Fast Software Encryption, pp. 145–162 (2005). Springer

[7] Buchmann, J., Pyshkin, A., Weinmann, R.-P.: A zero-dimensional gröbner basis for aes-128. In: Fast Software Encryption: 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers 13, pp. 78–88 (2006). Springer

[8] Murphy, S., Robshaw, M.J.: Essential algebraic structure within the aes. In: Annual International Cryptology Conference, pp. 1–16 (2002). Springer

[9] Toli, I., Zanoni, A.: An algebraic interpretation of 128. In: International Conference on Advanced Encryption Standard, pp. 84–97 (2004). Springer

[10] Zhao, K., Cui, J., Xie, Z., et al.: Algebraic cryptanalysis scheme of aes-256 using gröbner basis. Journal of Electrical and Computer Engineering **2017** (2017)

[11] Courtois, N.T., Pieprzyk, J.: Cryptanalysis of block ciphers with overdefined systems of equations. In: International Conference

on the Theory and Application of Cryptology and Information Security, pp. 267–287 (2002). Springer

[12] Murphy, S., Robshaw, M.: Comments on the security of the aes and the xsl technique. Electronic Letters **39**(1), 36–38 (2003)

[13] Cid, C., Leurent, G.: An analysis of the xsl algorithm. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 333–352 (2005). Springer

[14] Bulygin, S., Brickenstein, M.: Obtaining and solving systems of equations in key variables only for the small variants of aes. Mathematics in Computer Science **3**(2), 185–200 (2010)

[15] Nover, H.: Algebraic cryptanalysis of aes: an overview. University of Wisconsin, USA, 1–16 (2005)

[16] Bielik, M., Jureček, M., Jurečková, O., Lórencz, R.: Yet another algebraic cryptanalysis of small scale variants of aes. In: Int. Conf. on Security and Cryptography (SECRYPT), pp. 415–427 (2022)

[17] Bosma, W., Cannon, J., Playoust, C.: The magma algebra system i: The user language. Journal of Symbolic Computation **24**(3), 235–265 (1997) https://doi.org/10.1006/jsco.1996.0125

[18] Faugére, J.-C.: A new efficient algorithm for computing gröbner bases (f4). Journal of Pure and Applied Algebra **139**(1), 61–88 (1999) https://doi.org/10.1016/S0022-4049(99)00005-5

[19] Cid, C., Murphy, S., Robshaw, M.: Algebraic Aspects of the Advanced Encryption Standard. Springer, New York (2006). https://doi.org/10.1007/978-0-387-36842-9 . hhttps://link.springer.com/book/10.1007/978-0-387-36842-9

[20] Bulygin, S., Brickenstein, M.: Obtaining and solving systems of equations in key variables only for the small variants of aes. Mathematics in Computer Science **3**(2), 185–200 (2010) https://doi.org/10.1007/s11786-009-0020-y

[21] Kaufman, L., Rousseeuw, P.: Finding Groups in Data: An Introduction To Cluster Analysis. John Wiley, NewYork (1990). https://doi.org/10.2307/2532178

[22] Lewin, M.: All About XOR. Overload. [cit. 2022-07-01] (2012). https://accu.org/journals/overload/20/109/lewin_1915/

[23] Kaufman, L., Rousseeuw, P.J.: Finding groups in data: an introduction to cluster analysis. John Wiley & Sons (2009)

[24] Wang, J., Shen, H.T., Song, J., Ji, J.: Hashing for similarity search: A survey. CoRR **abs/1408.2927** (2014) https://doi.org/10.48550/ARXIV.1408.2927 . [cit. 2022-08-27]

[25] Briggs, J.: Faiss: The Missing Manual. [online]. [cit. 2022-07-27]. https://www.pinecone.io/learn/locality-sensitive-hashing/

[26] Giovini, A., Mora, T., Niesi, G., Robbiano, L., Traverso, C.: "one sugar cube, please" or selection strategies in the buchberger algorithm. In: ISSAC 91: International Symposium on Symbolic Algebraic Computation Bonn West Germany, pp. 49–54. Association for Computing Machinery, New York (1991). https://doi.org/10.1145/120694.120701

[27] Brickenstein, M., Dreyer, A.: Polybori: A framework for gröbner-basis computations with boolean polynomials. Journal of Symbolic Computation **44**(9), 1326–1345 (2009) https://doi.org/10.1016/j.jsc.2008.02.017