FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of master's thesis

Title:	Algebraic Cryptanalysis of Small-Scale Variants of Stream Cipher E0
Student:	Bc. Jan Dolejš
Supervisor:	Mgr. Martin Jureček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

The E0 stream cipher is used to secure Bluetooth communication. State-of-the-art attacks include linear cryptanalysis techniques as well as correlation attacks. In recent years, algebraic cryptanalysis, which converts a cipher into a set of polynomial equations whose solution is a secret key, has also found increasing use. The contribution of this work will be the application of standard techniques in the field of algebraic cryptanalysis and try to break the E0 cipher or its simplified version.

Instructions:
1) Design small-scale variants of the E0 cipher.
2) Implement the conversion of the cipher from Step 1) into a system of polynomial equations.
3) Briefly describe the basics of the theory of Groebner bases used in solving systems of polynomial equations.
4) Apply a suitable program (e.g., Magma) to compute Groebner bases for the system of equations and discuss the results.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 2 October 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Algebraic Cryptanalysis of Small-Scale Variants of Stream Cipher E0

Jan Dolejš

Department of Information Security
Supervisor: Mgr. Martin Jureček, Ph.D.

January 11, 2024

Acknowledgements

I express my profound gratitude to my supervisor, Mgr. Martin Jureček, Ph.D., for his helpful insights and ability to give my research direction.

I also thank my mom and my brother, who, whatever the situation may be, support me and help me. The same goes for my friends; I appreciate you and admire you.

I dedicate this work to my dad; thank you for what you have done for me. I wish we had more time.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the ``Work''), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on January 11, 2024 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Dolejš, Jan. *Algebraic Cryptanalysis of Small-Scale Variants of Stream Cipher E0*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstract

This work introduces and demonstrates innovative progress in the algebraic analysis of the small-scale variants of the stream cipher E0 from the Bluetooth standard. We design the small-scale variants and represent them using a set of polynomial equations. Our work reveals a possible linear relation between the number of keystream bits and the size of the small-scale E0 variants, improving the performance of the used solvers. Our best run finds the initial configuration in 178.5 seconds for the 22-bit E0 version. Using local sensitivity hashing, we improved the computational time of the SAT solver from 453.1 seconds to 85.3 seconds for the 19-bit E0 version.

Keywords E0, small-scale variants, algebraic cryptanalysis, Gröbner bases, SAT, LSH

Abstrakt

Tato práce představuje a demonstruje nové postupy v algebraické analýze zmenšených variant proudové šifry E0 ze standardu Bluetooth. V práci jsme formulovali podobu zmenšených variant šifry a ty reprezentujeme pomocí množiny polynomiálních rovnic. Naše práce odhaluje existenci možného lineárního vztahu mezi počtem keystream bitů a počtem neznámých, snižujíc výpočetní čas použitých řešičů. Náš nejlepší experiment odhalil počáteční konfiguraci za $178,5$ sekund pro 22 bitovou verzi E0. Pomocí lokálně citlivého hašování jsme zlepšili výpočetní čas SAT řešiče z 453,1 sekundy na 85,3 sekundy pro 19bitovou verzi E0.

Klíčová slova E0, zmenšené varianty, algebraická kryptoanalýza, Gröbnerovy báze, SAT, LSH

Contents

List of Figures

Introduction

Encryption has become one of the significant parts of our daily lives. And, without much of an exaggeration, our lives depend on encryption. Encryption is used while we perform online bank transactions, message our friends or family, or even watch the news online. Without encryption, our lives would become much easier to track by large companies or government agencies. And yet, last year has shown that encryption may be endangered, as the politicians of Great Britain [1] and the members of the EU council [2] stood against it. What was the reason for such a decision? Encryption not only protects innocent citizens but also criminals, who use it for drug deals, sexual child abuse, ransomware, and many more. Law enforcement agencies refer to this as going dark, as it is hard for them, if not impossible, to uncover the encrypted messages [3].

But what makes it difficult for the agencies to uncover encrypted messages? One would assume they require the secret key for the messages only. Not precisely; they need the knowledge of the key, but they also require the knowledge of the encryption scheme used to uncover the original message, in other words, plaintext. We can assume we know the encryption scheme and only search for the key. The encryption scheme should make guessing the key so hard that brute force should be the only option. Keys have various lengths. Modern ciphers, such as AES [4], can use keys that have 128, 196, or 256 bits. And to brute force such a key, we would need to try 2^{128}, 2^{196}, or 2^{256} combinations. If every combination were to take a second, 2^{13} combinations would take a little over an hour, 2^{17} a day and a half, 2^{21} three weeks and a half, and 2^{25} a little over a year. This makes the brute force search impossible, considering that the universe's age is approximately $2^{33.67}$ years.

In summary, it is the quality of the cipher that ensures that the key is going to be hard to guess. This is where cryptanalysis comes into play. Cryptanalysis aims to find weaknesses that may be included in the cipher [5]. We can break the cipher and show that the claimed bit security is not as strong as claimed. For example, the secret key length is 128, but a cipher's weakness reduces the

key's security to 96 bits. Thus, we require only 2^{96} combinations instead of the 2^{128}.

In this work, we perform an algebraic cryptanalysis of the stream cipher E0 from the Bluetooth standard [6]. E0 has been studied over the last 25 years, yet there is a missing specification of small-scale variants, and the use of the Gröbner basis has been applied just recently. In this thesis, we aim to fill in the gap and perform the cryptanalysis on the small-scale versions using an appropriate cipher formulation and suitable solvers.

Chapter 1 goes through the essential mathematical structures required to understand the core concepts used in the experimental part.

Chapter 2 reviews the mathematical description of Linear Feedback Shift Registers, and describes the E0 stream cipher.

Chapter 3 briefly explains our implementation and the tools required for our work.

Chapter 4 combines the knowledge from previous chapters and analyzes the usage of two computational methods. Both methods are supplied with a different number of keystream bits, which improves the computational times significantly in the case of one of the approaches. Furthermore, we test an equation reduction technique to improve the results of the second approach.

Chapter 5 reviews related work that has either been used as a literature source or provides different perspectives on the E0 cipher.

Chapter 1

Essential Mathematical Structures

This chapter introduces the underlying theory of Gröbner bases for ideals in polynomial rings. The name comes from Austrian mathematician Bruno Buchberger [7], who named Gröbner bases after his advisor Wolfgang Gröbner in his dissertation in 1965. A Russian mathematician, Nikolai Maximovich Gjunter (or Ghunter) published a similar idea in 1913 [8]. It remained unknown until 1941, when he published a paper on his previous work.

Buchberger summarized the idea of Gröbner bases in [9] in less than 20 pages. However, it should be noted the text assumes previous knowledge of the theory that Gröbner bases are built upon. Undergraduate and graduate texts, such as [10, 11, 12], capture the theory of Gröbner bases in a few hundred pages. However, covering Gröbner bases in depth would be beyond the scope of this work; thus, we lay the background with the most important mathematical concepts and refer the reader to the mentioned literature. Unless stated otherwise, we will follow the theory given by [10].

1.1 Polynomials

Polynomials became familiar to many during earlier education. We learned to analyze their roots, derivatives, asymptotes, and various other characteristics, often focusing on polynomials with a single variable. These properties are also of interest for the general form of a polynomial. Consider the following polynomial $h(x, y, z) = x + yz + xyz$. The atomic parts of h are called monomials and are the products of the input variables.

Definition 1 (Monomial)**.** *A product of the following form*

$$x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdots x_n^{\alpha_n} \tag{1.1}$$

is called a monomial, where $\alpha_1, \ldots, \alpha_n \in \mathbb{N}_0$*. The total degree of this monomial is* $\alpha_1 + \ldots + \alpha_n$*.*

We can simplify the notation by introducing an n-tuple $\alpha = (\alpha_1, \ldots, \alpha_n)$ and setting

$$x^\alpha = x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdots x_n^{\alpha_n}, \tag{1.2}$$

and let $|\alpha| = \alpha_1 + \ldots + \alpha_n$ denote the total degree of the monomial x^α.

Following the definition, we see that h contains three monomials, x, yz and xyz. However, we have not yet properly defined what a polynomial is.

Definition 2 (Polynomial)**.** *A sum over a finite number of n-tuples* $\alpha = (\alpha_1, \ldots, \alpha_n)$

$$f = \sum_\alpha a_\alpha x^\alpha, a_\alpha \in k, \tag{1.3}$$

is called a polynomial. We call a_α *the coefficients in the field* k*. The set of all polynomials in the field* k *is denoted* $k[x_1, \ldots, x_n]$*.*

We refer to $k[x_1, \ldots, x_n]$ as a polynomial ring. In our case, h would be defined over a polynomial ring $k[x, y, z]$. For example, we can set $k = \mathbb{F}_2$, where $\mathbb{F}_2$ is a finite field of order two (or a Galois field of order two) [13]. Then, $a_\alpha \in \{0, 1\}$. We extend the terminology for polynomials with the following definition.

Definition 3. *For polynomial* $f = \sum_\alpha a_\alpha x^\alpha$ *in* $k[x_1, \ldots, x_n]$*, we use the following terminology.*

- *We refer to* a_α *as the coefficient of the monomial* x^α*.*
- *We call* $a_\alpha x^\alpha$ *a term of* f *if* $a_\alpha \neq 0$*.*
- *If* $f \neq 0$*, then the total degree of* f *is the maximum* $|\alpha|$*, where* $a_\alpha \neq 0$*. The total degree of* $f = 0$ *is undefined. We denote the total degree of* f *as* $\deg(f)$*.*

Thus, we refer to xyz as the term of h, as its coefficient $a_\alpha = 1$. The total degree of h is $\deg(h) = 3$.

1.2 Varieties and Ideals

When working with polynomials, we may be interested in evaluating a polynomial with specific values for its variables. By considering all possible values of the variables, we get an affine space.

Definition 4 (Affine Space)**.** *Let* k *be a field and* $n \in \mathbb{N}$*. The* n*-dimensional affine space over* k *is defined as follows.*

$$k^n = \{(a_1, \ldots, a_n) \mid a_1, \ldots, a_n \in k\} \tag{1.4}$$

Let $h(x, y) = x + y + xy$ be a polynomial in $\mathbb{F}_2[x, y]$. Then, its affine space is $\mathbb{F}_2^2 = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$. Note that h is an algebraic representation of logical OR. Naturally, this leads to a question: At what points does the polynomial evaluate to zero? The answer is given by affine varieties.

Definition 5 (Affine Variety)**.** *Given a field k, and polynomials $f_1, \ldots, f_s$ in $k[x_1, \ldots, x_n]$, we define an affine variety to be the set*

$$\mathbf{V}(f_1, \ldots, f_s) = \{(a_1, \ldots, a_n) \in k^n \mid f_i(a_1, \ldots, a_n) = 0, \forall i : 1 \leq i \leq s\}. \quad (1.5)$$

We say that $\mathbf{V}(f_1, \ldots, f_s)$ is defined by $f_1, \ldots, f_s$.

Given the $h(x, y)$ above, we see its affine variety is $\mathbf{V}(h) = \{(0, 0)\}$.

The next definition introduces an ideal, an important mathematical concept from ring theory [13]. We can understand ideals as a way to factorize a ring, in our case, a polynomial ring, and understand them as a language to help us compute affine varieties [10]

Definition 6 (Ideal)**.** *An ideal I is a subset of $k[x_1, \ldots, x_n]$ that satisfies the following:*

- $0 \in I$.
- *If $f, g \in I$, then $f + g \in I$.*
- *If $f \in I$ and $h \in k[x_1, \ldots, x_n]$, then $hf \in I$.*

The next definition gives us a naturally occurring ideal that can be used to represent a system of polynomial equations.

Definition 7. *Let $k[x_1, \ldots, x_n]$ be a polynomial ring and $f_1, \ldots, f_s$ its polynomials. We set*

$$\langle f_1, \ldots, f_s \rangle = \left\{ \sum_{i=1}^{s} h_i f_i \mid h_1, \ldots, h_s \in k[x_1, \ldots, x_n] \right\} \quad (1.6)$$

Before giving an example of the definition, we will first state that the definition indeed represents an ideal.

Lemma 1. *Let $f_1, \ldots, f_s$ be polynomials in $k[x_1, \ldots, x_n]$. Then, $\langle f_1, \ldots, f_s \rangle$ is an ideal of $k[x_1, \ldots, x_n]$ and we say that it is generated by $f_1, \ldots, f_s$.*

We can prove Lemma 1 using the definition of an ideal (Definition 6). The proof is shown in [10].

Definition 7 can be interpreted using a set of polynomial equations [10]. Let $f_1, \ldots, f_s$ and $h_1, \ldots, h_s$ be from $k[x_1, \ldots, x_n]$ and we have the following set of equations:

$$\begin{aligned} f_1 &= 0, \\ f_2 &= 0, \\ &\vdots \\ f_s &= 0. \end{aligned}$$

We can then write

$$h_1 f_1 + h_2 f_2 + \ldots + h_s f_s = 0.$$

The following proposition puts a relation on two bases representing the same ideal.

Proposition 1. *Let $f_1, \ldots, f_s$ and $g_1, \ldots, g_t$ be bases of an ideal in $k[x_1, \ldots, x_n]$, and $\langle f_1, \ldots, f_s \rangle = \langle g_1, \ldots, g_t \rangle$, then $\boldsymbol{V}(f_1, \ldots, f_s) = \boldsymbol{V}(g_1, \ldots, g_t)$.*

Consider the following variety $\mathbf{V}(2x^2+3y^2-30, x^2-y^2-5)$. We can show, that $\langle 2x^2 + 3y^2 - 30, x^2 - y^2 - 5 \rangle = \langle x^2 - 9, y^2 - 4 \rangle$, so that

$$\mathbf{V}(2x^2 + 3y^2 - 30, x^2 - y^2 - 5) = \mathbf{V}(x^2 - 9, y^2 - 4) = \{(\pm 3, \pm 2)\} \tag{1.7}$$

using the proposition above. By using a different basis, it may be easier to determine the variety.

Let us now consider the opposite direction. Given an affine variety, we search for an ideal that represents it.

Definition 8. *Given an affine variety V, where V is a subset of k^n, we set*

$$\mathbf{I}(V) = \{f \in k[x_1, \ldots, x_n] \mid f(a_1, \ldots, a_n) = 0,\ \forall (a_1, \ldots, a_n) \in V\}. \tag{1.8}$$

Without proving we state, that $\mathbf{I}(V)$ is an ideal (see [10]).

Lemma 2. *Let $V \subseteq k^n$ be an affine variety. We say that $\boldsymbol{I}(V) \subseteq k[x_1, \ldots, x_n]$ is an ideal, and we will call the ideal of V.*

Consider the variety $\{(0,0)\}$. Its ideal is $\mathbf{I}(\{(0,0)\}) = \langle x, y \rangle$ [10]. Naturally, we could ask whether $\mathbf{I}(\mathbf{V}(f_1, \ldots, f_s)) = \langle f_1, \ldots, f_s \rangle$ is true. We can give one simple counter-example. Consider the variety $\mathbf{V}(x^2, y^2) = \{(0,0)\}$. We know that $\mathbf{I}(\{(0,0)\}) = \langle x, y \rangle$ and thus $\mathbf{I}(\mathbf{V}(x^2, y^2)) \neq \langle x^2, y^2 \rangle$ $(x, y \notin \langle x^2, y^2 \rangle)$.

1.3 Monomials and Orderings

In solving linear equations with a limited number of variables, e.g., x, y, and z, some algorithms decide in what order the variables will be solved. The order of the elimination can be better illustrated through Gaussian elimination. By swapping some of the columns, we change the order in which we eliminate the variables. The result remains the same. However, one order could potentially lead to the solution quicker than the other.

Definition 9 (Linear Ordering)**.** *A relation $\succ$ on $\mathbb{N}_0^n$ is called a linear ordering, if for each pair $\alpha \in \mathbb{N}_0^n$ and $\beta \in \mathbb{N}_0^n$ only one of the three following statements is true.*

$$\alpha \succ \beta,\ \alpha = \beta,\ \beta \succ \alpha \tag{1.9}$$

This corresponds to what we have written above and generalizes the idea in terms of relations. We would naturally expect the existence of the least significant term in such a case. If so, we say that the terms are well-ordered.

Definition 10 (Well-Ordering)**.** *Given a relation $\succ$ on $\mathbb{N}_0^n$ and a subset $A \subseteq \mathbb{N}_0^n$, such that $A \neq \emptyset$, we call $\succ$ a well-ordering, if there exists $\alpha \in A$, such that $\beta \succ \alpha$ for every $\beta \neq \alpha$ in A.*

We can now combine both properties of linear and well-ordering together with transitivity to get the following definition of monomial ordering.

Definition 11 (Monomial Ordering)**.** *Let $k[x_1, \ldots, x_n]$ be a polynomial ring, a relation $\succ$ on $\mathbb{N}_0^n$ is called a monomial ordering on $k[x_1, \ldots, x_n]$ if and only if the following applies:*

- *$\succ$ is a linear ordering on $\mathbb{N}_0^n$.*
- *If $\alpha \succ \beta$ and $\gamma \in \mathbb{N}_0^n$, then $\alpha + \gamma \succ \beta + \gamma$.*
- *$\succ$ is a well-ordering on $\mathbb{N}_0^n$.*

One of the most well-known orderings is the alphabet. The alphabet is a lexicographic ordering, allowing us to sort words in alphabetical or lexicographical order. For example, we know that `FIT` $\succ_{lex}$ `Matfyz`.

Definition 12 (Lexicographic Ordering)**.** *Given two vectors $\alpha, \beta \in \mathbb{N}_0^n$, we say that $\alpha \succ_{lex} \beta$ if the leftmost nonzero entry of $\alpha - \beta \in \mathbb{Z}^n$ is positive. We call $\succ_{lex}$ a lexicographic ordering. If $\alpha \succ_{lex} \beta$, we write $x^\alpha \succ_{lex} x^\beta$.*

We can now distinguish between two monomials, for example, $x^2yz \succ_{lex} y^4z$, since $(2,1,1) - (0,4,1) = (2,-3,0)$ (assuming that $x \succ y \succ z$). The lexicographic ordering is a good academic example, although it is not much used in practice. The ordering may appear too simple in some cases, as, for example, $x \succ_{lex} y^2z^6$ for $x \succ y \succ z$. Another example of a monomial ordering

is graded lex ordering, which considers the total degree of a monomial first and then uses the lexicographic ordering to distinguish between monomials of the same order.

Definition 13 (Graded Lex Ordering)**.** *Let $\alpha, \beta \in \mathbb{N}_0^n$. If*

$$|\alpha| = \sum_{i=1}^{n} \alpha_i > |\beta| = \sum_{i=1}^{n} \beta_i,$$

or

$$|\alpha| = |\beta| \text{ and } \alpha \succ_{lex} \beta,$$

we write $\alpha \succ_{grlex} \beta$. $\succ_{grlex}$ is a graded lexicographic ordering.

Using the same example from above, we would write that $y^4z \succ_{lex} x^2yz$, since the total degree of the first monomial is 5 and the total degree of the second monomial is 4. The last example of a monomial ordering in this work is the graded reverse lex ordering, which is most commonly used in computations for its efficiency [10].

Definition 14 (Graded Reverse Lex Ordering)**.** *Let $\alpha, \beta \in \mathbb{N}_0^n$. If*

$$|\alpha| = \sum_{i=1}^{n} \alpha_i > |\beta| = \sum_{i=1}^{n} \beta_i,$$

or $|\alpha| = |\beta|$ and the rightmost nonzero entry of $\alpha - \beta \in \mathbb{Z}^n$ is negative, then we write $\alpha \succ_{grevlex} \beta$. $\succ_{grevlex}$ is a graded reverse lex ordering.

Let us now compare the monomial orderings. Let $f = x^3yz^4 + x^2y^7z + xy^4z + x^6y^2z^8 + x^5y^4z^7 \in k[x, y, z]$ with $x \succ y \succ z$. With respect to $\succ_{lex}$, we get:

$$f = x^6y^2z^8 + x^5y^4z^7 + x^3yz^4 + x^2y^7z + xy^4z.$$

With respect to $\succ_{grlex}$, we get:

$$f = x^6y^2z^8 + x^5y^4z^7 + x^2y^7z + x^3yz^4 + xy^4z.$$

With respect to $\succ_{grevlex}$, we get:

$$f = x^5y^4z^7 + x^6y^2z^8 + x^2y^7z + x^3yz^4 + xy^4z.$$

We end this section by extending the terminology for polynomials with regards to a given monomial ordering.

Definition 15. *Let $f = \sum_\alpha a_\alpha x^\alpha \in k[x_1, \ldots, x_n]$, $f \neq 0$, and let $\succ$ be a monomial ordering. We add the following terminology to polynomials.*

- *The multidegree of f is*

$$\text{multideg}(f) = \max(\alpha \in \mathbb{N}_0^n \mid a_\alpha \neq 0), \tag{1.10}$$

  where the maximum is taken with respect to $\succ$.

- *The leading coefficient of f is*

$$\text{LC}(f) = a_{\text{multideg}(f)} \in k. \tag{1.11}$$

- *The leading monomial of f is*

$$\text{LM}(f) = x^{\text{multideg}(f)}. \tag{1.12}$$

- *The leading term of f is*

$$\text{LT}(f) = \text{LC}(f) \cdot \text{LM}(f). \tag{1.13}$$

1.4 Gröbner Bases

Without the knowledge of the Gröbner bases, we will assume, for now, that Gröbner basis gives us a different finite generating set for an ideal, allowing us to find varieties more easily. Division is one of the main components of the algorithms that compute the Gröbner basis. The division allows us to describe one polynomial in terms of other polynomials. Although the theory in this text may seem not to require the division algorithm, it is used in some of the proofs omitted from this text.

Theorem 1 (Division Algorithm)**.** *Given a monomial ordering $\succ$ on $\mathbb{N}_0^n$, and given an ordered s-tuple of polynomials $F = (f_1, \ldots, f_s)$ in $k[x_1, \ldots, x_n]$, we can write every $f \in k[x_1, \ldots, x_n]$ as follows*

$$f = q_1 f_1 + \ldots + q_s f_s + r, \tag{1.14}$$

where $q_i, r \in k[x_1, \ldots, x_n]$, and r must be either 0, or a linear combination with coefficients in $k[x_1, \ldots, x_n]$, which are not divisible by any of $\text{LT}(f_1), \ldots, \text{LT}(f_s)$. r is called a remainder of f on division by F. Additionally, if $q_i f_i \neq 0$, then

$$\text{multideg}(f) \geq \text{multideg}(q_i f_i). \tag{1.15}$$

We will denote the remainder r as $\overline{f}^F$.

A special case of ideals is a monomial ideal, which we will now define.

Definition 16 (Monomial Ideal)**.** *Let $I \subset k[x_1, \ldots, x_n]$ be an ideal. We say that I is a monomial ideal if there is a subset $A \subset \mathbb{N}_0^n$, such, that I contains all polynomials $\sum_{\alpha \in A} h_\alpha x^\alpha$, where $h_\alpha \in k[x_1, \ldots, x_n]$. We write $I = \langle x^\alpha \mid \alpha \in A \rangle$.*

For example, $\langle x^2y^3, x^5y^7\rangle$ is a monomial ideal. We can then determine whether a monomial lies in the ideal through division.

Lemma 3. *Given a monomial ideal $I = \langle x^\alpha \mid \alpha \in A\rangle$ we say that a monomial x^β lies in I if and only if $x^\alpha \mid x^\beta$ (x^β is divisible by x^α) for some α.*

Using the example above, we say that $x^3y^4 \in \langle x^2y^3, x^5y^7\rangle$, as $x^2y^3|x^3y^4$. But $x^2y^2 \notin \langle x^2y^3, x^5y^7\rangle$, as none of the monomials in the ideal divide x^2y^2.

The division algorithm (see Theorem 1) uses the leading terms of polynomials to check whether one polynomial divides the other [10]. In the following definition, we only consider the leading terms of an ideal and the ideal they span.

Definition 17. *Given an ideal $I \subset k[x_1, \ldots, x_n]$ other than $\{0\}$ and a fixed monomial ordering on $k[x_1, \ldots, x_n]$ we introduce the following notation:*

- *Let $\mathrm{LT}(I)$ be the set of leading terms of nonzero elements of I:*

$$\mathrm{LT}(I) = \{cx^\alpha \mid \exists f \in I \setminus \{0\}, \mathrm{LT}(f) = cx^\alpha\} \tag{1.16}$$

- *$\langle \mathrm{LT}(I)\rangle$ denotes the ideal generated by the elements of $\mathrm{LT}(I)$.*

Note that for an ideal $I = \langle f_1, \ldots f_s\rangle$ it is not necessarily true that $\langle \mathrm{LT}(f_1), \ldots, \mathrm{LT}(f_s)\rangle$ would be the same as $\langle \mathrm{LT}(I)\rangle$ [10]. The next theorem gives us an important feature of ideals.

Theorem 2 (Hilbert Basis Theorem)**.** *For every ideal $I \subset k[x_1, \ldots, x_n]$ there exists a finite generating set. We can write $I = \langle g_1, \ldots, g_t\rangle$ for some $g_1, \ldots, g_t \in I$.*

One of the major outcomes of the Hilbert Basis Theorem is the existence of a basis with special properties, which we call the Gröbner basis.

Definition 18 (Gröbner Basis)**.** *Let $I \subset k[x_1, \ldots, x_n]$ be an ideal with fixed monomial order on $k[x_1, \ldots, x_n]$, and $I \neq \{0\}$. A finite subset $G = \{g_1, \ldots, g_t\}$ of I is called a Gröbner basis, if*

$$\langle \mathrm{LT}(g_1), \ldots, \mathrm{LT}(g_t)\rangle = \langle \mathrm{LT}(I)\rangle. \tag{1.17}$$

In some cases, an infinite amount of Gröbner bases for a given ideal can exist. The ambiguity can be avoided if we apply a stricter condition on the form of the bases.

Definition 19 (Reduced Gröbner Basis)**.** *Let I be a polynomial ideal and let G be a Gröbner basis of I. We will call G a reduced Gröbner basis, if*

- $\forall p \in G : \mathrm{LC}(p) = 1$.
- $\forall p \in G : p \notin \langle \mathrm{LT}(G \setminus \{p\})\rangle$.

Chapter 2

The E0 Algorithm

In this chapter, we describe one of the main building blocks used within the E0 cipher, the linear feedback shift registers. We proceed to describe the cipher itself, formulating the specification from the Bluetooth standard in mathematical terms. We design small-scale cipher variants and express the cipher using polynomial equations. Lastly, we illustrate an approach that could be used to generate more equations and an approach that would reduce the equations to speed up retrieving the initial configuration of the linear feedback shift registers used within E0.

2.1 Linear Feedback Shift Register

Linear feedback shift registers (**LFSR**s) are one of the building blocks of E0. LFSRs are generally used for their efficiency and undemanding hardware implementation requirements [14]. LFSRs are also used for their excellent statistical properties but are cryptographically insecure. Thus, the output of LFSRs is usually nonlinearly combined, which is also the case of E0. Unless stated otherwise, this section is based on [14].

Definition 20 (Linear Feedback Shift Register)**.** *An LFSR of length L over $\mathbb{F}_q$ produces a semi-infinite* [1] *sequence $(s_t)_{t\geq 0}$, which satisfies a linear recurrence relation of degree L over $\mathbb{F}_q$*

$$s_{t+L} = \sum_{i=1}^{L} c_i s_{t+L-i}, \forall t \geq 0, \tag{2.1}$$

where the coefficients $c_i \in \mathbb{F}_q, \forall i \in \{1, \ldots, L\}$. We call the coefficients c_i the feedback coefficients of the LFSR.

[1] Semi-infinite refers to mathematical objects bounded in one direction but unlimited in the other [15]. For example, the interval $(c, +\infty)$ is semi-infinite, as it is bounded by the constant c.

There are two ways to represent an LFSR: the Fibonacci and the Galois representation. The Fibonacci representation (see Figure 2.1) essentially corresponds to the Definition 20. A current state is represented by the contents of $(s_t, \ldots, s_{t+L-1})$. An initial state of the register, $(s_0, \ldots, s_{L-1})$, is filled with arbitrary values from $\mathbb{F}_q$.

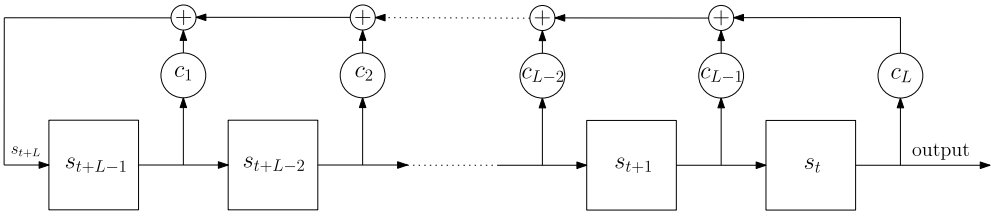


Figure 2.1: Fibonacci LFSR

In Figure 2.1, to get the next state, we first calculate s_{t+L}, and then we shift all register states to the right. This process is controlled by an external clock, which may or may not be regular. Note that in the figure, the output of the LFSR is the rightmost stage, s_t. This is not the case for E0, as it uses the output from stage s_{t+k} (see Section 2.2.1). However, the choice of the output stage does not change the properties of the LFSR. We can clock the LFSR k times to get s_{t+k} as the rightmost stage, thus, the output. By clocking the LFSR, we change its internal state.

In this thesis, we will be working over $\mathbb{F}_2$. We can simplify the visualization of the LFSR, as the feedback coefficients are from $\{0, 1\}$. For example, consider the LFSR of length 5 with feedback coefficients $(c_1, c_2, c_3, c_4, c_5) = (1, 0, 0, 1, 1)$, which is displayed in Figure 2.2.

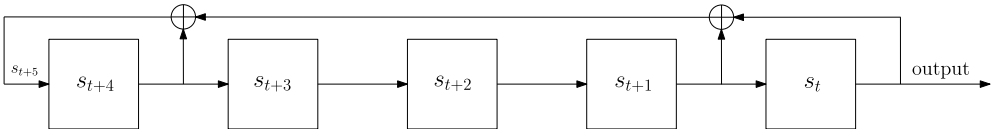


Figure 2.2: Fibonacci LFSR over $\mathbb{F}_2$
$L = 5$ and feedback coefficients are $(1, 0, 0, 1, 1)$.

We can represent the feedback coefficients of an LFSR of length L with a vector $(c_1, \ldots, c_L)$. However, this is not very common, and we usually represent the LFSR with its feedback polynomial or its characteristic polynomial.

Definition 21 (Feedback and Characteristic Polynomial)**.** *For an LFSR of length L with feedback coefficients $(c_1, \ldots, c_L)$ we define its feedback polynomial P as follows:*

$$P(X) = 1 - \sum_{i=1}^{L} c_i X^i. \tag{2.2}$$

Similarly, we define its characteristic polynomial as follows:

$$P^*(X) = X^L P(1/X) = X^L - \sum_{i=1}^{L} c_i X^{L-i}. \tag{2.3}$$

Thus, for the LFSR with feedback coefficients $(1, 0, 0, 1, 1)$ over $\mathbb{F}_2$, we would get $P(X) = X^5 + X^4 + X + 1$ and $P^*(X) = X^5 + X^4 + X + 1$.

An LFSR of length L over $\mathbb{F}_q$ can produce up to q^L different sequences, depending on the initial state. Using the following theorem, we can characterize all such sequences.

Theorem 3. *Given an LFSR of length L over $\mathbb{F}_q$ with feedback polynomial P, we say that a sequence $(s_t)_{t\geq 0}$ is generated by the LFSR if and only if there exists a polynomial $Q \in \mathbb{F}_q[X]$ with $\deg(Q) < L$ such that the linear recurrence $(s_t)_{t\geq 0}$ of degree L satisfies*

$$\sum_{t\geq 0} s_t X^t = \frac{Q(X)}{P(X)}. \tag{2.4}$$

We say, that the polynomial Q is completely determined by the coefficients of P and the initial state of the LFSR:

$$Q(X) = -\sum_{j=0}^{L-1} X^j \left(\sum_{k=0}^{j} s_k c_{j-k} \right), \tag{2.5}$$

and $P(X) = -\sum_{i=0}^{L} c_i X^i$.

The proof can be found in [14]. Theorem 3 has two key implications. Any LFSR with a feedback polynomial P is also generated by a different LFSR with a feedback polynomial, which is a multiple of P. We can also use this property to find a more suitable representation of the LFSR that could potentially have better properties. For example, the hardware implementation cost of LFSR could be reduced. Hence the following question: what is the minimal representation of the LFSR? The following definition answers the question.

Definition 22. *Let $(s_t)_{t\geq 0}$ be a linear recurring sequence, then there exists a unique polynomial P_0 such that its constant term is equal to 1, and the generating function of $(s_t)_{t\geq 0}$ is given by*

$$\sum_{t\geq 0} s_t X^t = \frac{Q_0(X)}{P_0(X)}, \tag{2.6}$$

where P_0 and Q_0 are relatively prime.

The shortest LFSR which generates $(s_t)_{t\geq 0}$ has length $L = \max(\deg(P_0), \deg(Q_0) + 1)$, and its feedback polynomial is equal to P_0. The characteristic polynomial of the shortest LFSR, which generates $(s_t)_{t\geq 0}$, is called the minimal polynomial of the sequence.

The minimal polynomial has a crucial role in determining the period of an LFSR.

Proposition 2. *Let P_0 be a minimal polynomial. Then, the least period of a linear recurring sequence equals minimal $e \in \mathbb{N}_0$ such that $P_0(X)|X^e + 1$.*

A minimal polynomial P_0, for which a sequence has a maximal period $2^{\deg(P_0)} - 1$ is called a *primitive polynomial*. In this work, we will use primitive polynomials for the LFSRs to design small-scale variants of the E0 cipher.

Earlier in this chapter, we stated that the choice of the output does not change the properties of LFSRs and that we can clock k times to get the output from stage s_{t+k}. Once we know the state of the LFSR at time $t + k$, we can also determine the state at time t. In other words, we can also clock the LFSR in the opposite direction, which we prove in the following theorem.

Theorem 4. *For an LFSR of length L over $\mathbb{F}_q$ with its feedback coefficients $(c_1, \ldots, c_L)$, where $c_L \neq 0$, there exists an* LFSR$'$ *of length L over $\mathbb{F}_q$ with feedback coefficients $-c_L^{-1}(c_{L-1}, \ldots, c_2, c_1, -1)$ that produces the output of the LFSR in reverse.*

Proof. By using Definition 20 we can write

$$\begin{aligned} s_{t+L} = \sum_{i=1}^{L} c_i s_{t+L-i} &= c_1 s_{t+L-1} + \ldots + c_L s_t \\ c_L s_t &= s_{t+L} - (c_1 s_{t+L-1} + \ldots + c_{L-1} s_{t+1}) \\ s_t &= c_L^{-1} s_{t+L} - c_L^{-1} c_1 s_{t+L-1} - \ldots - c_L^{-1} c_{L-1} s_{t+1} \end{aligned}$$

Let $c'_i = -c_L^{-1} c_{L-i}$ for $i \in \{1, \ldots, L-1\}$, and $c'_L = c_L^{-1}$. We can then simplify s_t to

$$s_t = \sum_{i=1}^{L} c'_i s_{t+i}. \tag{2.7}$$

Let $s'_{t+k} = s_{t-k}$. We can then get the general form of s'_{t+L}

$$s'_{t+L} = \sum_{i=1}^{L} c'_i s_{t-L+i} = \sum_{i=1}^{L} c'_i s'_{t+L-i}, \tag{2.8}$$

which is the linear recurrence relation of degree L over $\mathbb{F}_q$ for LFSR$'$. □

2.2 Description Of The E0 Algorithm

As of writing this text, the Bluetooth technology is already 25 years old [16]. It was officially launched in 1998 to connect computers and mobile devices [17].

The name itself was supposed to be a placeholder only; however, after a thorough research from the marketing team, they have decided to keep the name as is.

The Bluetooth standard has changed over the course of its existence [6]. The encryption within the Bluetooth Basic Rate/Enhanced Data Rate (**BR/EDR**) utilizes Advanced Encryption Standard (**AES**). Some older devices can still use the legacy stream cipher E0. That is, if at least one of the two devices does not support the newer Bluetooth standard, the encryption is downgraded to E0.

We can divide the encryption process of E0 into three parts:

- initialization of four LFSRs,
- keystream generation,
- encryption/decryption.

In this work, we will mainly focus on the second part. E0 is a symmetric cipher; thus, the keystream is used for both encryption and decryption – it is XORed with a plaintext or a ciphertext.

2.2.1 Encryption

We begin by describing the encryption part of E0. E0 is a combination generator built from four regularly clocked LFSRs, whose output is combined with a Boolean function with four memory bits. The generator is used twice, once during the initialization and then during the keystream generation.

The four LFSRs used in E0 have lengths $L_1 = 25$, $L_2 = 31$, $L_3 = 33$, and $L_4 = 39$, that is, their lengths in total are 128. Their feedback polynomials are

$$\begin{aligned} P_1(x) &= x^{25} + x^{20} + x^{12} + x^8 + 1 \\ P_3(x) &= x^{31} + x^{24} + x^{16} + x^{12} + 1 \\ P_3(x) &= x^{33} + x^{28} + x^{24} + x^4 + 1 \\ P_4(x) &= x^{39} + x^{36} + x^{28} + x^4 + 1, \end{aligned}$$

where all $P_i(x)$ are primitive polynomials. The output bit of the i-th LFSR, where $i \in \{1, 2, 3, 4\}$ is $x_t^{(i)}$. The outputs of the LFSRs are combined using a finite state machine (**FSM**) with 2^4 states. The FSM is also referred to as a summation combiner [14]. The outline of the encryption used in E0 is shown in Figure 2.3.

As a first step, a 3-bit value $y_t \in \mathbb{Z}$ is computed

$$y_t = x_t^{(1)} + x_t^{(2)} + x_t^{(3)} + x_t^{(4)}. \tag{2.9}$$

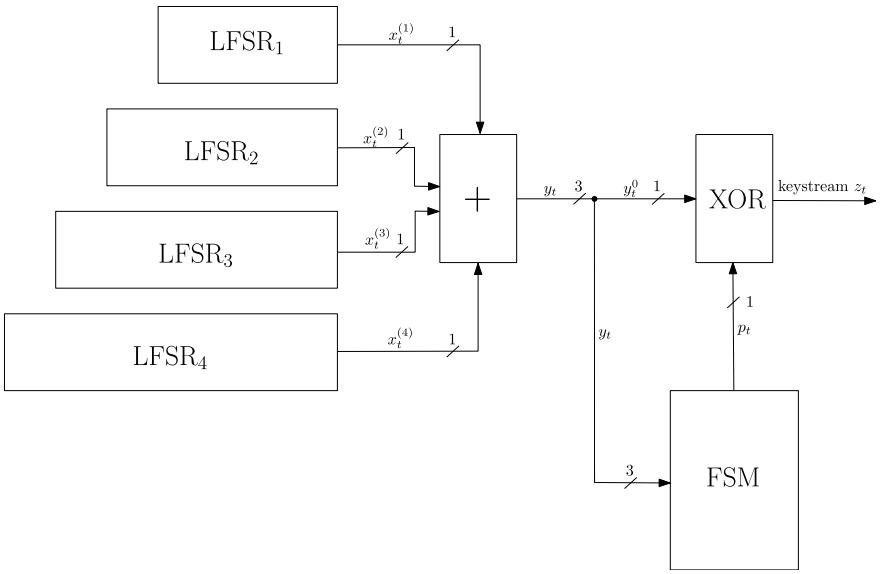


Figure 2.3: Outline of E0 encryption

y_t is then used as an input for the FSM, and its least significant bit $y_t^{(0)}$ is XORed with output from the FSM. The FSM uses an internal memory that consists of four bits, $c_t = (q_t, p_t) \in \{0,1\}^2$ and $c_{t-1} = (q_{t-1}, p_{t-1}) \in \{0,1\}^2$. The memory bits are first set in the initialization part of the algorithm. Inside of the FSM, values y_t and c_t are combined as follows:

$$s_{t+1} = \left\lfloor \frac{y_t + c_t}{2} \right\rfloor, \tag{2.10}$$

where $s_{t+1} \in \{0,1,2,3\}$. Next, the update of the memory bits contains two linear bijections T_1 and T_2:

$$T_1 : (x_1, x_0) \mapsto (x_1, x_0),$$
$$T_2 : (x_1, x_0) \mapsto (x_0, x_1 \oplus x_0).$$

Finally, the update of the memory bits is:

$$c_{t+1} = (q_{t+1}, p_{t+1}) = s_{t+1} \oplus T_1(c_t) \oplus T_2(c_{t-1}). \tag{2.11}$$

In Figure 2.4, Z is a lag operator

$$Zc_{t+1} = c_t, \forall t > 0. \tag{2.12}$$

Finally, the generated keystream from is the combination of the outputs of the LFSRs the FSM:

$$z_t = x_t^{(1)} \oplus x_t^{(2)} \oplus x_t^{(3)} \oplus x_t^{(4)} \oplus p_t = y_t^{(0)} \oplus p_t. \tag{2.13}$$

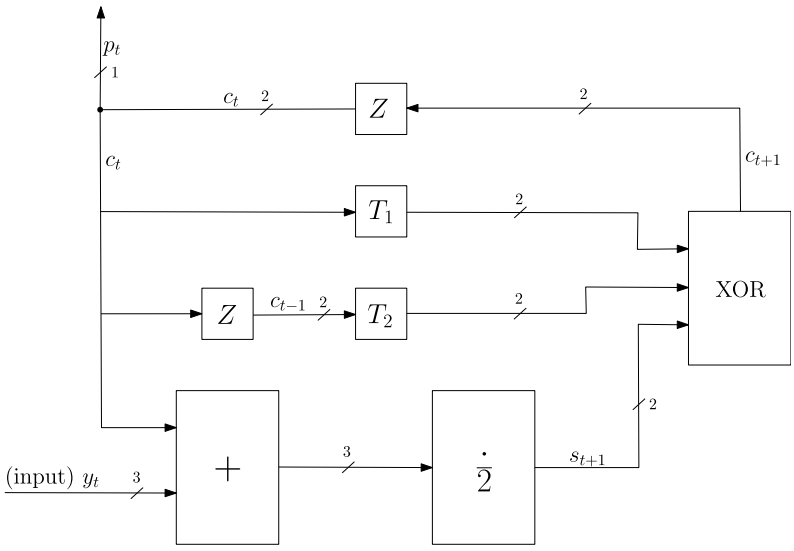


Figure 2.4: E0 Finite State Machine

2.2.2 Initialization

The Bluetooth communication is carried out between a *central* device and peripheral devices. The central device supplies a synchronization reference to the peripheral devices; in the case of E0, the central device shares its clock. E0 is a symmetric cipher; therefore, all devices require the same initialization variables for the algorithm. More specifically, the initialization variables are required to initialize the four LFSRs and the four memory bits, as mentioned in Section 2.2.1.

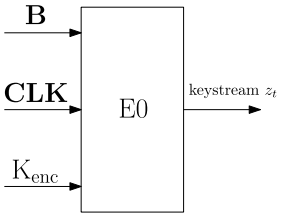


Figure 2.5: E0 with its inputs and output

E0 accepts three inputs – the central device's Bluetooth device address and its real-time clock, and an encryption key generated by a hash function E3. The Bluetooth device address, which we will denote by **B** ($\mathbf{B_C}$ for the central device), has 48 bits. From the central device's clock, **CLK**, 26 bits are used. We will not go into the details of the hash function E3. Its output is the encryption key, K_{enc}, which always has 128 bits. The size of K_{enc} can be internally reduced to multiples of 8, more specifically, to size $8L$, where

$L \in \{1, \ldots, 16\}$. The resulting key with the new size, $\mathrm{K_{ses}}$, is calculated as

$$\mathrm{K_{ses}} = \left. g_2^{(L)} \mathrm{K_{enc}} \right|_{g_1^{(L)}}, \tag{2.14}$$

where $g_1^{(L)}$ and $g_2^{(L)}$ are polynomials specified in Table C.1. Additionally, to the three inputs, six constant bits `111001` are used to initialize the LFSRs and the memory bits. In total, 208 bits are used for the initialization. A simple way to visualize E0 is given by Figure 2.5. However, it is unclear that the encryption process is carried out twice. A better way to visualize E0 is given by Figure 2.6.

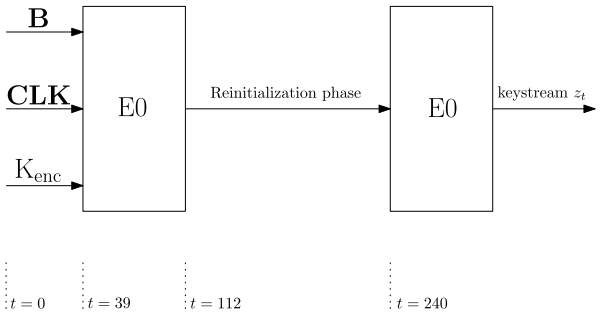


Figure 2.6: E0 with initialization

Before describing the process of initialization of the LFSRs, we are going to separate the 208 initialization bits into four vectors. For the initialization variables, we will use the following bit notation: $\mathbf{B_C} = (b_1, \ldots, b_{48})$, $\mathbf{CLK} = (\mathrm{clk}_1, \ldots, \mathrm{clk}_{26})$, and $\mathrm{K_{ses}} = (k_1, \ldots, k_{128})$. Then, we define vectors $I^{(1)}, I^{(2)}, I^{(3)}$, and $I^{(4)}$, each corresponding to $\mathrm{LFSR}_1, \mathrm{LFSR}_2, \mathrm{LFSR}_3$, and LFSR_4 respectively.

$$\begin{aligned}
I^{(1)} &= (\mathrm{clk}_{25}, k_1, \ldots, k_8, k_{33}, \ldots, k_{40}, k_{65}, \ldots, k_{72}, k_{97}, \ldots, k_{104}, \\
&\quad \mathrm{clk}_9, \ldots, \mathrm{clk}_{16}, b_{17}, \ldots, b_{24}), \\
I^{(2)} &= (1, 0, 0, \mathrm{clk}_1, \ldots, \mathrm{clk}_4, k_9, \ldots, k_{16}, k_{41}, \ldots, k_{48}, k_{73}, \ldots, k_{80}, \\
&\quad k_{105}, \ldots, k_{112}, b_1, \ldots, b_8, b_{25}, \ldots, b_{32}), \\
I^{(3)} &= (\mathrm{clk}_{26}, k_{17}, \ldots, k_{24}, k_{49}, \ldots, k_{56}, k_{81}, \ldots, k_{88}, k_{113}, \ldots, k_{120}, \\
&\quad \mathrm{clk}_{17}, \ldots, \mathrm{clk}_{24}, b_{33}, \ldots, b_{40}), \\
I^{(4)} &= (1, 1, 1, \mathrm{clk}_5, \ldots, \mathrm{clk}_8, k_{25}, \ldots, k_{32}, k_{57}, \ldots, k_{64}, k_{89}, \ldots, k_{96}, \\
&\quad k_{121}, \ldots, k_{128}, b_9, \ldots, b_{16}, b_{41}, \ldots, b_{48}).
\end{aligned} \tag{2.15}$$

Note, that the size of $I^{(1)}$ and $I^{(3)}$ is 49 bits, and the size of $I^{(2)}$ and $I^{(4)}$ is 55 bits. At the beginning, we set $t = 0$ for the LFSRs. All of the operations below are done simultaneously bit by bit. However, different LFSRs may be doing different operations at time t. For example, for $t = 38$, LFSR_4 has last bit of

its initial state set ($x_{38}^{(4)} = I_{39}^{(4)}$, see Equation 2.16), while LFSR_1 is updating its stage $x_{38}^{(1)}$ (see Definition 20) mixed with the initialization variable bit $I_{39}^{(1)}$ (see Equation 2.17)

Using the notation for the LFSRs' output from Section 2.2.1, we set

$$\begin{aligned}
(x_0^{(1)}, \ldots, x_{24}^{(1)}) &= (I_1^{(1)}, \ldots, I_{25}^{(1)}), \\
(x_0^{(2)}, \ldots, x_{30}^{(2)}) &= (I_1^{(2)}, \ldots, I_{31}^{(2)}), \\
(x_0^{(3)}, \ldots, x_{32}^{(3)}) &= (I_1^{(3)}, \ldots, I_{33}^{(3)}), \\
(x_0^{(4)}, \ldots, x_{38}^{(4)}) &= (I_1^{(4)}, \ldots, I_{39}^{(4)}).
\end{aligned} \tag{2.16}$$

Then, stages $x_{25}^{(1)}, \ldots, x_{48}^{(1)}$. $x_{31}^{(2)}, \ldots, x_{54}^{(2)}$, $x_{33}^{(3)}, \ldots, x_{48}^{(3)}$, and $x_{39}^{(4)}, \ldots, x_{54}^{(4)}$ are updated according to Definition 20 mixed with bits from the initialization variables as follows:

$$\begin{aligned}
x_{\ell+25}^{(1)} &= x_\ell^{(1)} \oplus x_{\ell+5}^{(1)} \oplus x_{\ell+13}^{(1)} \oplus x_{\ell+17}^{(1)} \oplus I_{\ell+L+1}^{(1)}, \forall \ell \in \{0, \ldots, 23\} \\
x_{\ell+31}^{(2)} &= x_\ell^{(2)} \oplus x_{\ell+7}^{(2)} \oplus x_{\ell+15}^{(2)} \oplus x_{\ell+19}^{(2)} \oplus I_{\ell+L+1}^{(2)}, \forall \ell \in \{0, \ldots, 23\} \\
x_{\ell+33}^{(3)} &= x_\ell^{(3)} \oplus x_{\ell+5}^{(3)} \oplus x_{\ell+9}^{(3)} \oplus x_{\ell+29}^{(3)} \oplus I_{\ell+L+1}^{(3)}, \forall \ell \in \{0, \ldots, 15\} \\
x_{\ell+39}^{(4)} &= x_\ell^{(4)} \oplus x_{\ell+3}^{(4)} \oplus x_{\ell+11}^{(4)} \oplus x_{\ell+35}^{(4)} \oplus I_{\ell+L+1}^{(4)}, \forall \ell \in \{0, \ldots, 15\}
\end{aligned} \tag{2.17}$$

After this, the stages of the LFSRs are updated in a standard manner up(as in Definition 20). The LFSRs are shifted with each clock regularly until $t = 239$.

At the same time, we initialize the memory bits with zeros,

$$(c_0, \ldots, c_{39}) = (0, \ldots, 0). \tag{2.18}$$

The memory bits c_{t+1} for $t \geq 39$ and $t \leq 239$ are updated according to Equation 2.11.

Starting at $t = 39$, the output symbols [2] (or output bits, see Equation 2.13) are generated. Thus, in total, 200 output symbols are generated. Out of them, the last 128 output symbols are used. We will use the following notation: the output symbol at time $t = 112$ is denoted o_1, at time $t = 113$ it is denoted o_2, up to $t = 239$, where the last output symbol is denoted o_{128}. We define four reinitialization vectors $O^{(1)}, O^{(2)}, O^{(3)}$, and $O^{(4)}$, each corresponding to

[2] Section 2.2.1 refers to the output symbols as keystream bits. However, the output bits are not used for encryption but rather to reinitialize the LFSRs.

$\text{LFSR}'_1, \text{LFSR}'_2, \text{LFSR}'_3$, and LFSR'_4 respectively.

$$\begin{aligned} O^{(1)} &= (o_{97}, o_{72}, \ldots, o_{65}, o_{40}, \ldots, o_{33}, o_8, \ldots, o_1), \\ O^{(2)} &= (o_{104}, \ldots, o_{98}, o_{80}, \ldots, o_{73}, o_{48}, \ldots, o_{41}, \\ &\quad o_{16}, \ldots, o_9), \\ O^{(3)} &= (o_{121}, o_{112}, \ldots, o_{105}, o_{88}, \ldots, o_{81}, o_{56}, \ldots, o_{49}, \\ &\quad o_{24}, \ldots, o_{17}), \\ O^{(4)} &= (o_{128}, \ldots, o_{122}, o_{120}, \ldots, o_{113}, o_{96}, \ldots, o_{89}, \\ &\quad o_{64}, \ldots, o_{57}, o_{32}, \ldots, o_{25}). \end{aligned} \tag{2.19}$$

Note, that the sizes of $O^{(i)}$ for $i \in \{1, 2, 3, 4\}$ each correspond to the sizes of the LFSRs respectively. We also use a different notation, LFSR'_i, as after the reinitialization the LFSRs are used to generate the keystream bits. Finally, using the reinitialization variables, we can set the initial state of all of the LFSRs as follows.

$$\begin{aligned} (x_0'^{(1)}, \ldots, x_{24}'^{(1)}) &= (O_1^{(1)}, \ldots, O_{25}^{(1)}), \\ (x_0'^{(2)}, \ldots, x_{30}'^{(2)}) &= (O_1^{(2)}, \ldots, O_{31}^{(2)}), \\ (x_0'^{(3)}, \ldots, x_{32}'^{(3)}) &= (O_1^{(3)}, \ldots, O_{33}^{(3)}), \\ (x_0'^{(4)}, \ldots, x_{38}'^{(4)}) &= (O_1^{(4)}, \ldots, O_{39}^{(4)}). \end{aligned} \tag{2.20}$$

2.2.3 Small-Scale Variants of E0

Since working with the original size of E0 would not be feasible for the equipment we are using (see Chapter 4), we will be working with small-scale variants of E0. In short, we will be reducing the sizes of the LFSRs used within the cipher. We will not change the FSM, as it is the main source of the unique behavior of E0.

To reduce the size of the LFSRs, we can focus on their feedback polynomials, which are, by the Bluetooth standard [6], required to be primitive; otherwise, the period of the LFSRs would not be maximal. The authors of E0 have decided to use such polynomials, whose hamming weight (**HW**) is equal to 5. HW of a polynomial is defined as the number of non-zero coefficients of the polynomial. The choice of HW equal to 5 is reasoned by good statistical properties and better hardware design [6]. However, for lower-degree polynomials, having HW equal to 5 is not possible. Thus, if it is not possible to follow the requirements, we choose polynomials with HW [3] equal to 3 – this choice is then kept the same among all of the four polynomials.

The calculation of HW can be seen well from the binary representation of a polynomial. For example, the polynomial x^4+x^3+1 has binary representation

[3] We have evaluated all primitive polynomials from [18] up to length 20 and we have not found any primitive polynomials with HW equal to 2 or 4.

11001, and its HW is equal to 3. It is also a primitive polynomial. In general, every primitive polynomial includes the constant 1; thus, we can simplify the binary representation to 1100. The hexadecimal form of this polynomial is then `0xC`. We will represent all primitive and feedback polynomials using the hexadecimal notation.

We will use the following notation for small-scale E0 variants:

$$\mathrm{E0}(A, B, C, D), \tag{2.21}$$

where A, B, C, D are primitive polynomials of LFSR_1, LFSR_2, LFSR_3, and LFSR_4 respectively. For example, E0(`0x3`, `0x6`, `0xC`, `0x14`) tells us, that LFSR_1 uses primitive polynomial `0x3`, i.e. $x^2 + x + 1$, LFSR_2 uses primitive polynomial `0x6`, i.e. $x^3 + x^2 + 1$, and so on. From now on we will assume that only hexadecimal representation is used when referring to the small-scale variants; thus, using the example above we will write E0(`3`, `6`, `C`, `14`). We do not change the initialization, as the experiments will be carried out only after the reinitialization phase of the LFSRs. The small-scale variants of E0 that keep the ratio of the original sizes will be denoted $\mathrm{E0}^*(\ldots)$. We calculate the respective lengths of the new LFSRs L'_i, where $i \in \{1, 2, 3, 4\}$ as

$$L'_i = \begin{cases} \left\lfloor L' \frac{L_i}{L} \right\rfloor, & \text{if } i = 1 \\ L'_{i-1} + \max\left(1, \left\lfloor \frac{L_i - L_{i-1}}{L} L' \right\rfloor\right), & \text{otherwise,} \end{cases} \tag{2.22}$$

where L' is the total length of the LFSRs in the small-scale version of E0, and L is the original length of the full-scale version of E0 ($L = 128$). For example, for $L' = 18$ we get $L'_1 = \left\lfloor 18 \cdot \frac{25}{128} \right\rfloor = 3$, for L'_2 we get

$$L'_2 = L'_1 + \max\left(1, \left\lfloor \frac{31 - 25}{128} \cdot 18 \right\rfloor\right) = 3 + 1 = 4. \tag{2.23}$$

Similarly, for $L'_3 = 5$ and for $L'_4 = 6$. Note, that Equation 2.22 also holds for $L' = L = 128$.

2.2.4 Representing E0 Encryption Using Polynomial Equations

To represent the whole encryption algorithm algebraically, we need to convert the FSM from Section 2.2. The FSM updates the memory bits, for which we need the equations. By representing the FSM with a truth table and using, for example, `Sage` [19] mathematical software, we can extract its algebraical normal form (**ANF**). See Appendix C.2 for more details. Before writing down the equations for the memory bits, let us first define a symmetric polynomial.

Definition 23 (Symmetric Polynomial)**.** *A symmetric polynomial is defined as follows*

$$\pi^n_{t,N} = \bigoplus_{1 \le i_1 < i_2 < \ldots < i_n \le N} x_t^{(i_1)} x_t^{(i_2)} \cdots x_t^{(i_n)}, \tag{2.24}$$

where $x_t^{(i)} \in \mathbb{F}_2$.

Since E0 uses $N = 4$, we will assume that $\pi_{t,4}^n = \pi_t^n$. Furthermore, x_t^i is the output of the i-th LFSR. For example, $\pi_{t,4}^2 = \pi_t^2 = x_t^{(1)}x_t^{(2)} \oplus x_t^{(1)}x_t^{(3)} \oplus x_t^{(1)}x_t^{(4)} \oplus x_t^{(2)}x_t^{(3)} \oplus x_t^{(2)}x_t^{(4)} \oplus x_t^{(3)}x_t^{(4)}$. Using Definition 23 of a symmetric polynomial, we can rewrite the Equation 2.13 for the keystream bit z_t:

$$z_t = \pi_t^1 \oplus p_t. \tag{2.25}$$

From Appendix C.2 we have that the memory bits q_{t+1} and p_{t+1} are updated with the following equations:

$$\begin{aligned} q_{t+1} &= \pi_t^4 \oplus \pi_t^3 p_t \oplus \pi_t^2 q_t \oplus \pi_t^1 p_t q_t \oplus q_t \oplus p_{t-1} \\ p_{t+1} &= \pi_t^2 \oplus \pi_t^1 p_t \oplus q_t \oplus q_{t-1} \oplus p_{t-1} \oplus p_t \end{aligned} \tag{2.26}$$

This way, we get a tool for generating equations for each bit of the keystream z_t. However, the equations for q_{t+1} and p_{t+1} will grow quickly in size while increasing their degree. We can express the equations differently by following Armknecht's approach [20] (see Appendix C.3). The main idea behind the transformation is based on the fact that the output of the FSM and the LFSRs is linearly combined (see Equation 2.25). We can eliminate the memory bits one by one and get the following equation that applies to every four subsequent keystream bits:

$$\begin{aligned} 0 = {} & z_{t+3}(z_{t+1}\pi_{t+1}^1 \oplus \pi_{t+1}^2 \oplus \pi_{t+1}^1 \oplus 1) \oplus \\ & z_{t+2}(z_{t+1}\pi_{t+2}^1\pi_{t+1}^1 \oplus z_{t+1}\pi_{t+1}^1 \oplus \pi_{t+1}^2\pi_{t+2}^1 \oplus \\ & \quad \pi_{t+1}^2 \oplus \pi_{t+2}^1\pi_{t+1}^1 \oplus \pi_{t+2}^1 \oplus \pi_{t+1}^1 \oplus 1) \oplus \\ & z_{t+1}(z_t\pi_{t+1}^1 \oplus \pi_{t+1}^3 \oplus \pi_{t+2}^2\pi_{t+1}^1 \oplus \pi_{t+1}^2 \oplus \\ & \quad \pi_{t+3}^1\pi_{t+1}^1 \oplus \pi_{t+1}^1\pi_t^1 \oplus \pi_{t+1}^1 \oplus 1) \oplus \\ & z_t(\pi_{t+1}^2 \oplus \pi_{t+1}^1 \oplus 1) \oplus \\ & \pi_{t+1}^4 \oplus \pi_{t+2}^2(\pi_{t+1}^2 \oplus \pi_{t+1}^1 \oplus 1) \oplus \\ & \pi_{t+1}^2(\pi_{t+3}^1 \oplus \pi_t^1) \oplus \\ & \pi_{t+3}^1(\pi_{t+1}^1 \oplus 1) \ \oplus \pi_{t+1}^1\pi_t^1 \oplus \pi_t^1 \end{aligned} \tag{2.27}$$

Note, that the Equation 2.27 has degree four, specifically given by $\pi_{t+1}^2\pi_{t+2}^2$ and π_{t+1}^4. Thanks to Armknecht's results, we know that this is the lowest degree for four and five subsequent keystream bits.

2.2.5 Generating Additional Equations

Let us now explore the cipher, considering the generated keystream first. The keystream is generated by the algorithm described in Section 2.2.1 above. For each keystream bit z_t at time t, we have 8 bits that we can set, the LFSRs'

output, $x_t^{(1)}, x_t^{(2)}, x_t^{(3)}, x_t^{(4)}$, and the memory bits q_t, p_t, q_{t-1} and p_{t-1}. Thus, we have 2^8 possible combinations, which give us the keystream bit z_t. If $z_t = 0$, we have 2^7 possible configurations of the bits. The same applies if $z_t = 1$. When given a keystream $(z_t, z_{t+1}, \ldots, z_{t+\ell})$, we should conclude that there exist $2^2(2^5)^{\ell+1}$ possible configurations of the LFSR bits and the memory bits. This is not necessarily true, as the number of possible states will decline with more keystream bits. However, it is not clear how many keystream bits would be required to reduce the number of possible states sufficiently.

The following question can be raised: can we build valid equations for certain consecutive keystream bits and equations possibly simpler than the derived Equation 2.27 in Section 2.2.4? To answer the question, we propose the following. Given a keystream $(z_t, z_{t+1}, \ldots, z_{t+\ell})$, we will find valid configurations of the LFSR and the memory bits[4]. Let $\mathbf{A}$ be a matrix whose columns represent the LFSR and the memory bits at times $t, t+1, \ldots, t+\ell$. The rows of $\mathbf{A}$ represent the valid configurations.

We can use the column variables to build new monomials. For example, we can build a monomial of degree two using $x_t^{(1)}$ and $x_t^{(2)}$, getting $x_t^{(1)}x_t^{(2)}$. We can do this for all column variables, but similarly, as in Section 2.2.4, we will use only the LFSR output variables. Note that combining two or more variables corresponds to a logical AND of the specific values in the columns. Let $\mathbf{B}$ be the matrix which consists of all possible monomials of a degree up to k, where k can be at most $4(\ell + 1)$. The number of columns of $\mathbf{B}$ will be $m = \sum_{i=1}^{k} \binom{4(\ell+1)}{i}$.

Finally, we will build matrix $\mathbf{C}$, whose columns will represent polynomials built from the monomials given by matrix $\mathbf{B}$. For example, we can combine columns $x_t^{(1)}x_t^{(2)}$ and $x_t^{(2)}x_t^{(3)}$, from which we get the polynomial $x_t^{(2)}(x_t^{(1)} + x_t^{(3)})$. The combination of the columns of the matrix $\mathbf{B}$ corresponds to a logical XOR. Similarly, as we did for the matrix $\mathbf{B}$, we will limit the number of terms by l (that is, the number of monomials in a polynomial cannot surpass l). l can be at most $2^m - 1$ (we ignore zero polynomial). The number of columns of $\mathbf{C}$ will be $\sum_{i=1}^{l} \binom{m}{i}$. In matrix $\mathbf{C}$ we will search for columns whose values are the same row-wise (that is, the columns are either $\mathbf{0}$ or $\mathbf{1}$). Such columns represent polynomials, which are valid for keystream $(z_t, z_{t+1}, \ldots, z_{t+\ell})$, when equal to the value in the column.

Such a computational approach is expensive, as the number of monomials and the number of polynomials grows with factorial, and we have not been able to identify any equations for $\ell = 3, k = 4$ and $l = 2$ (4 consecutive keystream bits, monomials of maximum degree 4 and 2 terms in a polynomial). The Jupyter Notebook `Possible States` implements the approach. Some of the steps could be eliminated; for example, if the search is conducted for two consecutive keystream bits first, one could use this as prior knowledge when

[4]We can recursively search for correct configurations, checking whether the bits satisfy equations given by Section 2.2.1.

examining three consecutive keystream bits (that is, one does not have to examine the relationship between states for the second and the third bit). Despite the expense of the approach, it only needs to run once.

2.3 Reduction of the Equations

Following the works of Berušková [21] and Minarovič [22], we decided to employ a preprocessing technique on the generated equations from the E0 cipher. The idea is to reduce the equations to improve the computational time of a given technique. This work will test one reduction technique, Local Sensitivity Hashing (**LSH**).

LSH aims to find the most similar elements in a given set and place such elements in different sets, called buckets [23]. We refer to the elements that are the most similar as nearest neighbors. We can formalize the problem of searching for the nearest neighbor as follows: Given an element q we search for $x \in \{x_1, \ldots, x_n\}$ such, that minizes the distance between q and x, $\text{dist}(q, x)$. We can define the distance between the elements in various ways; however, the properties of a metric space must be met [21]. One such metric is given by the XOR of two polynomials, also utilized by [21, 22]

We can separate the process of LSH into three steps [24, 22]: shinling, MinHashing, Bucketing.

2.3.1 Shinling

This step aims to create a set of monomials (shinlings) for each polynomial in the polynomial system of equations. Then, by assigning all of the unique monomials in the polynomial system a unique integer, we create a dictionary. For example, consider two polynomials $f = x + y + z + xy$ and $g = x + xy + xz + yz$. The set of unique monomials for f is $f_s = \{x, y, z, xy\}$, and for g $g_s = \{x, xy, xz, yz\}$. We then proceed to create a dictionary, incrementing a counter for each unique monomial in the polynomial system: $D = \{x : 1, y : 2, z : 3, xy : 4, xz : 5, yz : 6\}$.

Using the dictionary D, we can now create sparse vectors for f and g, representing whether the monomial is present (1) in the original polynomial or not (0). Hence, following the results from Table 2.1, $v_1 = (1, 1, 1, 1, 0, 0)$ and $v_2 = (1, 0, 0, 1, 1, 1)$.

2.3.2 MinHashing

The MinHashing step creates dense vectors, or signatures, from the sparse vectors from the previous step. The MinHashing step uses m random permutations whose sizes equal the number of unique polynomials in the polynomial system. We search for the minimum number such that the position in the sparse vector is non-zero for each permutation. For example, we will use

Table 2.1: Creating sparse vectors for shinlings

f_s	g_s	D	v_1	v_2
x	x	$x:1$	1	1
y		$y:2$	1	0
z		$z:3$	1	0
xy	xy	$xy:4$	1	1
	xz	$xz:5$	0	1
	yz	$yz:6$	0	1

Table 2.2: Creating dense vectors – signatures

p_1	p_2	p_3	v_1
6	6	**2**	1
5	5	4	1
4	2	3	1
3	**1**	5	1
2	4	6	0
1	3	1	0

$m = 3$ random permutations for the sparse vectors from the previous section. In Table 2.2, we first search for the minimum number where the position given by permutation p_1 in the sparse vector v_1 is non-zero. For the first two numbers, 1 and 2, the positions in v_1 are 0. The first non-zero number for the permutation is 3. For p_2, we have the first non-zero element in v_1 for 1. And finally, for p_3, we have the first non-zero element in v_1 for 2. Thus, the resulting signature of v_1 is $s_1 = (3, 1, 2)$. Similarly, for v_2 we get $s_2 = (1, 1, 1)$.

2.3.3 Bucketing

The last step of LSH is to create buckets from the signatures. The signatures are separated into subparts used to identify similar items from different signatures in the buckets. After adding a signature to the buckets, its items are used as an index into the bucket, adding the index of the signature as an item. Using the signatures from the previous section, we use s_1 to index the empty buckets with each of its items. Since the index of s_1 is 1, we add 1 to all of the buckets at the indices.

$$(3:1)$$
$$(1:1)$$
$$(2:1)$$

Using signature s_2 we index the buckets again, this time using index 2.

$$(3:1,1:2)$$
$$(1:1,2)$$
$$(2:1,1:2)$$

As we can see, the second element of both signatures is 1, thus resulting in bucket with two two items, $(1:1,2)$.

Finally, we search for buckets containing at least two items. Then, we combine all the items in such buckets, resulting in candidate pairs, which can be used for equation reduction. All of the candidate polynomial pairs are XORed together.

We then use the following metric: if the number of monomials in a polynomial is less than $t \cdot M$, where t is a threshold and M is the average number of monomials for all polynomials from the original polynomial system, we add the polynomial to a new set. We search for such polynomials from the original polynomial system and the new polynomials created through the candidate pairs.

Chapter 3

Implementation

We used several different programming/scripting languages, each suitable for different tasks:

- `C/C++` for the initial implementation of E0,
- `Python >v3.8` for computational tasks, scripting, visualisations,
- `Sage v9.0 & v10.1` [19] for algebraic tasks and connection with Magma,
- `Magma 2.25-5` [25] for computation of Gröbner basis,
- `{ba,z}sh` for scripting.

3.1 Implementation of E0

Following Section 2.2.1 and Arnaud Delmas' implementation [26] of E0, we produced our version of E0 (see `bt_utils.cpp`) to give us a better understanding of the inner workings of the cipher. We verified the correctness of the implementation using Sections 1.1.2-1.1.5 of the Bluetooth Standard [6].

File `keystream_gen.py` contains a simplified version of the E0 algorithm without the initialization phase. We use the script to generate keystream bits with randomly initialized LFSRs at the beginning. The random initialization of the LFSRs substitutes the reinitialization phase. After initializing each LFSR, we check whether their initial states are non-zero. If the initial state were a zero one, we simply perform the random initialization again until we find a non-zero initial state.

We can specify the number of keystreams we want to generate using `-runs`, the number of keystream bits we want to generate using `-kbits`, and the random seed using `-random`. The primitive polynomials used in the LFSRs and their outputs are specified using `-lfsrX` and `-oX`, where `X` is the number

corresponding to the LFSR. For example, to specify the primitive polynomial and the output of the first LFSR, we would write `-lfsr1 0x3 -o1 0`.

We use primitive polynomials only for the LFSRs in the cipher.

3.2 Symbolic Representation of E0

Similarly, we implemented a simplified symbolic version of E0. We use a symbolic representation of the LFSRs – instead of specific bits, we output the combinations of the LFSRs' unknowns, which are then used in Equation 2.27. The specific outputs of the LFSRs, $x_t^{(i)}$, where $i \in \{1, 2, 3, 4\}$ (see Section 2.2.1), are precalculated first.

File `E0.sage` implements the symbolic representation of E0 after the reinitialization phase and implements a way to generate equations depending on what formulation we require – both recursive and Armknecht's formulation are implemented. Equations are then pickled using Python's `pickle` module and we store information about the generated equations in a separate file.

Using the file `generate_rec.py` we can specify the primitive polynomials and outputs of the LFSRs in the same manner as in Section 3.1. We can also set the type of equations we would like to generate. For Armknecht's formulation, we would use `-type armknecht`.

The `sage` to `magma` interface is not as efficient when converting the equations. The interface has to represent the polynomials as a list of strings and is limited by the size of the strings. Converting one polynomial to `magma` requires more operation than just generating the equations. For this reason, we created a file `generate_equations.m` that generates the equations directly in `magma`. The difference is that this has to be done for each keystream, as, for performance reasons, one has to substitute the keystream bits while generating the equations.

3.3 Local Sensitivity Hashing

File `LSH.py` implements local sensitivity hashing (**LSH**), which is a method we use to reduce the number of equations used in a given experiment. The implementation is based on pinecone's [27] implementation in Python. The specifics of LSH can be found in Section 2.3 and its usage in Section 4.2.

3.4 Running the Experiments

Our work uses two approaches to equation solving – `Magma`'s [25] F4 algorithm and CryptoMiniSat's [28] SAT solver. Since the symbolic implementation of E0 is done solely in `Sage`, we need to convert the equations to `Magma`. Even though `Sage` has an interface to `Magma`, we needed to implement the conversion

of equations from `Sage` to `Magma`, as the interface is unable to convert large equations. This problem was also encountered in Bielik's work [29].

Although the implementation is done efficiently in `Magma`, for some instances, it tends to either not finish the computation or freeze entirely. It is unclear what causes these issues since `Magma` is a closed source software; however, when encountered, we re-run the experiments, which usually solved the problem. After `Magma` finishes the computation of the Gröbner basis, we let it calculate the affine variety of the calculated basis. This gives us a better understanding of the sparsity of the solutions and how much time would be needed to verify the initial configuration of the LFSRs.

To compare the efficiency of calculating the Gröbner basis, we also use CryptoMiniSat's SAT solver. In the version of `Magma` we used, the F4 algorithm is only single-threaded; thus, we set the number of threads of the SAT solver to one. Since we search for all possible solutions with `Magma`, we do the same with the SAT solver. Since the equations are in ANF, they must be converted to the conjunctive normal form (**CNF**). We use Martin Albrecht's converter [30], which is implemented in `sage` [19]. Note that this is the dense form of the ANF to CNF conversion. `sage` [30] also implements a sparse form. Unfortunately, it is computationally unfeasible for large equation systems because of the approach through truth tables [30].

Chapter 4

Experiments

In this chapter, we describe our experimental approach to attacking small-scale variants of E0. In the experiments we assume the knowledge of both the plaintext and the ciphertext, thus allowing us to work with keystream only. The experiments were run on a single computer platform with two processors (Intel Xeon Gold 6136 CPU @ 3.00 GHz), with 755 GB of DDR4 RAM running the Ubuntu 20.04.5 LTS operating system.

In each experiment, we use different versions of the small-scale variants of E0 (see Section 2.2.3). For each version, we generate 15 different keystreams. The keystreams are generated with randomly initialized LFSRs after the reinitialization phase following the E0 algorithm (see Section 2.2.1). All of the LFSRs must have a non-zero initial state.

We compare two approaches, the computation of Gröbner basis using Magma's [25] implementation of the F4 algorithm and CryptoMiniSat's SAT solver [28]. Once the Gröbner basis is calculated, we also search for its affine variety to verify the solution. CryptoMiniSat allows parallelizing the computation, but since Magma runs on a single thread only, we run the SAT solver with a single thread only. Using the SAT solver, we also search for all possible solutions; thus, the results from the computation of Gröbner basis (and their affine varieties) and SAT solver are essentially the same. The SAT solver times include conversion from the algebraic form of the equations to the logical form (CNF) [30].

In the early stage of this work, we tried to utilize the recursive formulation of the memory bits (see Equations 2.26). However, this approach proved to be computationally infeasible as the equations grow exponentially in size, and also the fact that we are required to find the solution for four more unknows, $(q_t, p_t, q_{t-1}, p_{t-1})$.

Table 4.1: Initial experiments with keystream bits equal to the number of unknowns

		F4			SAT
E0 type	Kb[a] Uc[b]	G. Basis Time (s)	Variety Time (s)	Mem (GiB)	Time (s)
E0*(3, 6, c, 14)	14	4.9 ± 0.1	2.1 ± 0.1	0.3 ± 0.0	3.1 ± 0.1
E0(3, 6, c, 30)	15	12.0 ± 0.1	6.2 ± 0.6	0.5 ± 0.0	6.6 ± 0.2
E0(3, 6, c, 60)	16	47.7 ± 1.1	27.5 ± 8.5	2.1 ± 0.0	12.0 ± 0.2
E0(3, 6, 12, 60)	17	196.6 ± 2.6	79.5 ± 50.4	8.1 ± 0.3	32.9 ± 0.5
E0*(6, c, 14, 30)	18	1283.9 ± 29.2	253.6 ± 162.0	28.1 ± 1.3	120.1 ± 2.1
E0(6, c, 14, 60)	19	3465.5 ± 49.3	494.3 ± 41.9	53.7 ± 2.3	230.0 ± 5.2
E0(6, c, 30, 60)	20	26400.3 ± 324.9	2077.6 ± 756.1	224.3 ± 2.5	624.7 ± 28.3

[a] Keystream bits
[b] Unknowns count

4.1 Initial Experiments Using Armknecht's Formulation

Armknecht's formulation of the E0 cipher (see Section 2.2.1) efficiently generates the equations. The set of equations is only generated once; it is then necessary to substitute the keystream into the equations. The substitution becomes more demanding with the increasing number of unknowns and the increasing number of equations; however, for the experiments in this work, we assume that the substitution is negligible.

The Bluetooth standard [6] sets the maximum number of keystream bits for the full-scale version of the E0 cipher to 2745. It is unclear how this number would scale for small-scale cipher variants, but for the minimum number of keystream bits, we will assume that it is equal to the number of unknowns (for example, a 14-bit variant of E0 would generate at least 14 keystream bits before reset). Thus, we begin the experiments using an equal amount of bits of the keystream to the number of unknowns. We show the average computational times and the average memory consumption for the F4 algorithm with the standard deviation [5] in Table 4.1.

We can immediately see the exponential growth in computational time when the number of unknowns increases. There is a notable difference between the two approaches. The time for the F4 algorithm increases around 6.5 times when moving from 17 to 18 unknowns, while the time for the SAT solver increases 3.7 times. Similarly, when moving from 19 to 20 variables, the time for the F4 algorithm increases around 7.6 times, while the time for the SAT solver increases around 2.7 times. The more significant jumps between times in the F4 algorithm may be related to higher memory consumption. Note that

[5] average $\pm$ standard deviation

Table 4.2: Unique monomials in the equations

		monomial degrees				
E0 type	Uc[a] Kb[b]	1	2	3	4	unique monomials
E0*(3, 6, c, 14)	14	14	90.1	347.7	842.0	88.08%
E0(3, 6, c, 30)	15	15	104.2	1000.0	426.9	79.75%
E0(3, 6, c, 60)	16	16	117.9	1094.0	507.9	69.03%
E0(3, 6, 12, 60)	17	17	134.0	1509.0	622.8	71.08%
E0*(6, c, 14, 30)	18	18	152.2	2439.0	778.3	83.73%
E0(6, c, 14, 60)	19	19	170.1	2669.0	911.1	74.88%
E0(6, c, 30, 60)	20	20	189.0	3333.0	1069.6	74.46%
E0(6, 14, 30, 60)	21	21	209.4	1255.3	4364.0	77.53%
E0*(c, 14, 30, 60)	22	22	230.5	1468.0	5636.0	80.78%
E0(6, 14, 30, 110)	23	23	251.7	1639.8	5972.0	72.35%
E0(c, 14, 30, 110)	24	24	275.6	1890.9	7473.0	74.63%
E0(c, 14, 30, 204)	25	25	289.7	2068.0	9424.0	77.30%
⋮						
E0	128	128	8128.0	319807.0	8005426.0	75.64%

[a] Keystream bits
[b] Unknowns count

the memory used by the SAT solver does not surpass 1 GiB, which is why it is omitted from the statistics.

We examined the percentage of all monomials used within the equations using the same equations as above (as in Table 4.1). Within the few iterations, E0 and its small-scale variants can use most possible combinations of monomials. We show this in Table 4.2. We omitted the standard deviation from the table as it was close to zero for most values. The average is calculated from 15 random keystreams except for the full-scale version, where two were used. The percentage of all unique monomials used for one random keystream is calculated as

$$\frac{\#\text{unique monomials}}{\sum_{i=1}^{4} \binom{\text{Uc}}{i}} \cdot 100, \tag{4.1}$$

where Uc is the number of unknowns. Table 4.1 calculates the percentage of unique monomials as the average across all the random keystreams. Using Armknecht's results [20], we can improve the estimate of all possible monomial combinations to $2^{23.07}$, which gives us 94.63% instead of 75.64%. We could improve the estimates for the other small-scale variants as well.

Table 4.3: Determing the minimum number of bits to find one solution

			F4		SAT
E0 type	Uc[a]	Keystream bits	G. Basis Time (s)	Mem (GiB)	Time (s)
E0*(3, 6, c, 14)	14	50.5 ± 6.9	1.4 ± 0.3	0.1 ± 0.0	2.6 ± 0.4
E0(3, 6, c, 30)	15	58.7 ± 5.2	3.0 ± 0.3	0.2 ± 0.0	5.7 ± 0.4
E0(3, 6, c, 60)	16	60.5 ± 7.7	7.2 ± 1.9	0.5 ± 0.1	10.1 ± 0.8
E0(3, 6, 12, 60)	17	60.9 ± 4.7	27.5 ± 8.5	2.5 ± 0.9	27.5 ± 1.4
E0*(6, c, 14, 30)	18	65.2 ± 10.1	137.2 ± 45.1	6.8 ± 2.3	79.9 ± 14.0
E0(6, c, 14, 60)	19	67.1 ± 5.5	402.4 ± 35.5	15.1 ± 4.6	215.1 ± 31.1
E0(6, c, 30, 60)	20	73.5 ± 9.4	1208.2 ± 108.1	33.9 ± 7.8	487.2 ± 97.1

[a] Unknowns count

At the beginning of this section, we discussed what amount of keystream bits can be generated by each version of the small-scale variant of the cipher. Let us now assume that the number of keystream bits generated by the variants would scale with the number of all possible configurations for the initial states of the LFSRs (after the reinitialization phase). The maximum number of bits generated per frame for the full-scale version of E0 is 2745, while all possible initial values are 2^{128}. In logarithmic scale we have $\log_2 \frac{2745}{2^{128}} = -116.57$. If we wanted this to be the same for, e.g., the 14 unknown versions of E0, we would need to set the number of bits to be equal to $\frac{2745}{2^{114}}$. Meaning that the algorithm should not generate even one keystream bit[6]. A very rough estimate on the minimum number of bits required to solve the system is also given by Ars [31]: $2^n - 2$.

If the keystream bits were generated using a combination of the LFSRs only (without the FSM), we would only require n keystream bits, where n is the number of unknowns. This applies if we know the primitive polynomials used within the LFSRs. Without their knowledge, the number of required keystream bits to find the initial configuration is $2n$ and can be found using the Berlekamp–Massey algorithm [14]. Thus, in the next experiment, we first search for the minimum number of bits that give us exactly one solution. In Table 4.3, we display the number of minimum keystream bits to find a unique system solution. On average, the number of required keystream bits does not surpass $4n$, where n is the number of unknowns. However, the sample is too small to be statistically significant, and we cannot determine whether the relationship would hold for more unknowns. On the other hand, we can see the improvement in computational times for both methods, mainly for the F4 algorithm. We visualized the minimum number of bits according to the number of unknowns in Figure 4.1. The figure indicates the linear growth

[6]The first small-scale version of E0 would have to have 117 unknowns to be allowed to generate at least one bit.

of the minimum number of keystream bits necessary to solve the system. In total, we have five outliers for 105 runs in this case. It would be interesting to see if this trend holds for even more data.

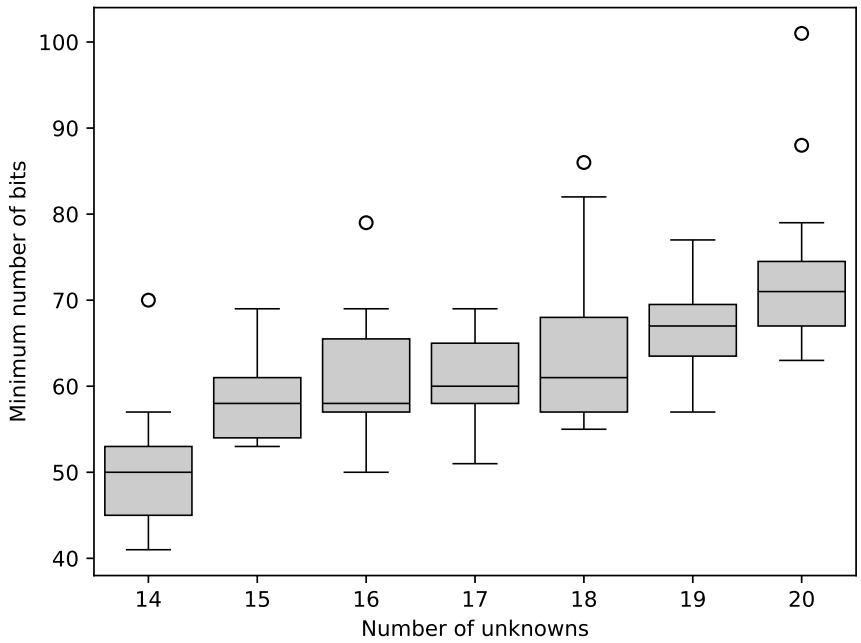


Figure 4.1: Minimum number of bits required to find one solution

We focused on the small-scale version of E0 with 20 unknowns – E0(`6`, `C`, `30`, `60`), and we run the experiments with a different number of keystream bits (increasing in multiples of 10). The results are shown in Table 4.4. The computational time results for the F4 algorithm do not change significantly once supplied with more than 60 bits. Notice that memory usage does not necessarily decrease with more keystream bits, which is probably caused by the new equations leading to different and more expensive computational steps. The computational times for the SAT solver increase after adding more than 40 keystream bits. CryptoMiniSat does not, by default, do any preprocessing of the equations[7]. Thus, in the case of the SAT solver, the increased times are reasonable, as the SAT solver needs to ensure that all clauses are satisfied. In Table 4.5, we examine the change of degrees of polynomials in the resulting Gröbner bases. We display the percentage of the degrees across all of the calculated Gröbner bases for the given number of keystream bits. The last column shows the average number of found solutions (affine varieties). Note

[7]Before compiling, one can turn on the Gauss-Jordan Elimination [32]. We did not test the performance of this option.

Table 4.4: Changing the number of keystream bits for E0(6, C, 30, 60)

		F4			SAT
E0 type	Kb[a]	G. Basis Time (s)	Variety Time (s)	Mem (GiB)	Time (s)
E0(6, c, 30, 60)	20	26400.3 ± 324.9	2077.6 ± 756.1	224.3 ± 2.5	624.7 ± 28.3
E0(6, c, 30, 60)	30	13503.2 ± 460.1	406.3 ± 1337.9	138.3 ± 0.1	522.2 ± 22.2
E0(6, c, 30, 60)	40	6757.9 ± 206.5	2.8 ± 3.2	87.2 ± 0.0	508.5 ± 18.1
E0(6, c, 30, 60)	50	6499.1 ± 205.1	0.1 ± 0.1	100.0 ± 0.0	549.9 ± 24.4
E0(6, c, 30, 60)	60	1376.7 ± 7.8	0.0 ± 0.0	43.1 ± 0.0	627.3 ± 59.4
E0(6, c, 30, 60)	70	1285.2 ± 16.1	0.0 ± 0.0	38.8 ± 0.0	566.1 ± 32.0
E0(6, c, 30, 60)	80	1041.6 ± 6.6	0.0 ± 0.0	26.8 ± 0.0	623.7 ± 55.2
E0(6, c, 30, 60)	90	1225.7 ± 33.5	0.0 ± 0.0	37.4 ± 0.0	697.5 ± 79.3
E0(6, c, 30, 60)	100	1078.9 ± 36.2	0.0 ± 0.0	15.4 ± 0.0	729.7 ± 90.9

[a] Keystream bits

Table 4.5: Degrees of polynomials in the resulting Gröbner bases for E0(6, c, 30, 60)

Kb[a]	deg 1	deg 2	deg 3	deg 4	deg 5	deg 6	solutions
20	-	-	-	0.13%	31.79%	68.08%	16374.2 ± 505.4
30	-	-	-	99.94%	0.06%	-	2200.1 ± 473.0
40	-	-	100.00%	-	-	-	256.9 ± 21.1
50	0.04%	99.63%	0.33%	-	-	-	30.5 ± 5.2
60	70.97%	29.03%	-	-	-	-	4.9 ± 1.5
70	99.66%	0.34%	-	-	-	-	1.7 ± 0.6
80	100.00%	-	-	-	-	-	1.1 ± 0.3
90	100.00%	-	-	-	-	-	1.1 ± 0.2
100	100.00%	-	-	-	-	-	1.1 ± 0.2

[a] Keystream bits

that for 20 unknowns, there are 2^{20} possible combinations. When ten more keystreams are added, the degree of the polynomials in the resulting Gröbner basis is, in most cases, reduced by one (except when moving from 50 to 60 keystream bits). After using more than 70 keystream bits, the degree of the found Gröbner basis is 1, which means that the number of found solutions must be close to 1.

For the next experiment, we will assume that the maximum number of keystream bits generated by the small-scale variants of E0 is linearly related to the number of unknowns. We allow the variants to generate at most $\lfloor \mathrm{Uc} \cdot \frac{2745}{128} \rfloor$ (Uc is the number of unknowns) keystream bits, which are then used in the equations. We show the results in Table 4.6. The F4 algorithm efficiently utilized the new equations and significantly reduced the computational times.

Table 4.6: Using as many keystream bits as possible

			F4		SAT
E0 type	Uc[a]	Kb[b]	G. Basis Time (s)	Mem (MiB)	Time (s)
E0*(3, 6, c, 14)	14	300	0.1 ± 0.0	32.1 ± 0.0	1.7 ± 0.2
E0(3, 6, c, 30)	15	321	0.3 ± 0.0	64.1 ± 0.0	3.7 ± 1.2
E0(3, 6, c, 60)	16	343	1.6 ± 0.0	64.1 ± 0.0	17.5 ± 1.6
E0(3, 6, 12, 60)	17	364	3.0 ± 0.0	192.2 ± 0.0	58.3 ± 4.4
E0*(6, c, 14, 30)	18	386	10.3 ± 0.1	1257.3 ± 0.0	140.9 ± 14.3
E0(6, c, 14, 60)	19	407	20.0 ± 0.2	2091.3 ± 0.0	453.1 ± 30.8
E0(6, c, 30, 60)	20	428	36.7 ± 0.3	4302.8 ± 0.0	653.8 ± 71.8
E0(6, 14, 30, 60)	21	450	79.8 ± 0.3	7539.4 ± 0.0	1334.8 ± 110.7
E0*(c, 14, 30, 60)	22	471	178.5 ± 1.2	9462.8 ± 0.0	3678.7 ± 281.5
E0(6, 14, 30, 110)	23	493	1320.6 ± 73.2	12012.4 ± 0.0	7257.0 ± 1022.4

[a] Unknowns count
[b] Keystream bits

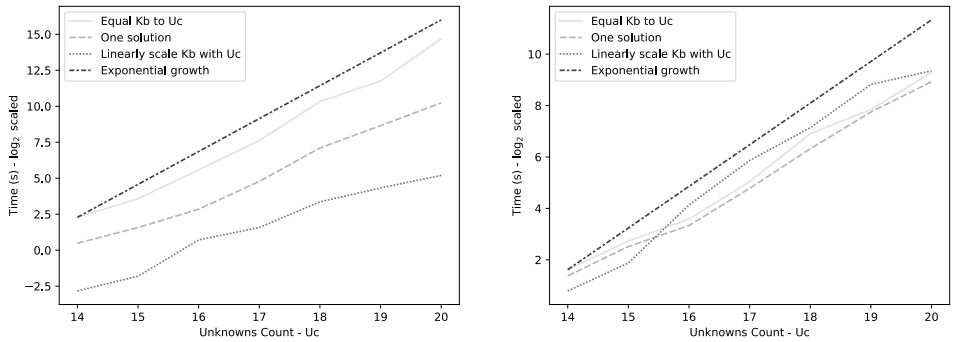
For E0(6, C, 30, 60) with 20 unknowns, the computation required around 7 hours on average when only 20 keystream bits were used (see Table 4.1). With 428 keystream bits, the F4 algorithm requires around 36 seconds on average (approximately 733x faster with only 21x more bits). We added three more small-scale versions of E0 to see whether using more keystream bits would resolve in lower computational times for different small-scale E0 variants with more unknowns. Although the versions with 21 and 22 unknowns benefit from the additional keystream bits, the version with 23 unknowns significantly increases computational time. The SAT solver's time increased compared to the results in Table 4.1, which may be associated with the missing preprocessing we discussed earlier in this chapter.

In Figure 4.2, we compare the computational times of the F4 algorithm and CryptoMiniSat's SAT solver based on the number of keystream bits. As we have already indicated, the F4 algorithm can use more keystream bits (hence equations) to speed up the computation. On the other hand, the SAT solver does not seem to benefit from the larger portion of the data without any preprocessing.

4.2 Using Local Sensitivity Hashing For Reduction

Following the works [21, 22], we employed the Local Sensitivity Hashing (**LSH**) method to reduce the number of monomials in polynomials (see Section 2.3). We will use the equations from the experiments in Table 4.6.

As described in Section 2.3, our version allows us to pick equations that contain fewer monomials than a given threshold t multiplied by the mean



(a) Gröbner basis – F4 algorithm (b) SAT solver

Figure 4.2: Comparison of computational times required based on the number of keystream bits (Kb)

Table 4.7: Using LSH with $t = 0.5$ on equations from Table 4.6

		F4		SAT
		G. Basis		
E0 type	Uc[a]	Time (s)	Mem (GiB)	Time (s)
E0*(3, 6, c, 14)	14	4.4 ± 0.5	0.3 ± 0.0	1.2 ± 0.1
E0(3, 6, c, 30)	15	1.0 ± 0.2	0.1 ± 0.0	1.2 ± 0.3
E0(3, 6, c, 60)	16	25.8 ± 6.4	1.4 ± 0.4	5.5 ± 0.5
E0(3, 6, 12, 60)	17	98.5 ± 18.4	3.9 ± 1.4	18.6 ± 1.9
E0*(6, c, 14, 30)	18	254.0 ± 50.6	8.9 ± 1.5	32.7 ± 2.1
E0(6, c, 14, 60)	19	964.0 ± 647.5	22.7 ± 12.9	85.3 ± 6.9
E0(6, c, 30, 60)	20	1355.5 ± 173.6	26.5 ± 9.5	544.5 ± 38.9

[a] Unknowns count

number of monomials in polynomials for a given keystream. For this experiment, we are using 20 buckets and 100 random permutations. Using LSH, we aim to decrease the computational times for the SAT solver. We tested three different thresholds ($t \in \{0.5, 0.55, 0.6\}$), out of which the threshold $t = 0.5$ gave us the best results for the SAT solver regarding the computational time. The results for $t = 0.5$ are displayed in Table 4.7, and the results for $t = 0.55$ and $t = 0.6$ are displayed in Table C.3 and Table C.5 in Appendix. The computational times for the SAT solver decreased compared to the results in Table 4.6, although the small-scale variant with 20 unknowns seems to require the use of fewer equations. Table 4.8 shows how much the polynomials were reduced on average. In most cases, the condition with the threshold $t = 0.5$

Table 4.8: Polynomials and monomials after LSH with $t = 0.5$ on equations from Table 4.6

	⌀ #polynomials		⌀ #monomials in polynomials	
E0 type	Before LSH	After LSH	Before LSH	After LSH
E0*(3, 6, c, 14)	300	67.8 ± 35.2	375.7 ± 67.9	126.4 ± 65.5
E0(3, 6, c, 30)	321	248.4 ± 31.0	462.7 ± 70.5	175.7 ± 91.6
E0(3, 6, c, 60)	343	267.3 ± 14.2	559.9 ± 48.2	216.0 ± 96.7
E0(3, 6, 12, 60)	364	328.7 ± 19.5	683.7 ± 55.6	277.3 ± 113.5
E0*(6, c, 14, 30)	386	402.3 ± 9.0	828.0 ± 82.6	361.3 ± 143.1
E0(6, c, 14, 60)	407	375.3 ± 5.0	961.2 ± 65.8	412.5 ± 148.7
E0(6, c, 30, 60)	428	668.3 ± 19.8	1181.9 ± 242.2	555.9 ± 247.0

⌀ average

generated fewer equations than the initial number, except for E0(6, c, 14, 30) and E0(6, c, 30, 60). For the latter, the average number of equations increased by approximately 55%, which may explain the increase in computational time for the SAT solver. It may be useful to limit the number of equations as well.

4.3 Searching For Additional Equations With Certain Probability

In Section 2.2.5, we described how we could derive additional equations. In this section, we will verify whether it is possible to derive any equations using the keystream bits. In the Jupyter Notebook `Possible States`, we used `Python` to search for valid configurations recursively, and we built matrices $\mathbf{B}$ and $\mathbf{C}$ specified in Section 2.2.5 using `numpy` (mainly `np.prod` and `np.sum`). Note that this task is well-parallelizable.

As stated in Section 2.2.5, we could not find any equations for $\ell = 3, k = 4$ and $l = 2$. The `Python` implementation is cumbersome, even though we try to use the most effective methods available. The upcoming research aims to reimplement this part using `C++`, allowing us to work with the arrays more efficiently. Although we were unable to do further calculations, we calculated the number of unique states for $4, 5, 6, 7$ and 8 consecutive keystream bits[8] in Table 4.9. The number of expected states is 2^{4c}, where c is the number of consecutive keystream bits. Adding more consecutive keystream bits shows that the number of unique states does not grow as quickly as expected. After

[8]The number of unique states for $1, 2$, and 3 consecutive keystream bits equals the number of expected states.

Table 4.9: Unique number of states for a given number of consecutive keystream bits

Consecutive bits	4	5	6	7	8
Expected states	65 536	1 048 576	16 777 216	268 435 456	4 294 967 296
Approximate growth	65 536	524 288	4 194 304	33 554 432	268 435 456
Unique states	53 248	487 424	4 083 712	33 230 848	267 440 128

four successive keystream bits, the number of unique states approximately grows with $2^{16}2^{3(c-4)}$ (Approximate growth in Table 4.9).

The alternative to searching for the equations is to search for equations that are true with a certain probability. That is, for $\ell+1$ consecutive bits, we search for equations that would hold with probability p, such that $p^{c-\ell} > P$, where c is the number of keystream bits, and P is the probability that all the keystream bits must hold. For example, if $P = 0.99$, $c = 14$ and $\ell = 3$, we would require $p = \sqrt[c-\ell]{P} = \sqrt[11]{0.99} \approx 0.999$. Using $\ell = 3, k = 4$ and $l = 2$, we found equations with probability $p = 0.96$, which does not satisfy our requirements that $p^{11} > 0.99$ ($p^{11} \approx 0.63$). For four consecutive keystream bits $(z_t, 0, z_{t+2}, z_{t+3})$, we found 49 monomials of degree 4 and 436 polynomials of degree 4 with two terms. For $(z_t, 1, z_{t+2}, z_{t+3})$ there exists only one monomial of degree 4.

Chapter 5

Related Works

Over the course of the existence of the E0 cipher, many analyses were conducted together with various attack techniques that generally show that E0 is not a safe cipher. In this chapter we briefly review some of the works regarding E0, with the focus on algebraic attacks.

One of the main personalities involved in E0 research was Frederik Armknecht. Armknecht used the idea of algebraic attacks combined with linearization. In his first work [33], Armknecht removed the memory bits out of the equations, and made an estimate that the total number of distinct monomials would be approximately $2^{24.0569}$. The linearization attack would require an equal amount of bits. Armknecht improved the estimation on the number of bits to $2^{23.07}$ in his follow-up research [20], while using an improved version of Courtois [34] algebraic attacks, called fast algebraic attacks. Armknecht showed, that no equations of degree lower than four exist for four and five consecutive keystream bits (see Section 2.2.4).

Armknecht [31] also investigated the use of Gröbner bases on LFSRs with simple combiners. He compared the computation of Gröbner basis (using the F5 algorithm [35]) with correlation attacks and searched for the minimum number of outputs require to find a unique solution of the system.

A different way to attack E0 is through using Ordered Binary Decision Diagram (**OBDD**) [36], using a short keystream attack (in case of E0, only 128 are sufficient). Using Binary Decision Diagrams (**BDD**s), we can represent a boolean function as acyclic graph, where the leaf vertices are considered sinks, and are unique for each assignment of the boolean function. The estimated time complexity is 2^{87} and the authors claim the task is massively parallelizable with very low memory requirements (2^{23}).

The most recent research on E0 was done by Scala [37], using guess-and-determine technique. Scala et al. used 14 special variables, which, after eval-

[9] We discussed this in Section 4.1, the maximum number of monomials of degree 4, without considering the properties of E0, is $\sum_{i=1}^{4} \binom{128}{i} \approx 2^{23.393}$.

uation, led to linear relations among other variables. Then, with 83 more bits guessed, they performed an algebraic cryptanalysis, employing both Gröbner basis and SAT solvers. Scala et al. estimated the complexity of the attack to 2^{79} seconds.

In addition to the attacks already mentioned, Lu et al. [38] performed a correlation analysis of analysis, which assumes that E0 would be used outside of the Bluetooth environment (without the limitation of 2745 keystream bits per frame). The attack uses some weak of the weak statistical properties of the FSM used in E0, resulting in attack of complexity 2^{39} with 2^{39} bits required.

Conclusion

The theoretical part of this work outlined the most important mathematical concepts necessary to understand the underlying theory used in the experimental part. Furthermore, we analyzed the inner workings of the E0 cipher, mathematically formulated the initialization, and designed small-scale cipher variants using linear feedback shift registers with feedback primitive polynomials of lower degrees. To be more specific, the small-scale variants of the cipher were designed using four LFSRs of lower lengths.

We suggested an exhaustive search of new equations dependent on specific keystream bits. This part deserves more attention in the future, as the current environment does not allow us to scale appropriately for more bits. In the experimental part, we suggested using additional equations that hold with a certain probability. This leaves us with an open problem of whether more probable equations exist and how to recover if the found solution is wrong.

Using our described implementation, we run several experiments, testing how different amounts of keystream bits (hence equations) influence the solvers (F4 algorithm and SAT solver). We have found that using additional keystream bits improves the performance of the F4 algorithm and that a linear relation may exist between the number of required keystream bits and a unique solution. For the 20-bit version of E0, using 428 equations resulted in an improvement of the computational time of the F4 algorithm from 26 400.3 seconds to 36.7 seconds. The SAT solver did not benefit from the more significant number of equations; for this reason, we have used a reduction method (local sensitivity hashing) and improved the run times of the SAT solver. For the 19-bit E0 version, we reduced the computational time from 453.1 seconds to 85.3 seconds. However, a comparison needs to be made between this approach and Gauss elimination.

Lastly, the formulation of the cipher we used has not been previously used for the Guess-and-determine approach. To use the approach, we will need to identify such variables whose guessing will lead to an overall improvement of the computation. Lastly, since the implementation is mainly done in `Sage`,

we will need to re-implement some of the parts for better performance since `Sage` was not designed for tasks of this scale.

Bibliography

1. CHRIS VALLANCE. UK amends encrypted message scanning plans. *BBC News* [online]. 2023 [visited on 2024-01-09]. Available from: https://www.bbc.com/news/technology-66240006.
2. DANIEL BOFFEY. EU lawyers say plan to scan private messages for child abuse may be unlawful. *The Guardian* [online]. 2023 [visited on 2024-01-09]. Available from: https://www.theguardian.com/world/2023/may/08/eu-lawyers-plan-to-scan-private-messages-child-abuse-may-be-unlawful-chat-controls-regulation.
3. PIETER HARTEL; ROLF VAN WEGBERG. Going dark? Analysing the impact of end-to-end encryption on the outcome of Dutch criminal court cases. *Crime Science*. 2023, vol. 12. Available from DOI: https://doi.org/10.1186/s40163-023-00185-4.
4. MORRIS J. DWORKIN; ELAINE B. BARKER; JAMES R. NECHVATAL; JAMES FOTI; LAWRENCE E. BASSHAM; E. ROBACK; JAMES F. DRAY JR. *Advanced Encryption Standard (AES)*. NIST, 2001. Available from DOI: https://doi.org/10.6028/NIST.FIPS.197.
5. REZOS; KRISTENS; KINGTHORIN. *Cryptanalysis* [online]. [N.d.]. [visited on 2024-01-09]. Available from: https://owasp.org/www-community/attacks/Cryptanalysis.
6. BLUETOOTH SIG. *Bluetooth Core Specification* [online]. 2021. [visited on 2022-09-08]. Available from: https://www.bluetooth.com/specifications/specs/core-specification-5-3/.
7. BRUNO BUCHBERGER. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem Nulldimensionalen Polynomideal* [online]. Austria, 1965 [visited on 2023-11-17]. PhD thesis. Johannes Kepler University of Linz.

8. BODO RENSCHUCH; HARTMUT ROLOFF; GEORGIJ G. RASPUTIN; MICHAEL ABRAMSON; ET AL. Contributions to constructive polynomial ideal theory XXIII: forgotten works of Leningrad mathematician N. M. Gjunter on polynomial ideal theory [online]. 2003, vol. 37, no. 2 [visited on 2023-11-17]. ISSN 0163-5824. Available from DOI: 10.1145/944567.944569.
9. BRUNO BUCHBERGER; FRANZ WINKLER. *Gröbner Bases and Applications*. Cambridge University Press, 1998. London Mathematical Society Lecture Note Series, no. 251. ISBN 978-0-521-63298-0.
10. DAVID A. COX; JOHN LITTLE; DONAL O'SHEA. *Ideals, Varieties, and Algorithms*. Springer, 2015. Undergraduate Texts in Mathematics, no. 666. ISBN 978-3-319-16720-6.
11. TAKAYUKI HIBI. *Gröbner Bases*. Springer Japan, 2013. ISBN 978-4-431-54573-6.
12. WILLIAM W. ADAMS; PHILIPPE LOUSTAUNAU. *An Introduction to Gröbner Bases*. Vol. 3. American Mathematical Society, 1996. Graduate Studies in Mathematics. ISBN 0-8218-3804-0.
13. RUDOLF LIDL; HARALD NIEDERREITER. *Introduction to finite fields and their applications*. Cambridge University Press, 1986. ISBN 0-521-30706-6.
14. ANNE CANTEAUT. *LFSR-based Stream Ciphers* [online]. 2016. [visited on 2022-08-09]. Available from: https://www.rocq.inria.fr/secret/Anne.Canteaut/MPRI/chapter3.pdf.
15. LEONID POSITSELSKI. *Semi-Infinite Algebraic Geometry* [online]. Prague, 2015 [visited on 2024-01-06]. Available from: https://users.math.cas.cz/~positselski/semi-inf.pdf.
16. MARIE-MADELEINE RENAULD. *The Collector* [online]. 2023. [visited on 2023-11-04]. Available from: https://www.thecollector.com/harald-bluetooth-viking-technology/.
17. BLUETOOTH SIG. *Origin of the Bluetooth name* [online]. [N.d.]. [visited on 2023-11-04]. Available from: https://www.bluetooth.com/about-us/bluetooth-origin/.
18. PHIL KOOPMAN. *Maximal Length LFSR Feedback Terms* [online]. [N.d.]. [visited on 2023-01-05]. Available from: https://users.ece.cmu.edu/~koopman/lfsr/.
19. DEVELOPERS, The SageMath. *sagemath/sage: 9.7*. Zenodo, 2022. Version 9.7. Available from DOI: 10.5281/zenodo.7132659.

20. FREDERIK ARMKNECHT; MATTHIAS KRAUSE. Algebraic Attacks on Combiners with Memory. In: *Advances in Cryptology - CRYPTO 2003*. SPRINGER, 2003, pp. 162–175. ISBN 978-3-540-45146-4. Available from DOI: 10.1007/978-3-540-45146-4_10.

21. JANA BERUŠKOVÁ. *Reducing Overdefined Systems of Polynomial Equations Derived from Small Scale Variants of the AES*. Prague, 2023. Available also from: http://hdl.handle.net/10467/107072. MA thesis. FIT CTU.

22. DANIEL MINAROVIČ. *Algebraic Cryptanalysis of Small Scale Variants of the Grain-128AEAD cipher*. Prague, 2023. Available also from: http://hdl.handle.net/10467/108972. MA thesis. FIT CTU.

23. WANG, Jingdong; SHEN, Heng Tao; SONG, Jingkuan; JI, Jianqiu. Hashing for Similarity Search: A Survey. *CoRR*. 2014, vol. abs/1408.2927. Available from arXiv: 1408.2927.

24. PINECONE. *Locality Sensitive Hashing (LSH): The Illustrated Guide* [online]. [N.d.]. [visited on 2024-01-11]. Available from: https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing/.

25. BOSMA, WIEB; CANNON, JOHN; PLAYOUST, CATHERINE. The Magma Algebra System I: The User Language. *Journal of Symbolic Computation*. 1997, vol. 24, no. 3, pp. 235–265. ISSN 0747-7171. Available from DOI: https://doi.org/10.1006/jsco.1996.0125.

26. ARNAULD DELMAS; RABASGREGORY. *e0*. 2022. Available also from: https://github.com/adelmas/e0.

27. JAMES BRIGGS. *testing_lsh.ipynb* [online]. 2023. [visited on 2023-12-24]. Available from: https://github.com/pinecone-io/examples/blob/master/learn/search/faiss-ebook/locality-sensitive-hashing-traditional/testing_lsh.ipynb.

28. SOOS, Mate; NOHL, Karsten; CASTELLUCCIA, Claude. Extending SAT Solvers to Cryptographic Problems. In: KULLMANN, Oliver (ed.). *Theory and Applications of Satisfiability Testing - SAT 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 244–257. ISBN 978-3-642-02777-2.

29. MAREK BIELIK. *Algebraic Cryptanalysis of Small Scale Variants of the AES*. Prague, 2021. Available also from: http://hdl.handle.net/10467/94634. MA thesis. FIT CTU.

30. THE SAGE DEVELOPMENT TEAM. *An ANF to CNF Converter using a Dense/Sparse Strategy* [online]. 2023. [visited on 2023-07-07]. Available from: https://doc.sagemath.org/html/en/reference/sat/sage/sat/converters/polybori.html.

31. FREDERIK ARMKNECHT; GWENOLÉ ARS. *Algebraic Attacks on Stream Ciphers with Gröbner Bases*. Berlin: Springer, 2009. ISBN 978-3-540-93805-7. Available also from: https://doi.org/10.1007/978-3-540-93806-4_18.

32. MATE SOOT. *CryptoMiniSat FAQ* [online]. [N.d.]. [visited on 2024-01-05]. Available from: https://www.msoos.org/cryptominisat-faq/.

33. FREDERIK ARMKNECHT. *A Linearization Attack on the Bluetooth Key Stream Generator*. 2002. Available also from: https://eprint.iacr.org/2002/191.

34. NICOLAS T. COURTOIS; WILLI MEIER. Algebraic Attacks on Stream Ciphers with Linear Feedback. *Springer*. 2003. Available from DOI: https://doi.org/10.1007/3-540-39200-9_21.

35. JEAN-CHARLES FAUGÈRE. *F4 algorithm, F5 algorithm*. Austria, 2013. Available also from: https://www-polsys.lip6.fr/~jcf/Papers/f4f5-gbrela-2013.pdf.

36. YANIV SHAKED; AVISHAI WOOL. Cryptanalysis of the Bluetooth E0 Cipher Using OBDD's. In: Berlin: Springer, [n.d.], vol. 4176, pp. 187–202. ISBN 978-3-540-38341-3. Available from DOI: 10.1007/11836810_14.

37. ROBERTO LA SCALA; SERGIO POLESE; SHARWAN K. TIWARI; ANDREA VISCNOTI. *An algebraic attack to the Bluetooth stream cipher E0* [online]. 2022. [visited on 2022-12-31]. Available from: https://eprint.iacr.org/2022/016.pdf.

38. YI LU; SERGE VAUDENAY. Faster Correlation Attack on Bluetooth Keystream Generator E0. In: *CRYPTO 2004*. Berlin: Springer, 2004. ISBN 978-3-540-22668-0. Available from DOI: https://doi.org/10.1007/978-3-540-28628-8_25.

39. ARMKNECHT, Frederik. Algebraic Attacks on Stream Ciphers. In: NEITTAANMÄKI, Pekka (ed.). *Proceedings / ECCOMAS 2004, 4th European Congress on Computational Methods in Applied Sciences and Engineering : Jyväskylä, Finland, 24 - 28 July 2004*. Jyväskylä: Univ. of Jyväskylä, 2004. Available also from: https://madoc.bib.uni-mannheim.de/5535/. CD-ROM. - Vol. 1: Plenary sessions, invited parallel sessions, contributed sessions and posters. - Vol. 2: Minisymposia and special technology sessions.

Appendix A

Acronyms

AES Advanced Encryption Standard

ANF Algebraic Normal Form

BDD Binary Decision Diagram

BR/EDR Bluetooth Basis Rate/Enhanced Data Rate

CLK Central Device Clock

CNF Conjunctive Normal Form

E0 Encryption Algorithm From The Bluetooth Standard

E3 Hash Function Used Together With The E0 Algorithm

FSM Finite State Machine

GB Gröbner Basis (or Bases)

GF Galois Field

HW Hamming Weight

LFSR Linear Feedback Shift Register

LSH Local Sensitivity Hashing

OBDD Ordered Binary Decision Diagram

Appendix B

Contents of enclosed SD Card

```
README.md ..... Description of the folder
src
    README.md ..... Description of the folder
    e0 ..... c implementation of the cipher
    generator ..... Tools used for the algebraic analysis
    text ..... Source code for the thesis
text
    MT_Dolejs_Jan.pdf ..... Master thesis
```

Appendix C

Additional Content

C.1 Polynomials Used For Transformation of K_{enc}

Table C.1: Polynomials used for K_{enc}

L	$\deg(g_1^{(L)})$	$g_1^{(L)}$	$\deg(g_2^{(L)})$	$g_2^{(L)}$
1	8	0x0000000000000000000000000000011d	119	0x00e275a0abd218d4cf928b9bbf6cb08f
2	16	0x0000000000000000000000000001003f	112	0x0001e3f63d7659b37f18c258cff6efef
3	24	0x000000000000000000000000010000db	104	0x000001bef66c6c3ab1030a5a1919808b
4	32	0x00000000000000000000000001000000af	96	0x000000016ab89969de17467fd3736ad9
5	40	0x0000000000000000000001000000039	88	0x0000000000163063291da50ec55715247
6	48	0x000000000000000000010000000.04591	77	0x0000000000.045c9352aa6cc054468311
7	56	0x00000000000000000100000000000095	71	0x00000000000000b3f7fffce279f3a073
8	64	0x0000000000000001000000000000001b	63	0x0000000000000000a1ab815bc7ec8025
9	72	0x000000000000010000000000000000609	49	0x00000000000000000.045c98011d8b04d
10	80	0x00000000001000000000000000.04515	42	0x0000000000000000000058e24f9a4bb
11	88	0x0000000010000000000000000013b	35	0x0000000000000000000000ca76024d7
12	96	0x000000010000000000000000000000dd	28	0x00000000000000000000001c9c26b9
13	104	0x00001000000000000000000000.0459d	21	0x000000000000000000000000.0456d9e3
14	112	0x0001000000000000000000000000014f	14	0x00000000000000000000000000.045377
15	120	0x010000000000000000000000000000e7	7	0x0000000000000000000000000000089
16	128	0x10000000000000000000000000000000	0	0x00000000000000000000000000000001

C.2 Finding Algebraic Representation Of q_{t+1} and p_{t+1} Over $\mathbb{F}_2$

In this section, we will derive an algebraic representation for q_{t+1} and p_{t+1} over $\mathbb{F}_2$. In Section 2.2.1 all of the bits $x_t^{(4)}$, $x_t^{(3)}$, $x_t^{(2)}$, $x_t^{(1)}$, q_t, p_t, q_{t-1} and p_{t-1} are combined to get q_{t+1} and p_{t+1}. We can evaluate the output bits for all possible combinations; we show this in Table C.2. Using the truth table, we can search for the ANF using, for example, `sage`. We will derive the following equations:

$$\begin{aligned} q_{t+1} = {} & x_t^{(1)}x_t^{(2)}x_t^{(3)}x_t^{(4)} \oplus \\ & (x_t^{(1)}x_t^{(2)}x_t^{(3)} \oplus x_t^{(1)}x_t^{(2)}x_t^{(4)} \oplus x_t^{(1)}x_t^{(3)}x_t^{(4)} \oplus \\ & x_t^{(2)}x_t^{(3)}x_t^{(4)})p_t \oplus (x_t^{(1)}x_t^{(2)} \oplus x_t^{(1)}x_t^{(3)} \oplus x_t^{(1)}x_t^{(4)} \oplus \\ & x_t^{(2)}x_t^{(3)} \oplus x_t^{(2)}x_t^{(4)} \oplus x_t^{(3)}x_t^{(4)})q_t \oplus \\ & (x_t^{(1)} \oplus x_t^{(2)} \oplus x_t^{(3)} \oplus x_t^{(4)})p_t q_t \oplus q_t \oplus p_{t-1} \\ = {} & \pi_t^4 \oplus \pi_t^3 p_t \oplus \pi_t^2 q_t \oplus \pi_t^1 p_t q_t \oplus q_t \oplus p_{t-1} \end{aligned} \tag{C.1}$$

$$\begin{aligned} p_{t+1} = {} & x_t^{(1)}x_t^{(2)} \oplus x_t^{(1)}x_t^{(3)} \oplus x_t^{(1)}x_t^{(4)} \oplus x_t^{(2)}x_t^{(3)} \oplus x_t^{(2)}x_t^{(4)} \oplus x_t^{(3)}x_t^{(4)} \oplus \\ & (x_t^{(1)} \oplus x_t^{(2)} \oplus x_t^{(3)} \oplus x_t^{(4)})p_t \oplus q_t \oplus q_{t-1} \oplus p_t \oplus p_{t-1} \\ = {} & \pi_t^2 \oplus \pi_t^1 p_t \oplus q_t \oplus q_{t-1} \oplus p_{t-1} \oplus p_t \end{aligned} \tag{C.2}$$

We used the definition of a symmetric polynomial (see Definition 23) to shorten the notation.

Table C.2: Calculating Values Of p_{t+1} and c_{t+1}

$x_t^{(4)}$	$x_t^{(3)}$	$x_t^{(2)}$	$x_t^{(1)}$	q_t	p_t	q_{t-1}	p_{t-1}	q_{t+1}	p_{t+1}	$x_t^{(4)}$	$x_t^{(3)}$	$x_t^{(2)}$	$x_t^{(1)}$	q_t	p_t	q_{t-1}	p_{t-1}	q_{t+1}	p_{t+1}
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	0	1
0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	1	1	1	0
0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	1	1	0	1	0	0	0	0	1	0	1	1	1
0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	1	1	0	0	1
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	1	1	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	0	0	1	0	0	0	1	1
0	0	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0
0	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0

Continued on next page

Table C.2 – continued from previous page

$x_t^{(4)}$	$x_t^{(3)}$	$x_t^{(2)}$	$x_t^{(1)}$	q_t	p_t	q_{t-1}	p_{t-1}	q_{t+1}	p_{t+1}	$x_t^{(4)}$	$x_t^{(3)}$	$x_t^{(2)}$	$x_t^{(1)}$	q_t	p_t	q_{t-1}	p_{t-1}	q_{t+1}	p_{t+1}
0	0	0	0	1	0	1	1	0	1	1	0	0	0	1	0	1	1	0	1
0	0	0	0	1	1	0	0	1	0	1	0	0	0	1	1	0	0	0	1
0	0	0	0	1	1	0	1	0	1	1	0	0	0	1	1	0	1	1	0
0	0	0	0	1	1	1	0	1	1	1	0	0	0	1	1	1	0	0	0
0	0	0	0	1	1	1	1	0	0	1	0	0	0	1	1	1	1	1	1
0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1
0	0	0	1	0	0	0	1	1	1	1	0	0	1	0	0	0	1	1	0
0	0	0	1	0	0	1	0	0	1	1	0	0	1	0	0	1	0	0	0
0	0	0	1	0	0	1	1	1	0	1	0	0	1	0	0	1	1	1	1
0	0	0	1	0	1	0	0	0	0	1	0	0	1	0	1	0	0	0	0
0	0	0	1	0	1	0	1	1	1	1	0	0	1	0	1	0	1	1	1
0	0	0	1	0	1	1	0	0	1	1	0	0	1	0	1	1	0	0	1
0	0	0	1	0	1	1	1	1	0	1	0	0	1	0	1	1	1	1	0
0	0	0	1	1	0	0	0	1	1	1	0	0	1	1	0	0	0	0	0
0	0	0	1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1
0	0	0	1	1	0	1	0	1	0	1	0	0	1	1	0	1	0	0	1
0	0	0	1	1	0	1	1	0	1	1	0	0	1	1	0	1	1	1	0
0	0	0	1	1	1	0	0	0	1	1	0	0	1	1	1	0	0	0	1
0	0	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	1	0
0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	1	0	0	0
0	0	0	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1
0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1
0	0	1	0	0	0	0	1	1	1	1	0	1	0	0	0	0	1	1	0
0	0	1	0	0	0	1	0	0	1	1	0	1	0	0	0	1	0	0	0
0	0	1	0	0	0	1	1	1	0	1	0	1	0	0	0	1	1	1	1
0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0	0	0	0
0	0	1	0	0	1	0	1	1	1	1	0	1	0	0	1	0	1	1	1
0	0	1	0	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1
0	0	1	0	0	1	1	1	1	0	1	0	1	0	0	1	1	1	1	0
0	0	1	0	1	0	0	0	1	1	1	0	1	0	1	0	0	0	0	0
0	0	1	0	1	0	0	1	0	0	1	0	1	0	1	0	0	1	1	1
0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	1
0	0	1	0	1	0	1	1	0	1	1	0	1	0	1	0	1	1	1	0
0	0	1	0	1	1	0	0	0	1	1	0	1	0	1	1	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	0	1	0	1	1	0	1	1	0
0	0	1	0	1	1	1	0	0	0	1	0	1	0	1	1	1	0	0	0
0	0	1	0	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1
0	0	1	1	0	0	0	0	0	1	1	0	1	1	0	0	0	0	0	1
0	0	1	1	0	0	0	1	1	0	1	0	1	1	0	0	0	1	1	0

Continued on next page

Table C.2 – continued from previous page

$x_t^{(4)}$	$x_t^{(3)}$	$x_t^{(2)}$	$x_t^{(1)}$	q_t	p_t	q_{t-1}	p_{t-1}	q_{t+1}	p_{t+1}	$x_t^{(4)}$	$x_t^{(3)}$	$x_t^{(2)}$	$x_t^{(1)}$	q_t	p_t	q_{t-1}	p_{t-1}	q_{t+1}	p_{t+1}
0	0	1	1	0	0	1	0	0	0	1	0	1	1	0	0	1	0	0	0
0	0	1	1	0	0	1	1	1	1	1	0	1	1	0	0	1	1	1	1
0	0	1	1	0	1	0	0	0	0	1	0	1	1	0	1	0	0	1	1
0	0	1	1	0	1	0	1	1	1	1	0	1	1	0	1	0	1	0	0
0	0	1	1	0	1	1	0	0	1	1	0	1	1	0	1	1	0	1	0
0	0	1	1	0	1	1	1	1	0	1	0	1	1	0	1	1	1	0	1
0	0	1	1	1	0	0	0	0	0	1	0	1	1	1	0	0	0	0	0
0	0	1	1	1	0	0	1	1	1	1	0	1	1	1	0	0	1	1	1
0	0	1	1	1	0	1	0	0	1	1	0	1	1	1	0	1	0	0	1
0	0	1	1	1	0	1	1	1	0	1	0	1	1	1	0	1	1	1	0
0	0	1	1	1	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0
0	0	1	1	1	1	0	1	1	0	1	0	1	1	1	1	0	1	1	1
0	0	1	1	1	1	1	0	0	0	1	0	1	1	1	1	1	0	0	1
0	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0
0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	1	1	0
0	1	0	0	0	0	1	0	0	1	1	1	0	0	0	0	1	0	0	0
0	1	0	0	0	0	1	1	1	0	1	1	0	0	0	0	1	1	1	1
0	1	0	0	0	1	0	0	0	0	1	1	0	0	0	1	0	0	0	0
0	1	0	0	0	1	0	1	1	1	1	1	0	0	0	1	0	1	1	1
0	1	0	0	0	1	1	0	0	1	1	1	0	0	0	1	1	0	0	1
0	1	0	0	0	1	1	1	1	0	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	0	0	0	1	1	1	1	0	0	1	0	0	0	0	0
0	1	0	0	1	0	0	1	0	0	1	1	0	0	1	0	0	1	1	1
0	1	0	0	1	0	1	0	1	0	1	1	0	0	1	0	1	0	0	1
0	1	0	0	1	0	1	1	0	1	1	1	0	0	1	0	1	1	1	0
0	1	0	0	1	1	0	0	0	1	1	1	0	0	1	1	0	0	0	1
0	1	0	0	1	1	0	1	1	0	1	1	0	0	1	1	0	1	1	0
0	1	0	0	1	1	1	0	0	0	1	1	0	0	1	1	1	0	0	0
0	1	0	0	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1
0	1	0	1	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	1
0	1	0	1	0	0	0	1	1	0	1	1	0	1	0	0	0	1	1	0
0	1	0	1	0	0	1	0	0	0	1	1	0	1	0	0	1	0	0	0
0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	1	1	1
0	1	0	1	0	1	0	0	0	0	1	1	0	1	0	1	0	0	1	1
0	1	0	1	0	1	0	1	1	1	1	1	0	1	0	1	0	1	0	0
0	1	0	1	0	1	1	0	0	1	1	1	0	1	0	1	1	0	1	0
0	1	0	1	0	1	1	1	1	0	1	1	0	1	0	1	1	1	0	1
0	1	0	1	1	0	0	0	0	0	1	1	0	1	1	0	0	0	0	0

Continued on next page

Table C.2 – continued from previous page

$x_t^{(4)}$	$x_t^{(3)}$	$x_t^{(2)}$	$x_t^{(1)}$	q_t	p_t	q_{t-1}	p_{t-1}	q_{t+1}	p_{t+1}	$x_t^{(4)}$	$x_t^{(3)}$	$x_t^{(2)}$	$x_t^{(1)}$	q_t	p_t	q_{t-1}	p_{t-1}	q_{t+1}	p_{t+1}
0	1	0	1	1	0	0	1	1	1	1	1	0	1	1	0	0	1	1	1
0	1	0	1	1	0	1	0	0	1	1	1	0	1	1	0	1	0	0	1
0	1	0	1	1	0	1	1	1	0	1	1	0	1	1	0	1	1	1	0
0	1	0	1	1	1	0	0	0	1	1	1	0	1	1	1	0	0	0	0
0	1	0	1	1	1	0	1	1	0	1	1	0	1	1	1	0	1	1	1
0	1	0	1	1	1	1	0	0	0	1	1	0	1	1	1	1	0	0	1
0	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0
0	1	1	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1
0	1	1	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	0
0	1	1	0	0	0	1	0	0	0	1	1	1	0	0	0	1	0	0	0
0	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	1	1	1	1
0	1	1	0	0	1	0	0	0	0	1	1	1	0	0	1	0	0	1	1
0	1	1	0	0	1	0	1	1	1	1	1	1	0	0	1	0	1	0	0
0	1	1	0	0	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0
0	1	1	0	0	1	1	1	1	0	1	1	1	0	0	1	1	1	0	1
0	1	1	0	1	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0
0	1	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	1	1
0	1	1	0	1	0	1	0	0	1	1	1	1	0	1	0	1	0	0	1
0	1	1	0	1	0	1	1	1	0	1	1	1	0	1	0	1	1	1	0
0	1	1	0	1	1	0	0	0	1	1	1	1	0	1	1	0	0	0	0
0	1	1	0	1	1	0	1	1	0	1	1	1	0	1	1	0	1	1	1
0	1	1	0	1	1	1	0	0	0	1	1	1	0	1	1	1	0	0	1
0	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0
0	1	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	1	0
0	1	1	1	0	0	0	1	1	0	1	1	1	1	0	0	0	1	0	1
0	1	1	1	0	0	1	0	0	0	1	1	1	1	0	0	1	0	1	1
0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	1	1	0	0
0	1	1	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	1
0	1	1	1	0	1	0	1	0	0	1	1	1	1	0	1	0	1	0	0
0	1	1	1	0	1	1	0	1	0	1	1	1	1	0	1	1	0	1	0
0	1	1	1	0	1	1	1	0	1	1	1	1	1	0	1	1	1	0	1
0	1	1	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	1	1	0
0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	0
0	1	1	1	1	0	1	1	1	0	1	1	1	1	1	0	1	1	1	1
0	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0
0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1
0	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	1

Continued on next page

Table C.2 – continued from previous page

$x_t^{(4)}$	$x_t^{(3)}$	$x_t^{(2)}$	$x_t^{(1)}$	q_t	p_t	q_{t-1}	p_{t-1}	q_{t+1}	p_{t+1}	$x_t^{(4)}$	$x_t^{(3)}$	$x_t^{(2)}$	$x_t^{(1)}$	q_t	p_t	q_{t-1}	p_{t-1}	q_{t+1}	p_{t+1}
0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0

C.3 Deriving Equations Without Memory Bits

Since the recursive equations may produce too complex, it would be appropriate to derive a different form of the equations. Following Armknecht's procedure [20], we can remove the memory bits since the output equation for keystream bits is linear. We include the steps to derive an equation without memory bits. We begin by defining two additional variables, a_t and b_t:

$$\begin{aligned} a_t &= \pi_t^4 \oplus \pi_t^3 p_t \oplus p_{t-1}, \\ b_t &= \pi_t^2 \oplus \pi_t^1 p_t \oplus 1. \end{aligned} \tag{C.3}$$

Using both variables a_t and b_t, we can now rewrite Equation C.1 and Equation C.2.

$$\begin{aligned} q_{t+1} &= a_t \oplus b_t q_t \\ p_{t+1} &= b_t \oplus 1 \oplus p_{t-1} \oplus p_t \oplus q_t \oplus q_{t-1} \end{aligned} \tag{C.4}$$

We can now multiply q_{t+1} by b_t and get a new equation:

$$0 = b_t(a_t \oplus q_t \oplus q_{t+1}). \tag{C.5}$$

Furthermore, we can rewrite p_{t+1} from Equation C.4:

$$q_t \oplus q_{t-1} = b_t \oplus 1 \oplus p_{t-1} \oplus p_t \oplus p_{t+1}. \tag{C.6}$$

We can now use the last Equation C.6 where we substitute t with $t+1$ and insert it into Equation C.5, thus, getting:

$$0 = b_t(a_t \oplus b_{t+1} \oplus 1 \oplus p_t \oplus p_{t+1} \oplus p_{t+2}). \tag{C.7}$$

Using the fact, that the memory bit p_t is combined linearly with π_t^1 ($z_t = \pi_t^1 + p_t$), we can substitute all memory bits in Equation C.7, deriving the

Table C.3: Using LSH with $t = 0.55$ on equations from Table 4.6

		F4		SAT
E0 type	Uc[a]	G. Basis Time (s)	Mem (GiB)	Time (s)
E0*(3, 6, c, 14)	14	2.3 ± 0.6	0.2 ± 0.0	1.1 ± 0.1
E0(3, 6, c, 30)	15	1.0 ± 0.2	0.1 ± 0.0	1.7 ± 0.4
E0(3, 6, c, 60)	16	7.5 ± 1.0	0.6 ± 0.1	7.1 ± 0.8
E0(3, 6, 12, 60)	17	26.3 ± 0.5	2.8 ± 0.1	25.5 ± 3.5
E0*(6, c, 14, 30)	18	152.9 ± 44.8	5.3 ± 0.3	41.4 ± 3.9
E0(6, c, 14, 60)	19	423.5 ± 57.0	14.8 ± 3.1	105.6 ± 7.2
E0(6, c, 30, 60)	20	1245.5 ± 68.9	35.6 ± 3.6	787.1 ± 60.0

[a] Keystream bits

following equation for four consecutive keystream bits:

$$
\begin{aligned}
0 = {} & z_{t+3}(z_{t+1}\pi^1_{t+1} \oplus \pi^2_{t+1} \oplus \pi^1_{t+1} \oplus 1) \oplus \\
& z_{t+2}(z_{t+1}\pi^1_{t+2}\pi^1_{t+1} \oplus z_{t+1}\pi^1_{t+1} \oplus \pi^2_{t+1}\pi^1_{t+2} \oplus \\
& \quad \pi^2_{t+1} \oplus \pi^1_{t+2}\pi^1_{t+1} \oplus \pi^1_{t+2} \oplus \pi^1_{t+1} \oplus 1) \oplus \\
& z_{t+1}(z_t\pi^1_{t+1} \oplus \pi^3_{t+1} \oplus \pi^2_{t+2}\pi^1_{t+1} \oplus \pi^2_{t+1} \oplus \\
& \quad \pi^1_{t+3}\pi^1_{t+1} \oplus \pi^1_{t+1}\pi^1_t \oplus \pi^1_{t+1} \oplus 1) \oplus \\
& z_t(\pi^2_{t+1} \oplus \pi^1_{t+1} \oplus 1) \oplus \\
& \pi^4_{t+1} \oplus \pi^2_{t+2}(\pi^2_{t+1} \oplus \pi^1_{t+1} \oplus 1) \oplus \\
& \pi^2_{t+1}(\pi^1_{t+3} \oplus \pi^1_t) \oplus \\
& \pi^1_{t+3}(\pi^1_{t+1} \oplus 1) \ \oplus \pi^1_{t+1}\pi^1_t \oplus \pi^1_t
\end{aligned}
\tag{C.8}
$$

Note that the final equation is incorrect in both Armknecht's papers [20, 39]. We verified the correctness with `sage` in the Jupyter Notebook `Armknecht's Formulation`. Armknecht [20] further analyzed Equation 2.27 and found out that the number of all possible monomials is approximately equal to $2^{23.07}$, which is the number of keystream bits needed in case of linearization. We refer the reader to [20, 39] for more details.

C.4 LSH - Additional Results

Table C.4: Polynomials and monomials after LSH with $t = 0.55$ on equations from Table 4.6

	⊘ Number of polynomials		⊘ #monomials in polynomials	
E0 type	Before LSH	After LSH	Before LSH	After LSH
E0*(3, 6, c, 14)	300	85.4 ± 36.1	375.7 ± 83.4	126.4 ± 77.9
E0(3, 6, c, 30)	321	297.9 ± 41.4	462.7 ± 94.7	175.7 ± 104.0
E0(3, 6, c, 60)	343	308.1 ± 15.4	559.9 ± 75.8	216.0 ± 120.1
E0(3, 6, 12, 60)	364	373.1 ± 19.0	683.7 ± 97.3	277.3 ± 148.0
E0*(6, c, 14, 30)	386	418.9 ± 9.0	828.0 ± 114.1	361.3 ± 168.1
E0(6, c, 14, 60)	407	387.7 ± 5.6	961.2 ± 103.1	412.5 ± 186.4
E0(6, c, 30, 60)	428	817.3 ± 29.6	1181.9 ± 325.9	555.9 ± 266.3

Table C.5: Using LSH with $t = 0.6$ on equations from Table 4.6

		F4		SAT
		G. Basis		
E0 type	Uc[a]	Time (s)	Mem (GiB)	Time (s)
E0*(3, 6, c, 14)	14	1.6 ± 0.3	0.1 ± 0.0	1.1 ± 0.1
E0(3, 6, c, 30)	15	1.0 ± 0.2	0.1 ± 0.0	2.4 ± 0.5
E0(3, 6, c, 60)	16	6.1 ± 0.7	0.4 ± 0.1	9.3 ± 0.8
E0(3, 6, 12, 60)	17	19.0 ± 4.9	1.0 ± 0.3	42.3 ± 5.3
E0*(6, c, 14, 30)	18	64.6 ± 1.1	6.9 ± 0.3	60.9 ± 6.1
E0(6, c, 14, 60)	19	444.1 ± 28.9	10.4 ± 4.3	172.0 ± 14.5
E0(6, c, 30, 60)	20	1165.1 ± 55.6	16.8 ± 0.5	1230.7 ± 82.4

[a] Unknowns count

Table C.6: Polynomials and monomials after LSH with $t = 0.60$ on equations from Table 4.6

	⊘ Number of polynomials		⊘ #monomials in polynomials	
E0 type	Before LSH	After LSH	Before LSH	After LSH
E0*(3, 6, c, 14)	300	94.2 ± 34.6	375.7 ± 118.5	126.4 ± 84.3
E0(3, 6, c, 30)	321	337.1 ± 48.1	462.7 ± 123.7	175.7 ± 113.4
E0(3, 6, c, 60)	343	357.1 ± 14.1	559.9 ± 113.2	216.0 ± 141.3
E0(3, 6, 12, 60)	364	463.1 ± 27.8	683.7 ± 149.0	277.3 ± 175.5
E0*(6, c, 14, 30)	386	503.1 ± 15.7	828.0 ± 161.6	361.3 ± 199.9
E0(6, c, 14, 60)	407	454.4 ± 8.1	961.2 ± 159.8	412.5 ± 228.7
E0(6, c, 30, 60)	428	999.8 ± 34.0	1181.9 ± 383.4	555.9 ± 279.8