ALGORITHMS FOR KLOOSTERMAN ZEROS

by

Yun-Jung Kim

B.Sc., University of Lethbridge, 2008

THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN THE

DEPARTMENT OF MATHEMATICS

FACULTY OF SCIENCE

SIMON FRASER UNIVERSITY

Summer, 2011

APPROVAL

Name: Yun-Jung Kim

Degree: Master of Science

Title of thesis: Algorithms for Kloosterman Zeros

Examining Committee: Dr. Imin Chen (Chair)

Dr. Petr Lisoněk
Senior Supervisor

Dr. Michael Monagan
Supervisor

Dr. Jonathan Jedwab
Examiner

Date Approved: June 30, 2011

Declaration of
Partial Copyright Licence

Abstract

Kloosterman sums are exponential sums on finite fields with important applications in Cryptography and Coding Theory. Of particular importance are those field elements at which the Kloosterman sum attains the value 0, which are called Kloosterman zeros. They exist only in fields of characteristic 2 and 3.

We prove an upper bound on the density of the classical modular polynomial when it is considered as a polynomial over GF(2). We develop an algorithm to list exhaustively Kloosterman zeros in a given field of characteristic 2. Using this algorithm we list all Kloosterman zeros in fields of order 2^m for $m \leq 63$, whereas this has been done only for $m \leq 14$ in the literature.

We develop an algorithm to discover relations satisfied by coefficients of minimal polynomials of Kloosterman zeros in characteristic 2. We rediscover five such relations that have been proved in the literature and we conjecture two new relations.

I dedicate my thesis to my family: Yoong-Yoon Kim (Father), Hae-Sook Han (Mother) and Hyung-Jung Gabriel Kim (Brother), whom I always respect.

Acknowledgments

This thesis would not have been possible without Research Assistantships from NSERC Discovery Grant from Dr. Petr Lisoněk, Research Assistantship from MITACS NCE of Canada, and Teaching Assistantship from Department of Mathematics at SFU.

I would like to express my sincere gratitude to my supervisor Dr. Petr Lisoněk for the continuous support of my M.Sc. study and research and for his patience and passion. His guidance helped me at all times of research and writing of this thesis.

Besides my supervisor, I would like to thank all committee members and chair of my defence, Dr. Michael Monagan, Dr. Jonathan Jedwab and Dr. Imin Chen.

My sincere thanks go to Fr. Fernando Mignone, Catholic Christian Outreach (CCO), particularly Scott Roy and Tim Killoran for encouraging me.

Special thanks are also addressed to all my friends and labmates for their support. In particular, I would like to thank Anthony Shoemaker, Jun-Kyun Oh, Jeff Kwon, Young-Chul Cho, Ted Oh, Joon Park, Duk-Hwan Jang, Jin Lim, Dong-Eun Lee, Josh Tso and Bonn Coyuco for their great friendship.

Lastly, I thank for my parents, Yoong-Yoon Kim and Hae-Sook Han, and my brother, Hyung-Jung Gabriel Kim who always support me being here and encourage me and mostly importantly love me.

Table of contents

List of tables

List of Algorithms

List of figures

Chapter 1

Classical Modular Polynomials

1.1 Introduction

In this thesis, we show mathematical concepts and computational methods for classical modular polynomials, Kloosterman zeros, and coefficients of the characteristic polynomials of Kloosterman zeros in characteristic 2. Since our goals are computing Kloosterman zeros for large extension fields of $GF(2)$ and discovering their properties through computation, we heavily rely on the computational perspective. We have successfully improved computational results through different approaches. We prove new theorems about the density of classical modular polynomials over $GF(2)$. For Kloosterman zeros, we are only concerned with that part of the theory that is needed for our algorithms.

Our approach to computing Kloosterman zeros over extension fields of $GF(2)$ in Chapter 2 makes a novel use of classical modular polynomials in characteristic 2, which are presented in this chapter. These polynomials have been used recently in elliptic curve cryptography [32] but they have a rich history in many research areas originating in the middle of the 19th century.

Throughout this thesis, we only focus on characteristic 2; however, we realize that Kloosterman sums over extension fields of $GF(3)$ have similar properties. For example, Kloosterman sums over $GF(p^m)$ where $p \in \{2, 3\}$ are integers whereas others may not be. Moreover, no Kloosterman zero belongs to a subfield of $\mathbb{F}_{p^m}$ where $p \in \{2, 3\}$ except $p^m = 16$ as proved by Lisoněk and Moisio [25]. We strongly believe that methods that we will introduce throughout this thesis for characteristic 2 will apply for characteristic 3 as

well (due to restriction of time, we do not pursue for characteristic 3).

In this chapter, we explain what the j-invariant and the l-th classical modular polynomial are and how they are computed over $\mathbb{Z}_2$. Then, we analyze properties of modular polynomials in characteristic 2 and plot graphs of their exponents to make conjectures about their density. We start by presenting classical modular polynomials over integers in order to allow for a reduction modulo 3 as well, which is a possible research topic as explained in previous paragraphs; however we do not pursue this topic in our thesis.

1.2 j-invariant and Modular Polynomials

In this section, we describe the j-invariant and the l-th classical modular polynomial (for prime l only). We recommend [1] as an accessible reference for these topics.

We rephrase the concepts from a preprint by Vercauteren [32]. This paper has never been published and we attach missing proofs along with correcting some errors. Furthermore, the use of modular polynomials in [32] is for point counting on elliptic curves whereas we use them for computing Kloosterman zeros.

We say S is a Laurent series if we can write

$$S = \sum_{n \in \mathbb{Z}} s_n x^n. \tag{1.1}$$

For this thesis, we limit ourselves to Laurent series with finitely many terms with negative exponents.

Let $H = \{z \in \mathbb{C} | \Im(z) > 0\}$ denote the upper half of the complex plane. For $\tau \in H$, the j-invariant $j(\tau)$ and the discriminant $\Delta(\tau)$ are defined in all books on elliptic curves and elliptic functions, for example in [1]. The function $j(\tau)$ is a holomorphic function on H, hence for $\tau \in H$ the value of $j(\tau)$ can be defined using its Laurent series. Denote

$$q = e^{2\pi i \tau}. \tag{1.2}$$

If $\tau \in H$, then $0 < |q| < 1$. Thus, let us emphasize that from now on, we can view j as a function of τ or as a function of q. The q-expansion for j then is given as follows.

We describe the discriminant and the j-invariant as q-series. We recommend [1] as an

accessible reference for these topics. The discriminant $\Delta(\tau)$ is given by

$$\Delta(\tau) = q\prod_{n\geq 1}(1-q^n)^{24}. \tag{1.3}$$

Applying Euler's identity for partitions, we get

$$\Delta(\tau) = q(1+\sum_{n\geq 1}(-1)^n(q^{n(3n-1)/2}+q^{n(3n+1)/2}))^{24}. \tag{1.4}$$

Furthermore, the relation between $j(\tau)$ and $\Delta(\tau)$ is

$$j(\tau) = \frac{(256f(\tau)+1)^3}{f(\tau)} \text{ where } f(\tau) = \frac{\Delta(2\tau)}{\Delta(\tau)}. \tag{1.5}$$

From equation (1.5), we get $j(\tau)$ as the q-expansion Laurent series

$$j(\tau) = \frac{1}{q} + 744 + \sum_{n\geq 1}^{\infty} c_n q^n \text{ where } c_n \in \mathbb{Z}. \tag{1.6}$$

For prime number l we now define the l-th classical modular polynomial using the j-invariant introduced above. The definition for nonprime l also exists, but it is more complicated.

Definition 1.2.1. (Section III.8 of [1]) *Let l be a prime. We define the l-th classical modular polynomial to be*

$$\Phi_l(x, j(\tau)) = (x - j(l\tau))\prod_{i=0}^{l-1}(x - j(\frac{\tau+i}{l})). \tag{1.7}$$

We say that l is the *level* of Φ_l.

It is not quite obvious from this definition that Φ_l is a polynomial in x and j; this will be asserted in Proposition 1.2.4.

Example 1.2.2. *There are some examples of Φ_l for $l = 2, 3$:*
$\Phi_2(x,y) = x^3 + y^3 - x^2y^2 + 1488(x^2y + xy^2) - 162000(x^2 + y^2) + 40773375xy + 8748000000(x + y) - 157464000000000$
$\Phi_3(x,y) = x^4 + y^4 - x^3y^3 + 2232(x^3y^2 + x^2y^3) - 1069956(x^3y + xy^3) + 36864000(x^3 + y^3) + 2587918086x^2y^2 + 8900222976000(x^2y + xy^2) + 452984832000000(x^2 + y^2) - 770845966336000000xy + 1855425871872000000000(x + y)$

First, from Definition 1.2.1, we can derive the following proposition.

Proposition 1.2.3. *We have*

$$\Phi_l(j(l\tau), j(\tau)) = 0. \tag{1.8}$$

From equation (1.2) and (1.6), we can obtain $j(l\tau)$ by substituting q^l for q in $j(\tau)$.

Proposition 1.2.4. (Section III.8 of [1]) *The l-th modular polynomial $\Phi_l(x, y)$ for prime l satisfies*

$$\Phi_l(x, y) = x^{l+1} - x^l y^l + y^{l+1} + \sum_{i,k \le l,\ \ i+k<2l} a_{ik} x^i y^k, \quad a_{ik} \in \mathbb{Z} \tag{1.9}$$

Moreover, $\Phi_l(x, y)$ is symmetric so that $a_{ik} = a_{ki}$ and $\Phi_l(x, y) = \Phi_l(y, x)$.

Furthermore, for later use let us also define $a_{(l+1)0} = a_{0(l+1)} = 1$ and $a_{ll} = -1$. Now we present an algorithm that computes all coefficients a_{ik}. This algorithm has been considered by many authors; our presentation follows [32].

Using Propositions 1.2.3 and 1.2.4, we have the equality of Laurent series in q

$$j^{l+1}(l\tau) + j^{l+1}(\tau) = -\sum_{i=0}^{l} \sum_{k=0}^{l} a_{ik} j^i(l\tau) j^k(\tau). \tag{1.10}$$

by defining $a_{ll} = -1$ (to remove term $-j^l(l\tau)j^l(\tau)$ from the left hand side). By equation (1.6), the left side of equation (1.10) is a Laurent series in q of the form

$$j^{l+1}(l\tau) + j^{l+1}(\tau) = \frac{1}{q^{l(l+1)}} + \sum_{n\ >\ -l(l+1)} c_n q^n \text{ where } c_n \in \mathbb{Z}. \tag{1.11}$$

Due to symmetry of $\Phi_l(x, y)$, we only need to consider the pairs (i, k) where $0 \le k \le i \le l$. By equation (1.6), we have

$$j^i(l\tau) j^k(\tau) = \frac{1}{q^{il+k}} + \sum_{n\ >\ -(il+k)} t_n q^n \text{ where } t_n \in \mathbb{Z}. \tag{1.12}$$

Also we have

$$-l(l+1) \le -(il+k) \le 0 \quad \text{for} \quad 0 \le k \le i \le l.$$

Starting from equation (1.10), in each iteration of Algorithm 1.1 the variable L contains the left side of the modified equation (1.10) as we move terms from the right side to the left

side. Let $p(L)$ denote the least exponent of q in L. We compute i and k such that $j^i(l\tau)j^k(\tau)$ contains a term $cq^{p(L)}$ such that $c \in \mathbb{Z}$ and $0 \leq k \leq i \leq l$. Note that these integers i and k are defined *uniquely* by the division algorithm when we divide $-p(L)$ by l. Note that we do not have exactly the conditions of the Euclidean division, since we may allow $k = l$. Now $k = l$ forces $i = l$ by the previous inequalities, then $-p(L) = l(l+1)$ and there is no other pair (i, k) satisfying all previous conditions. This shows the uniqueness of (i, k) in all cases.

By the uniqueness of (i, k), we obtain $a_{ik} = -c$ where c is as above. By the symmetry of Φ_l we also obtain $a_{ki} = -c$ if $i \neq k$.

Thus, in each iteration we determine one coefficient of Φ_l (up to symmetry, possibly). Since each series $j^i(l\tau)j^k(\tau)$ contains a term with a non-positive exponent of q, it is enough to perform the iterations while $p(L) \leq 0$.

As before we denote by c the coefficient at $q^{p(L)}$ at the left side of the modified equation (1.10). After computing a_{ik} and a_{ki} as above, we add

$$-c(j^i(l\tau)j^k(\tau) + j^k(l\tau)j^i(\tau))$$

to both sides of (1.10). If $i = k$, then we only add $-cj^i(l\tau)j^i(\tau)$ to both sides of (1.10). In any case, the least exponent of q in the modified equation (1.10) increases. This together with the condition $p(L) \leq 0$ ensures termination of the algorithm.

For example, the monomial with the least exponent of q at the left side of equation (1.10) is $q^{-l(l+1)}$. This implies that the right side of the equation must contain the monomial $q^{-l(l+1)}$. This monomial is only obtained when $i = l$ and $k = l$ so that we have $a_{ll} = -1$. By adding $-j^l(l\tau)j^l(\tau)$ to both sides, we have

$$j^{l+1}(l\tau) + j^{l+1}(\tau) - j^l(l\tau)j^l(\tau) = -\sum_{i=0}^{l}\sum_{k=0}^{l} a_{ik}j^i(l\tau)j^k(\tau) \text{ where } i + k < 2l. \quad (1.13)$$

In addition, the least exponent of q of the left side is now greater than $-l(l+1)$.

If we start the algorithm from equation (1.13) as opposed to starting from (1.10), then in each iteration we can compute the unique pair (i, k) using formulas $i = \lfloor \frac{-p(L)}{l} \rfloor$ and $k = -p(L) - il$. This is the version that is shown in the pseudocode in Algorithm 1.1.

Example 1.2.5. *For $l = 3$ we set $a_{40} = a_{04} = 1$ and $a_{33} = -1$.*

$$j^4(3\tau) + j^4(\tau) - j^3(3\tau)j^3(\tau) = -2232q^{-11} - 2251260q^{-10} - 1355201496q^{-9} + \ldots$$

We let L be the right side. We set $p(L) = -11$ and compute $(i, k) = (3, 2)$ so that $-(il + k) = -11$. Then we set $a_{32} = a_{23} = 2232$ and add $2232(j^3(3\tau)j^2(\tau) + j^2(3\tau)j^3(\tau))$ to both sides. We have

$$\begin{aligned} & j^4(3\tau)j^4(\tau) - j^3(3\tau)j^3(\tau) + 2232(j^3(3\tau)j^2(\tau) + j^2(3\tau)j^3(\tau)) \\ = \;& 1069956q^{-10} + 759183264q^{-9} + 208069299018q^{-8} + 10248720528384q^{-7} + \ldots \end{aligned}$$

We update L and set $p(L) = -10$. We get $(i, k) = (3, 1)$ and $a_{31} = a_{13} = -1069956$. Then add $-1069956(j^3(3\tau)j(\tau) + j(3\tau)j^3(\tau))$ to both sides. We iterate until the right side has positive exponent terms only. Then we obtain the coefficients of Φ_3. As a result, we have

$$\begin{aligned} \Phi_3(x, y) &= x^4 + y^4 - x^3y^3 + 2232(x^3y^2 + x^2y^3) - 1069956(x^3y + xy^3) \\ &+ 36864000(x^3 + y^3) + 2587918086x^2y^2 + 8900222976000(x^2y + xy^2) \\ &+ 452984832000000(x^2 + y^2) - 770845966336000000xy \\ &+ 1855425871872000000000(x + y). \end{aligned}$$

We can not compute the infinite q-series $j^i(l\tau)j^k(\tau)$ exactly. However, since the while loop of Algorithm 1.1 is driven only by the non-positive exponents of q, it is enough to ensure that $j^i(l\tau)j^k(\tau)$ have correct terms with non-positive exponents of q. As an easy consideration shows, to achieve this goal it is possible to truncate $j^i(l\tau)$, $j^k(\tau)$ and $j^i(l\tau)j^k(\tau)$ (as q-series) after the power $q^{l(l+1)}$, which gives a finite representation that we can compute with. Moreover, in the while loop it is enough to compute with the terms having non-positive exponents for all the series involved.

As mentioned earlier, in later chapters we do not use Algorithm 1.1 directly; rather we use its reduction modulo 2. Nonetheless, we include this algorithm so that it can be a guideline to compute modular polynomials over $\mathbb{Z}_3$ if desired. Notice that the algorithm for modular polynomials over $\mathbb{Z}_2$ and $\mathbb{Z}_3$ is just the same algorithm (with all computations performed modulo 2 or modulo 3, respectively), because all power series computed by the algorithm have integer coefficients and the modulo operation is well-defined over integers.

Note that Algorithm 1.1 is not useful for computing modular polynomials in characteristic 0 or in large prime characteristic. There are other methods to compute modular polynomials in those cases. However, we believe that it is useful in characteristic 2 or 3.

Algorithm 1.1 Classical Modular Polynomial over $\mathbb{Z}$

Input: Prime l

Output: l^{th} classical modular polynomial $\Phi_l(x, y)$ over $\mathbb{Z}$.

All power series computations are truncated after the exponent $l(l+1)$.

1: Determine $j(\tau)$ using equation (1.5) as q-expansion Laurent series
2: Compute $j^i(\tau)$ for $2 \leq i \leq l+1$ with setting $j^0(\tau) \leftarrow 1$
3: Determine $j^i(l\tau)$ for $0 \leq i \leq l+1$ by substituting q^l for q in $j^i(\tau)$ for $1 \leq i \leq l+1$
4: $A \leftarrow 0$, where $A[i,k] = a_{ik}$ as introduced in Proposition 1.2.4
5: $L \leftarrow j^{l+1}(l\tau) + j^{l+1}(\tau) - j^l(l\tau)j^l(\tau)$
6: $A[l+1, 0] \leftarrow 1$, $A[0, l+1] \leftarrow 1$, $A[l, l] \leftarrow -1$
7: **while** $p(L) \leq 0$, where $p(L)$ is the least exponent of q with nonzero coefficient in L **do**
8: $\quad i \leftarrow \lfloor \frac{-p(L)}{l} \rfloor$, $k \leftarrow -p(L) - il$; thus $p(L) = -(il + k)$
9: $\quad c \leftarrow$ the coefficient of $q^{p(L)}$ in L
10: $\quad A[i,k] \leftarrow -c$, $A[k,i] \leftarrow -c$
11: $\quad$ **if** $(i = k)$ **then**
12: $\quad\quad L \leftarrow L - cj^i(l\tau)j^i(\tau)$
13: $\quad$ **else**
14: $\quad\quad L \leftarrow L - c(j^i(l\tau)j^k(\tau) + j^k(l\tau)j^i(\tau))$
15: $\quad$ **end if**
16: **end while**
17: **return** $\Phi_l(x, y) = \sum_{i=0}^{l+1} \sum_{k=0}^{l+1} A[i,k]x^i y^k$

1.3 Arithmetics of Reduced Power Series over $\mathbb{Z}_2$

In this section, we introduce a reduced power series and its arithmetics over $\mathbb{Z}_2$. This representation and arithmetic operations are used in our C++ implementation of Algorithm 1.2 in Appendix A.1.

As mentioned earlier, we work with Laurent series only of the form

$$\sum_{n=s}^{\infty} a_n x^n \text{ where } s \leq 0.$$

For any series S over $\mathbb{Z}_2$, consider its representations as

$$S = q^{-v_s} \sum_{n \geq 0} s_n q^{m_s n} \text{ where } v_s \in \mathbb{Z}, m_s \in \mathbb{N}, s_n \in \mathbb{Z}_2 \text{ and } s_0 = 1. \tag{1.14}$$

In general, there is more than one such representation. The representation (1.14) can be made unique by requiring that m_s has to be the *largest possible* integer. If this condition is satisfied, then we call (1.14) a *power series in the reduced representation.* This representation utilizes memory space efficiently. Note that m_s is the greatest common divisor of all differences of exponents in S. Note that each power series in the reduced representation contains at least two nonzero coefficient terms, otherwise m_s is not well defined.

Throughout this section, we let

$$A = q^{-v_a} \sum_{n \geq 0} a_n q^{m_a n}, \quad B = q^{-v_b} \sum_{n \geq 0} b_n q^{m_b n} \quad \text{and} \quad C = q^{-v_c} \sum_{n \geq 0} c_n q^{m_c n}$$

be three power series in the reduced representation over $\mathbb{Z}_2$.

The following propositions are taken from a paper by Vercauteren [32], where they appear without proofs. We provide proofs in order to ensure the arithmetic over $\mathbb{Z}_2$ is correct. In fact, we correct the errors of properties in [32].

Proposition 1.3.1. *If $C = A + B$, then $v_c \leq \max(v_a, v_b)$ and $\gcd(v_a - v_b, m_a, m_b) \mid m_c$.*

Proof. We may assume $A \neq B$; otherwise $C = 0$. Suppose $C = A + B$. Rewrite

$$A = \sum_{i=\min(-v_a,-v_b)}^{\infty} a'_i q^i \quad \text{and} \quad B = \sum_{i=\min(-v_a,-v_b)}^{\infty} b'_i q^i$$

where $a'_i = 0$ if $i \not\equiv -v_a \pmod{m_a}$ and $b'_i = 0$ if $i \not\equiv -v_b \pmod{m_b}$. Thus, we have

$$C = \sum_{i=\min(-v_a,-v_b)}^{\infty} c'_i q^i.$$

This implies $v_c \leq \max(v_a, v_b)$. Moreover, $c'_k = 1$ if either $a'_k = 1$ or $b'_k = 1$ but not both. Assume that for some $m, m' \in \mathbb{Z}$, we have $c'_m = 1$ and $a'_m = 1$ and $b'_m = 0$, and $c'_{m'} = 1$ and $a'_{m'} = 0$ and $b'_{m'} = 1$. Then, we have

$$\begin{aligned} m &= -v_a + m_a z \quad \text{for some } z \in \mathbb{Z} \\ m' &= -v_b + m_b z' \quad \text{for some } z' \in \mathbb{Z}. \end{aligned}$$

It follows that the difference of any two exponents in C is either $-(v_a - v_b) + m_a z - m_b z'$ for some integers z, z' or $m_a z''$ or $m_b z'''$ for some integers z'', z'''. We denote $m_g = \gcd(v_a - v_b, m_a, m_b)$. Then, m_g divides the difference of any two exponents in C. We have

$$C = q^{-v_c} \sum_{n=0}^{\infty} c''_n q^{m_g n}.$$

Therefore, $\gcd(v_a - v_b, m_a, m_b)$ divides m_c. □

Proposition 1.3.2. *If $C = AB$, then $v_c = v_a + v_b$ and $\gcd(m_a, m_b) \mid m_c$.*

Proof. Suppose $C = AB$.

$$AB = q^{-(v_a+v_b)} \sum_{i=0}^{\infty} a_i q^{m_a i} \sum_{i=0}^{\infty} b_i q^{m_b i}.$$

Thus, $v_c = v_a + v_b$.

Let $A' = q^{v_a} A$, $B' = q^{v_b} B$ and $C' = A'B'$. It is clear that $C' = q^{v_a+v_b} C = q^{v_c} C$. We denote $m_g = \gcd(m_a, m_b)$ for simplicity. We can rewrite

$$A' = \sum_{i=0}^{\infty} a'_i q^{m_g i} \quad \text{and} \quad B' = \sum_{i=0}^{\infty} b'_i q^{m_g i}$$

where $a'_k = 0$ if $m_g k \not\equiv 0 \pmod{m_a}$ and $b'_k = 0$ if $m_g k \not\equiv 0 \pmod{m_b}$. Then, we have

$$\begin{aligned} A'B' &= \sum_{i=0}^{\infty} a'_i q^{m_g i} \sum_{k=0}^{\infty} b'_k q^{m_g k} \\ &= \sum_{k=0}^{\infty} a'_0 b'_k q^{m_g k} + \sum_{k=0}^{\infty} a'_1 b'_k q^{m_g (k+1)} + \sum_{k=0}^{\infty} a'_2 b'_k q^{m_g (k+2)} + \cdots \\ &= \sum_{k=0}^{\infty} c'_k q^{m_g k}. \end{aligned}$$

Therefore, $\gcd(m_a, m_b)$ divides m_c. □

Proposition 1.3.3. *If $C = A^2$, then $v_c = 2v_a$ and $m_c = 2m_a$.*

Proof. We know $(X+Y)^2 = X^2 + Y^2$ in characteristic 2.

$$A^2 = (q^{-v_a} \sum_{n \geq 0} a_n q^{m_a n})^2 = q^{-2v_a} \sum_{n \geq 0} a_n q^{2m_a n}$$

Thus, $v_c = 2v_a$ and $m_c = 2m_a$. □

Proposition 1.3.4. *If $C = A^{-1}$, then $v_c = -v_a$ and $m_c = m_a$.*

Proof. Suppose $AC = 1$. We have

$$AC = q^{-(v_a + v_c)} (\sum_{n \geq 0} a_n q^{m_a n})(\sum_{n \geq 0} c_n q^{m_c n}).$$

We have $q^{-(v_a+v_c)}$ is the least power of q in AC with nonzero coefficient, which is $a_0 c_0 = 1$. Then, we must have $v_c = -v_a$.

For $k \geq 0$ let $c'_k \in \mathbb{Z}_2$ be defined as follows: $c'_0 = 1$ and for $k > 0$, set c'_k such that $\sum_{i=1}^{k} a_i c'_{k-i} = 0$. Then we have

$$\begin{aligned} 1 &= a_0 c'_0 \\ 0 &= a_0 c'_1 + a_1 c'_0 \\ &\vdots \\ 0 &= a_0 c'_l + a_1 c'_{l-1} + \cdots + a_l c'_0 \\ &\vdots \end{aligned}$$

Let $C' = q^{-v_c} \sum_{k \geq 0} c'_k q^{m_a k}$. Then $AC' = 1$. It follows that $m_a | m_c$. Using a symmetric argument we show $m_c | m_a$. Hence $m_c = m_a$. □

We use these propositions to write the q-expansion Laurent series for the j-invariant as a power series in the reduced representation over $\mathbb{Z}_2$. This series occurs in the proof of Lemma 1.4.2. Moreover, the propositions that we just proved are utilized in Algorithm 1.2 which computes with power series in the reduced representation over $\mathbb{Z}_2$. Our C++ implementation of this algorithm using *NTL* library [31] can be found in Appendix A.1.

1.4 Algorithm for Modular Polynomials over $\mathbb{Z}_2$

Now we provide an algorithm for computing modular polynomials over $\mathbb{Z}_2$. This is just a slight modification of Algorithm 1.1. As was mentioned earlier already, our presentation follows [32].

We have introduced the j-invariant $j(\tau)$ and discriminant $\Delta(\tau)$ in Section 1.2. We define

$$\bar{j}(\tau) = j(\tau) \bmod 2 \text{ and } \bar{\Delta}(\tau) = \Delta(\tau) \bmod 2.$$

Note that $j(\tau)$ and $\Delta(\tau)$ have Laurent series with integer coefficients as given above, hence the reduction modulo 2 is well defined and it should be understood at the power series level. The following proposition and lemma appear in a paper by Vercauteren [32] without proofs. We include the proofs to check the correctness of them.

Proposition 1.4.1. *In characteristic 2, we compute the j-invariant $\bar{j}(\tau)$ as follows:*

$$\bar{j}(\tau) = q^{-1}\frac{1+\sum_{n\geq 1} q^{4n(3n-1)} + q^{4n(3n+1)}}{1+\sum_{n\geq 1} q^{16n(3n-1)} + q^{16n(3n+1)}}. \tag{1.15}$$

Proof. Let f be as defined in equation (1.5). Note that f has a Laurent series with integer coefficients, since the leading coefficient in the series for $\Delta(\tau)$ is 1. Define $\bar{f} = f \bmod 2$. Rewriting equation (1.5) over $\mathbb{Z}_2$, we have

$$\bar{j}(\tau) = \frac{(256\bar{f}(\tau)+1)^3}{\bar{f}(\tau)} = \frac{1}{\bar{f}(\tau)} = \frac{\bar{\Delta}(\tau)}{\bar{\Delta}(2\tau)}. \tag{1.16}$$

By applying reduction modulo two on equation (1.4), we get

$$\begin{aligned}\bar{\Delta}(\tau) &= q(1+\sum_{n\geq 1}(q^{4n(3n-1)}+q^{4n(3n+1)}))^3 \\ &= q(1+\sum_{n\geq 1}(q^{4n(3n-1)}+q^{4n(3n+1)}))(1+\sum_{n\geq 1}(q^{8n(3n-1)}+q^{8n(3n+1)}))\end{aligned}$$

and using equation (1.2)

$$\bar{\Delta}(2\tau) = q^2(1+\sum_{n\geq 1}(q^{8n(3n-1)}+q^{8n(3n+1)}))(1+\sum_{n\geq 1}(q^{16n(3n-1)}+q^{16n(3n+1)})).$$

We have $\bar{j}(\tau) = \frac{\bar{\Delta}(\tau)}{\Delta(2\tau)}$. Thus we have

$$\begin{aligned}\bar{j}(\tau) &= \frac{q(1+\sum_{n\geq 1}(q^{4n(3n-1)}+q^{4n(3n+1)}))(1+\sum_{n\geq 1}(q^{8n(3n-1)}+q^{8n(3n+1)}))}{q^2(1+\sum_{n\geq 1}(q^{8n(3n-1)}+q^{8n(3n+1)}))(1+\sum_{n\geq 1}(q^{16n(3n-1)}+q^{16n(3n+1)}))} \\ &= q^{-1}\frac{(1+\sum_{n\geq 1}(q^{4n(3n-1)}+q^{4n(3n+1)}))}{(1+\sum_{n\geq 1}(q^{16n(3n-1)}+q^{16n(3n+1)}))}.\end{aligned}$$

□

Using propositions from Section 1.3, we show $\bar{j}(\tau)$ as a reduced power series.

Lemma 1.4.2. *We have*

$$\bar{j}(\tau) = q^{-1}\sum_{n\geq 0} s_n q^{8n},\ s_i \in \mathbb{Z}_2. \tag{1.17}$$

Proof. Let us consider the numerator of equation (1.15). We claim that

$$1+\sum_{n\geq 1}(q^{4n(3n-1)}+q^{4n(3n+1)}) = \sum_{n\geq 0} a_n q^{8n}.$$

Since $4n(3n-1)+8n = 4n(3n+1)$, it suffices to show $12n^2-4n = 8n^2+4n(n-1) \equiv 0 \pmod 8$ (since $n(n-1) \equiv 0 \pmod 2$). Thus, $1+\sum_{n\geq 1}(q^{4n(3n-1)}+q^{4n(3n+1)}) = \sum_{n\geq 0} a_n q^{8n}$. By arithmetic operations on power series over $\mathbb{Z}_2$ from Section 1.3, we conclude

$$\begin{aligned}\bar{j}(\tau) &= q^{-1}\frac{1+\sum_{n\geq 1}(q^{4n(3n-1)}+q^{4n(3n+1)})}{1+\sum_{n\geq 1}(q^{16n(3n-1)}+q^{16n(3n+1)})} \\ &= q^{-1}\frac{\sum_{n\geq 0} a_n q^{8n}}{1+(\sum_{n\geq 1} q^{4n(3n-1)}+q^{4n(3n+1)})^4} \\ &= q^{-1}\frac{\sum_{n\geq 0} a_n q^{8n}}{(\sum_{n\geq 0} a_n q^{8n})^4} = q^{-1}\frac{\sum_{n\geq 0} a_n q^{8n}}{\sum_{n\geq 0} a_n q^{32n}} \text{ (by Proposition 1.3.3)} \\ &= q^{-1}\sum_{n\geq 0} a_n q^{8n}\sum_{n\geq 0} c_n q^{32n} \text{ (by Proposition 1.3.4)} \\ &= q^{-1}\sum_{n\geq 0} s_n q^{8n} \text{ (by Proposition 1.3.2).}\end{aligned}$$

□

Notice that $\bar{j}(\tau)$ has only one term with a negative exponent. We can get the first few terms using the `jInvariant` function in Magma [2]. We get:

$$\bar{j}(\tau) = q^{-1} + q^7 + q^{15} + q^{31} + q^{47} + \cdots$$

It follows that equation (1.17) is a power series for $\bar{j}(\tau)$ in the reduced representation, since the integer 8 in the exponent in (1.17) is the largest possible.

Recall that for prime number l, the l-th classical modular polynomial $\Phi_l(x, y)$ was introduced in Definition 1.2.1.

Definition 1.4.3. *Let l be prime. We define $\bar{\Phi}_l(x, y) = \Phi_l(x, y) \bmod 2$, the l-th modular polynomial over $\mathbb{Z}_2$.*

Example 1.4.4. *There are examples of $\bar{\Phi}_l$ for $l = 2, 3$:*
$\bar{\Phi}_2(x, y) = x^3 + x^2y^2 + xy + y^3$
$\bar{\Phi}_3(x, y) = x^4 + x^3y^3 + y^4$

Proposition 1.4.5. *Let l be prime. We have*

$$\bar{\Phi}_l(x, y) = x^{l+1} + x^l y^l + y^{l+1} + \sum_{i,k \le l,\quad i+k<2l} a_{ik} x^i y^k, a_{ik} \in \mathbb{Z}_2 \tag{1.18}$$

where $a_{ik} = a_{ki}$ for all i and k. The polynomial $\bar{\Phi}_l(x, y)$ is symmetric.

Proof. These properties immediately follow from the properties of Φ_l over $\mathbb{Z}$ given in Proposition 1.2.4. □

For prime l, Algorithm 1.2 computes the l-th modular polynomial over $\mathbb{Z}_2$. As in Algorithm 1.1, in the while loop it is enough to consider terms with non-positive exponents of q in all power series.

We have used Algorithm 1.2 to compute all classical modular polynomials $\bar{\Phi}_l$ for prime levels $l < 2000$ on a Pentium(R) D 3.00 GHz with 2 GB memory in C++ with *NTL* library [31]. In preprint [32] modular polynomials are computed also up to $l < 2000$. Our main goal was storing modular polynomials so that we can develop our algorithms for Chapter 2. We found that we do not require higher levels of modular polynomials. For this reason, we did not pursue computing modular polynomials of higher levels.

Algorithm 1.2 Classical Modular Polynomial over $\mathbb{Z}_2$

Input: Prime l

Output: l^{th} classical modular polynomial $\bar{\Phi}_l(x, y)$ over $\mathbb{Z}_2$

All power series computations are truncated after the exponent $l(l+1)$.

1: Determine $\bar{j}(\tau)$ using equation (1.15) as q-expansion Laurent series.
2: Compute $\bar{j}^i(\tau)$ for $2 \leq i \leq l+1$ with setting $\bar{j}^0(\tau) \leftarrow 1$
3: Determine $\bar{j}^i(l\tau)$ for $0 \leq i \leq l+1$ by substituting q^l for q in $\bar{j}^i(\tau)$ for $1 \leq i \leq l+1$
4: $A \leftarrow 0$, where $A[i,k] = a_{ik}$ as introduced in Proposition 1.4.5
5: $L \leftarrow \bar{j}^{l+1}(l\tau) + \bar{j}^{l+1}(\tau) + \bar{j}^l(l\tau)\bar{j}^l(\tau)$
6: $A[l+1, 0] \leftarrow 1$, $A[0, l+1] \leftarrow 1$, $A[l, l] \leftarrow 1$
7: **while** $p(L) \leq 0$, where $p(L)$ is the least exponent of q with nonzero coefficient in L **do**
8: $\quad i \leftarrow \lfloor \frac{-p(L)}{l} \rfloor$, $k \leftarrow -p(L) - il$; thus $p(L) = -(il + k)$
9: $\quad A[i,k] \leftarrow 1$, $A[k,i] \leftarrow 1$
10: $\quad$ **if** $(i = k)$ **then**
11: $\quad\quad L \leftarrow L + \bar{j}^i(l\tau)\bar{j}^i(\tau)$
12: $\quad$ **else**
13: $\quad\quad L \leftarrow L + \bar{j}^i(l\tau)\bar{j}^k(\tau) + \bar{j}^k(l\tau)\bar{j}^i(\tau)$
14: $\quad$ **end if**
15: **end while**
16: **return** $\bar{\Phi}_l(x, y) = \sum_{i=0}^{l+1} \sum_{k=0}^{l+1} A[i,k] x^i y^k$

B	250	500	750	1000	1500	2000
Precompute $\bar{j}(\tau)$	0	0	1	2	5	11
Precompute $\bar{j}^i(\tau)$ for $i = 2, \cdots, B$	0	1	3	9	58	161
Compute $\bar{\Phi}_l(x, y)$ for all prime $l < B$	11	417	6150	29064	374835	1853030
Total Time	11	418	6154	29075	374898	1853202

Table 1.1: Running time (in seconds) on a Pentium(R) D 3.00GHz for Algorithm 1.2

Table 1.1 presents the running times (in seconds) for computing the l-th classical modular polynomial over $\mathbb{Z}_2$ using Algorithm 1.2, for *all* $l < B$, with reduced power series representation from Section 1.3. Our C++ code can be found in Appendix A.1.

Notice B represents the upper bound of the prime level l as the input of the implementation; the program generates all modular polynomials with prime level $l < B$. We compute all modular polynomials simultaneously because we only have to compute the powers of $\bar{j}$ once.

For given B defined as above, the exponents in $\bar{j}(\tau)$ range from $-B^2$ to B^2 because we need to compute $\bar{j}^i(\tau)$ for $i = 2, \cdots, B$. Thus, as we double B, we end up with quadruple size of $\bar{j}(\tau)$.

Magma stores Φ_l over $\mathbb{Z}$ for prime $l \leq 59$. We have verified that our $\bar{\Phi}_l(x, y)$ from Algorithm 1.2 and those stored in Magma are equal for each prime $l \leq 59$.

1.5 Analysis of the Exponents of $\bar{\Phi}_l(x, y)$

In this section, we analyze the exponents of classical modular polynomial $\bar{\Phi}_l(x, y)$ over $\mathbb{Z}_2$. In particular we are interested in the number of non-zero coefficients of $\bar{\Phi}_l(x, y)$ for given l. That is because in Chapter 2 we develop an algorithm that often performs substitutions of elements of $\mathbb{F}_{2^m}$ for y in $\bar{\Phi}_l(x, y)$, and the number of terms of $\bar{\Phi}_l(x, y)$ influences the running time for this part of the algorithm. The results of this section are new.

Throughout this section, we let l be an odd prime. Let L be the truncated Laurent series in q that is initialized on line 5 and updated on either line 11 or 13 through iterations of Algorithm 1.2. Notice that L changes in each iteration. Eventually, when the algorithm terminates, L only has positive exponent terms. We let $p(L)$ be the least exponent with

nonzero coefficient of L for each iteration as on line 7 of Algorithm 1.2. Hence $p(L)$ increases in each iteration by adding $\bar{j}^i(l\tau)\bar{j}^k(\tau)$ where the leading term of this product is $q^{p(L)}$ (so that $q^{p(L)}$ will vanish).

Theorem 1.5.1. *In each iteration of Algorithm 1.2, all terms of L have degree congruent to $-(l+1) \pmod 8$.*

Proof. For brevity, we often skip the words "congruent to" when no confusion will arise. For example, we abbreviate "X is congruent to $5 \pmod 8$" as "X is $5 \pmod 8$."

First, we examine

$$L = \bar{j}^{l+1}(l\tau) + \bar{j}^{l+1}(\tau) + \bar{j}^l(l\tau)\bar{j}^l(\tau)$$

on line 5 of Algorithm 1.2. By Lemma 1.4.2 and equation (1.2), we have

$$\bar{j}(\tau) = \frac{1}{q}\sum_{n\geq 0} s_n q^{8n} \text{ and } \bar{j}(l\tau) = \frac{1}{q^l}\sum_{n\geq 0} s_n q^{8ln}.$$

The degree of each term of $\bar{j}(\tau)$ and $\bar{j}(l\tau)$ is $-1 \pmod 8$ and $-l \pmod 8$, respectively. Then, each term of $\bar{j}^{l+1}(\tau)$, $\bar{j}^{l+1}(l\tau)$ and $\bar{j}^l(\tau)\bar{j}^l(l\tau)$ has degree $-(l+1) \pmod 8$, $-l(l+1) \pmod 8$ and $-l(l+1) \pmod 8$, respectively. This shows all terms of L have degree $a \pmod 8$ when L is initialized.

We have $l^2 \equiv 1 \pmod 8$ for odd l. Therefore,

$$(l-1)(l+1) \equiv 0 \pmod 8 \text{ and}$$
$$-(l+1) \equiv -l(l+1) \pmod 8.$$

Suppose all terms of L have degree $-(l+1) \pmod 8$ at the start of the n-th iteration for some $n \geq 0$. We want to show all terms of L have degree $-(l+1) \pmod 8$ at the start of the $(n+1)$-st iteration. By line 11 or 13 of Algorithm 1.2, L is updated in the n-th iteration, and the least exponent term of L, whose degree is $p(L)$, is cancelled after updating L because $p(L) = -(il+k)$ from line 8. To reduce ambiguity, we denote by $newL$ the value of L after updating in the n-th iteration. Thus, $newL$ is the value of L at the beginning of the $(n+1)$-st iteration.

Case i) When $i = k$ on line 10 in the n-th iteration, we have

$$newL = L + \bar{j}^i(\tau)\bar{j}^i(l\tau)$$

by line 11. Each term of $\bar{j}^i(\tau)$ and $\bar{j}^i(l\tau)$ has degree $-i \pmod 8$ and $-il \pmod 8$, respectively. Then, all terms of $\bar{j}^i(\tau)\bar{j}^i(l\tau)$ have degree $-(il+i) \pmod 8$. We know L and $\bar{j}^i(\tau)\bar{j}^i(l\tau)$ have term $q^{p(L)}$ and all terms of L have degree $-(l+1) \pmod 8$ by inductive hypothesis. Thus, $-(il+i) \equiv -(l+1) \pmod 8$. This shows that each term of $newL$ has degree $-(l+1) \pmod 8$.

Case ii) When $i \neq k$ on line 10 in the n-th iteration, we have

$$newL = L + \bar{j}^k(\tau)\bar{j}^i(l\tau) + \bar{j}^i(\tau)\bar{j}^k(l\tau)$$

by line 13. We apply the same reasoning as in the case $i = k$. Then, we conclude all terms of $\bar{j}^k(\tau)\bar{j}^i(l\tau)$ and $\bar{j}^i(\tau)\bar{j}^k(l\tau)$ have degree $-(il+k) \pmod 8$ and $-(kl+i) \pmod 8$, respectively. Since $p(L) = -(il+k) = -(l+1) \pmod 8$ from line 8, we conclude that all terms of $\bar{j}^k(\tau)\bar{j}^i(l\tau)$ have degree $-(l+1) \pmod 8$. If we show $-(kl+i) = -(l+1) \pmod 8$ for given $-(il+k) = -(l+1) \pmod 8$, then all terms of $newL$ have degree $-(l+1) \pmod 8$. As mentioned earlier we have $il + k \geq kl + i$ for all i, k because $0 \leq k \leq i \leq l$.

Now, we show that $-(il+k) = -l(l+1) \pmod 8$ implies $-(kl+i) = -(l+1) \pmod 8$ for each possible $l \pmod 8$.

We have $-(il+k) \equiv -(l+1) \pmod 8$. Therefore,

$$\begin{aligned} 1-k &\equiv l(i-1) \pmod 8 \\ l(1-k) &\equiv l^2(i-1) \equiv i-1 \pmod 8 \text{ (since } l^2 \equiv 1 \pmod 8 \text{ for odd } l) \\ -(kl+i) &\equiv -(l+1) \pmod 8. \end{aligned}$$

We have shown that $-(il+k) \equiv -(kl+i) \equiv -(l+1) \pmod 8$. This implies all terms $newL$ have degree $-(l+1) \pmod 8$.

Therefore, we conclude that L only has terms with degree $-(l+1) \pmod 8$ for all iterations by the principle of induction. □

Let us state the following special cases of Theorem 1.5.1 as separate statements, since they will be used to make the first main statement of this section (Corollary 1.5.4).

Theorem 1.5.2. *We have*

$$p(L) \equiv -(l+1) \pmod 8 \tag{1.19}$$

for all iterations.

Theorem 1.5.3. *Let i and k be as defined on line 8 of Algorithm 1.2. We have*

$$il + k \equiv l + 1 \pmod 8. \tag{1.20}$$

Recall that by Proposition 1.4.5, the l-th modular polynomial $\bar{\Phi}_l(x, y)$ over $\mathbb{Z}_2$ where l is prime has the form

$$\bar{\Phi}_l(x, y) = x^{l+1} + x^l y^l + y^{l+1} + \sum_{i,k \le l,\quad i+k<2l} a_{ik} x^i y^k, \quad a_{ik} \in \mathbb{Z}_2. \tag{1.21}$$

Using Theorems 1.5.2 and 1.5.3 and Algorithm 1.2, we derive our first statement about the coefficients a_{ik} in (1.21).

Corollary 1.5.4. *Let l be a prime. If $a_{ik} = 1$, then $il + k \equiv (l + 1) \pmod 8$.*

After testing all prime levels $l < 2000$, we state the following conjecture:

Conjecture 1.5.5. *If $0 \le i + k < l + 1$, then $a_{ik} = 0$.*

The conjecture is illustrated in Figures 1.1 through 1.4. In each figure, each dot represents the coefficient 1 in $\bar{\Phi}_l(x, y)$. On the horizontal axis, exponents of x range from 0 to $l + 1$. On the vertical axis, exponents of y range from 0 to $l + 1$. Thus, for example, Conjecture 1.5.5 corresponds to the white triangle below the main diagonal in each figure.

We also observe a fractal-like structure of the white triangles in these figures. The second white triangle begins approximately at the lines $i = l - 2^m$ and $k = l - 2^m$ where $m = \lfloor \log_2 l \rfloor$ and i, k denote the two coordinates of the plane. We observe that $a_{ik} = 0$ if $i, k \in (l - 2^m, l + 1]$ and $i + k < l + 1 + 8n$ for some positive integer n except the points $(2^m, l + 1 - 2^m)$ and $(l + 1 - 2^m, 2^m)$. We have tried to find a formula for n but we were not able to obtain one.

Throughout the rest of this section we continue using a_{ik} to denote the coefficient of $x^i y^k$ in $\bar{\Phi}_l(x, y)$.

1.5.1 Prime Level l with $l \equiv 1 \pmod 8$

Theorem 1.5.6. *Assume that $l \equiv 1 \pmod 8$. If $a_{ik} = 1$, then $i + k \equiv 2 \pmod 8$.*

Proof. Proof immediately follows from Corollary 1.5.4 with $l \equiv 1 \pmod 8$. □

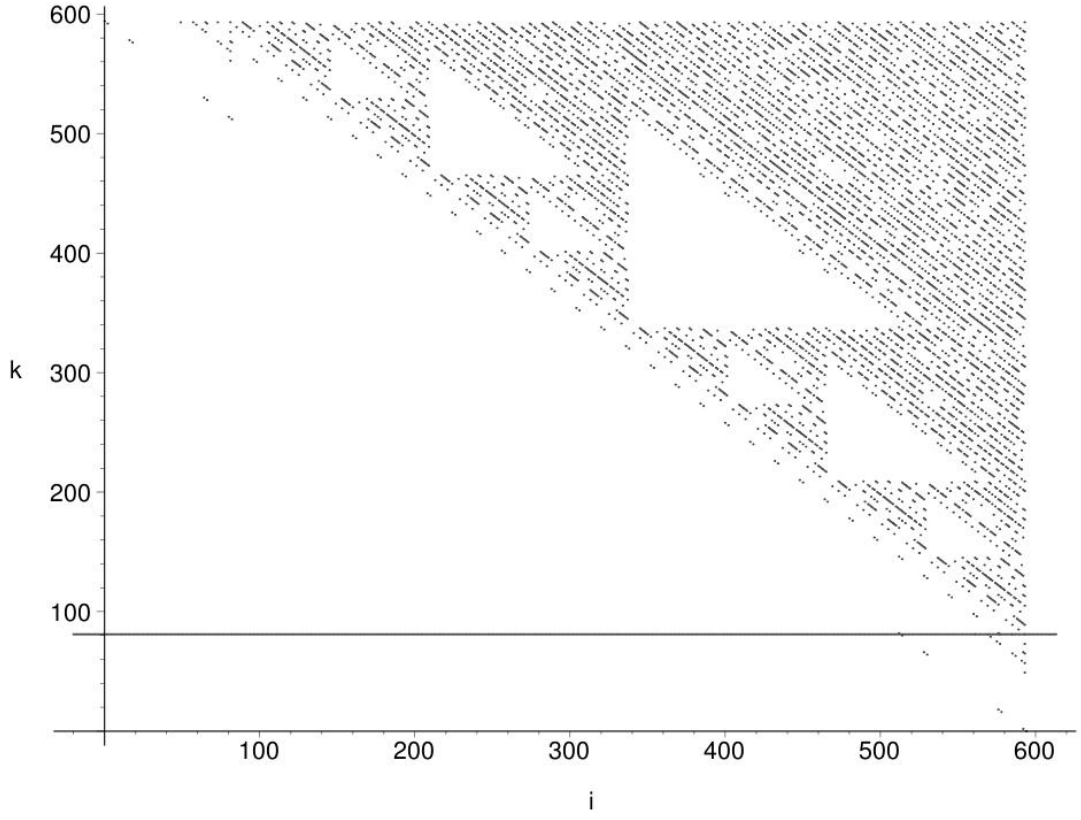

Figure 1.1: The coefficients of $\bar{\Phi}_l(x, y)$ for $l = 593$, the $l \equiv 1 \pmod 8$ case

We plot the coefficients of $\bar{\Phi}_{593}(x, y)$ in Figure 1.1. (The rules for plotting were explained several paragraphs above.) Because of Theorem 1.5.6, we see that dots in the figure belong to lines with slope -1 that are at vertical distance 8 apart.

1.5.2 Prime Level l with $l \equiv 3 \pmod 8$

Theorem 1.5.7. *Assume that* $l \equiv 3 \pmod 8$. *If* $a_{ik} = 1$, *then* $3i + k \equiv 4 \pmod 8$.

Proof. Proof immediately follows from Corollary 1.5.4 with $l \equiv 3 \pmod 8$. □

Theorem 1.5.8. *Assume that* $l \equiv 3 \pmod 8$. *If* $a_{ik} = 1$, *then either* $i - k \equiv 0 \pmod 8$ *and* $i + k \equiv 2 \pmod 4$, *or* $i - k \equiv 4 \pmod 8$ *and* $i + k \equiv 0 \pmod 4$.

Proof. By Theorem 1.5.7, we have $3i + k \equiv 4 \pmod 8$. Then

$$i + k \equiv 4 - 2i \pmod 8$$
$$i - k \equiv 4i - 4 \pmod 8$$

Case 1) When i is even, we have

$$i + k \equiv 0 \pmod 4 \text{ and } i - k \equiv 4 \pmod 8.$$

Case 2) When i is odd, we have

$$i + k \equiv 2 \pmod 4 \text{ and } i - k \equiv 0 \pmod 8.$$

□

We plot the coefficients of $\bar{\Phi}_{587}(x, y)$ in Figure 1.2.

1.5.3 Prime Level l with $l \equiv 5 \pmod 8$

Theorem 1.5.9. *Assume that* $l \equiv 5 \pmod 8$. *If* $a_{ik} = 1$, *then* $5i + k \equiv 6 \pmod 8$.

Proof. Proof immediately follows from Corollary 1.5.4 with $l \equiv 5 \pmod 8$. □

Theorem 1.5.10. *Assume that* $l \equiv 5 \pmod 8$. *If* $a_{ik} = 1$, *then either* $i + k \equiv 2 \pmod 8$ *and* $i - k \equiv 0 \pmod 4$, *or* $i + k \equiv 6 \pmod 8$ *and* $i - k \equiv 2 \pmod 4$.

Proof. By Theorem 1.5.9, we have $5i + k \equiv 6 \pmod 8$. Then

$$i + k \equiv 6 - 4i \pmod 8$$
$$i - k \equiv 6i - 6 \pmod 8$$

Case 1) When i is even, we have

$$i - k \equiv 2 \pmod 4 \text{ and } i + k \equiv 6 \pmod 8.$$

Case 2) When i is odd, we have

$$i - k \equiv 0 \pmod 4 \text{ and } i + k \equiv 2 \pmod 8.$$

□

We plot the coefficients of $\bar{\Phi}_{557}(x, y)$ in Figure 1.3.

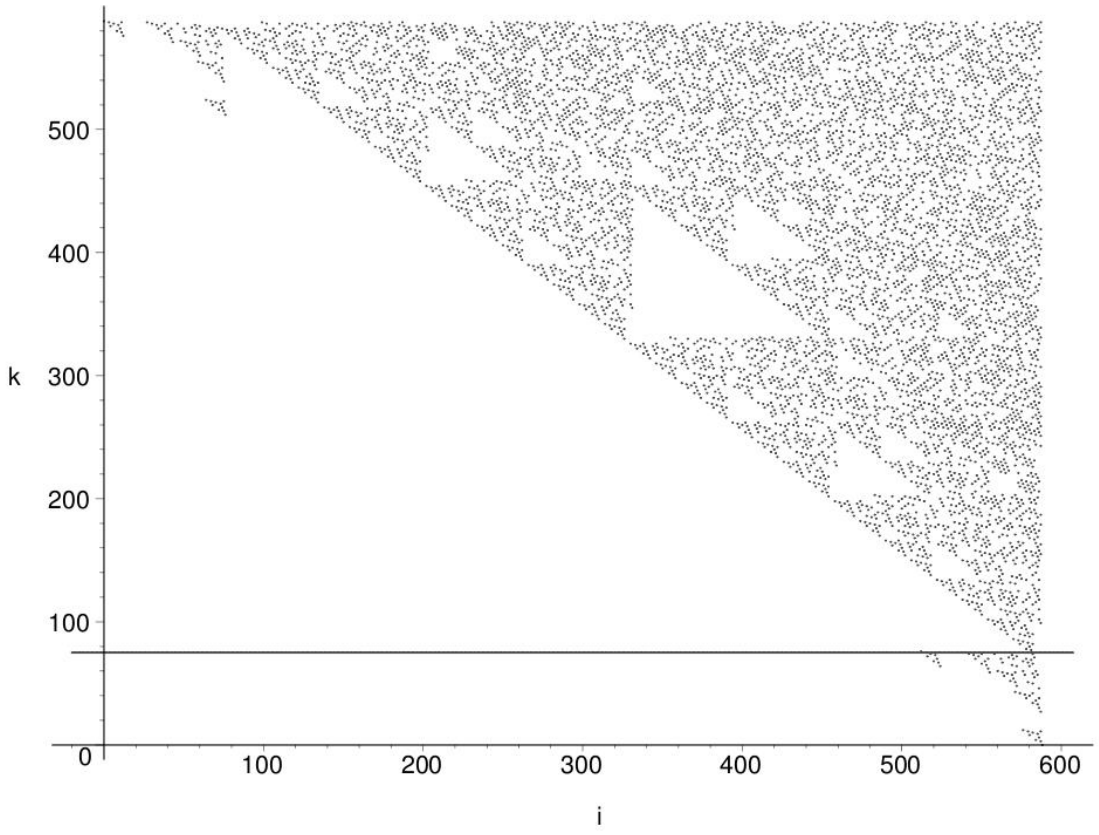

Figure 1.2: The coefficients of $\bar{\Phi}_l(x, y)$ for $l = 587$, the $l \equiv 3 \pmod 8$ case

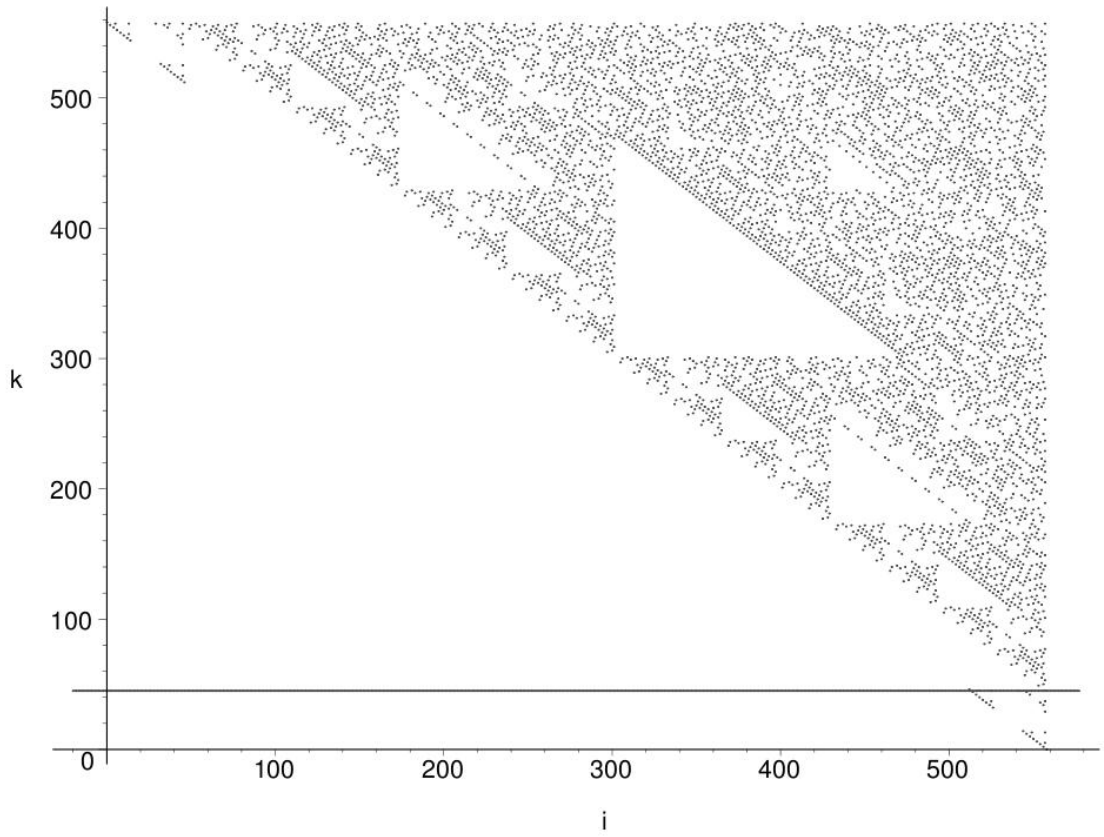

Figure 1.3: The coefficients of $\bar{\Phi}_l(x, y)$ for $l = 557$, the $l \equiv 5 \pmod 8$ case

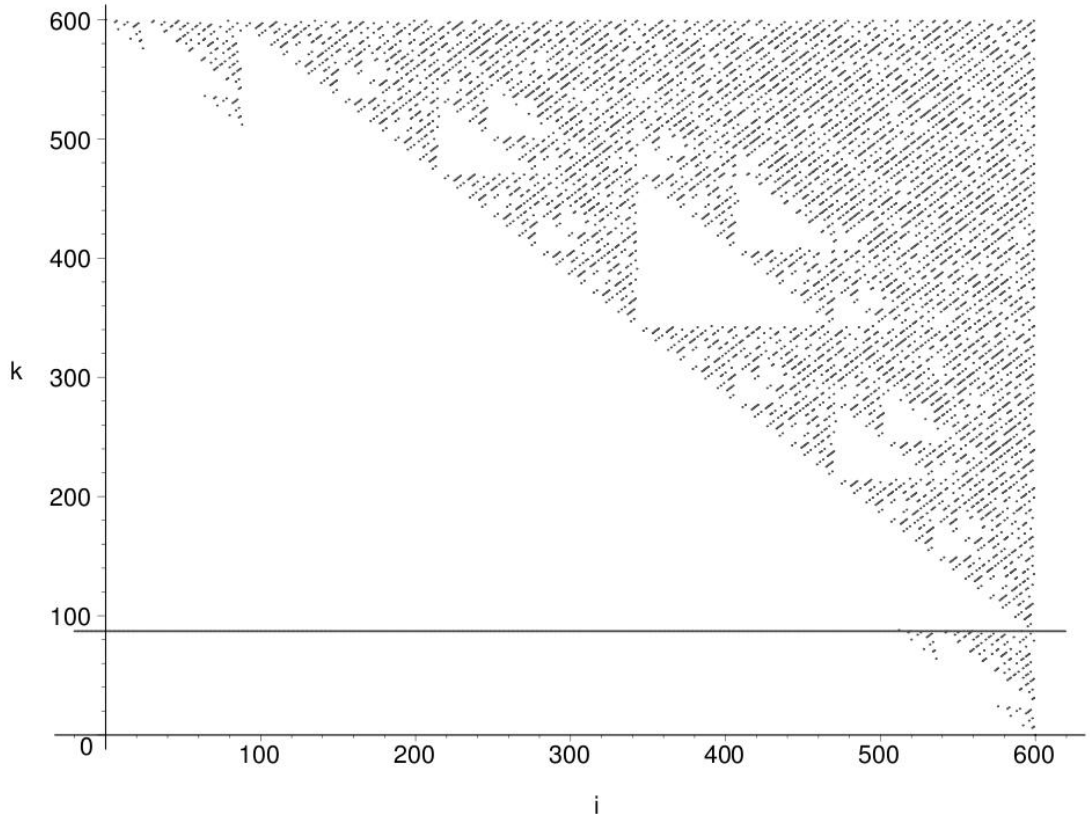

Figure 1.4: The coefficients of $\bar{\Phi}_l(x, y)$ for $l = 599$, the $l \equiv 7 \pmod 8$ case

1.5.4 Prime Level l with $l \equiv 7 \pmod 8$

Theorem 1.5.11. *Assume that* $l \equiv 7 \pmod 8$. *If* $a_{ik} = 1$, *then* $i - k \equiv 0 \pmod 8$.

Proof. Proof immediately follows from Corollary 1.5.4 with $l \equiv -1 \pmod 8$. □

We plot the coefficients of $\bar{\Phi}_{599}(x, y)$ in Figure 1.4. Because of Theorem 1.5.11, we see that dots in the figure belong to lines with slope 1 that are at vertical distance 8 apart.

1.6 Density of $\bar{\Phi}_l(x, y)$

Originally we defined the l-th modular polynomial $\Phi_l(x, y)$ as a polynomial over $\mathbb{Z}$. These polynomials are dense and their coefficients grow very quickly with l. They are also

expensive to compute. For that reason, Magma stores $\Phi_l(x, y)$ over $\mathbb{Z}$ only for prime levels $l \leq 59$ and for composite levels $l \leq 16$. On the other hand, modular polynomials $\bar{\Phi}_l(x, y)$ become very manageable when considered over $\mathbb{Z}_2$. Not only are the coefficients limited to one bit, but also the vast majority of terms in $\Phi_l(x, y)$ are 0 modulo 2. By observing patterns in plots of coefficients of $\bar{\Phi}_l(x, y)$ for prime $l < 2000$, we conjectured some properties of coefficients of $\bar{\Phi}_l(x, y)$, resulting in claims about the density of $\bar{\Phi}_l(x, y)$. We proved some of the results in the previous section, and in this section we derive a density result from them.

We show $\bar{\Phi}_p(x, y)$ and $\Phi_p(x, y)$ where $p \in \{2, 3, 5\}$. These polynomials can be obtained by using Magma.

Example 1.6.1. $\bar{\Phi}_2(x, y) = x^3 + x^2y^2 + xy + y^3$
$\Phi_2(x, y) = x^3 + y^3 - x^2y^2 + 1488(x^2y + xy^2) - 162000(x^2 + y^2) + 40773375xy$
$+ 8748000000(x + y) - 157464000000000$

Example 1.6.2. $\bar{\Phi}_3(x, y) = x^4 + x^3y^3 + y^4$
$\Phi_3(x, y) = x^4 + y^4 - x^3y^3 + 2232(x^3y^2 + x^2y^3) - 1069956(x^3y + xy^3)$
$+ 36864000(x^3 + y^3) + 2587918086x^2y^2 + 8900222976000(x^2y + xy^2)$
$+ 452984832000000(x^2 + y^2) - 770845966336000000xy$
$+ 1855425871872000000000(x + y)$

Example 1.6.3. $\bar{\Phi}_5(x, y) = x^6 + x^5y^5 + x^4y^2 + x^2y^4 + y^6$
$\Phi_5(x, y) = x^6 + y^6 - x^5y^5 + 3720(x^5y^4 + x^4y^5) - 4550940(x^5y^3 + x^3y^5)$
$+ 2028551200(x^5y^2 + x^2y^5) - 246683410950(x^5y + xy^5) + 1963211489280(x^5 + y^5)$
$+ 1665999364600x^4y^4 + 107878928185336800(x^4y^3 + x^3y^4)$
$+ 38308360977981121537 5(x^4y^2 + x^2y^4) + 1285417989068288163840 00(x^4y + xy^4)$
$+ 1284733132841424456253440(x^4 + y^4) - 4412069655129148352461 00x^3y^3$
$+ 2689848885838073157741772800 0(x^3y^2 + x^2y^3)$
$- 1924579346189282996551082311680 00(x^3y + xy^3)$
$+ 2802447778284395278043215652978688 00(x^3 + y^3)$
$+ 5110941777552418083110765199360000x^2y^2$
$+ 3655473658394962929570647233265664 0000(x^2y + xy^2)$
$+ 6692500042627997708487149415015068467200(x^2 + y^2)$
$- 2640734570766205962597157902479787829493 76xy$

$l \pmod 8$	Results
1	$i + k \equiv 2 \pmod 8$
3	$i - k \equiv 0 \pmod 8$ and $i + k \equiv 2 \pmod 4$, or $i - k \equiv 4 \pmod 8$ and $i + k \equiv 0 \pmod 4$
5	$i + k \equiv 2 \pmod 8$ and $i - k \equiv 0 \pmod 4$, or $i + k \equiv 6 \pmod 8$ and $i - k \equiv 2 \pmod 4$
7	$i - k \equiv 0 \pmod 8$

Table 1.2: Summary of relations between i and k for each l

$+ 532743308034244254504201602733565091512320000(x + y)$
$+ 14135994715472135869775347469107136275100467200 0$

Recall that, by Proposition 1.4.5, for prime l the l-th modular polynomial $\bar{\Phi}_l(x, y)$ over $\mathbb{Z}_2$ has the form

$$\bar{\Phi}_l(x, y) = x^{l+1} + y^{l+1} + \sum_{i,k \le l} a_{ik} x^i y^k, a_{ik} \in \mathbb{Z}_2. \tag{1.22}$$

Notice that we treat $x^l y^l$ as a special case on line 6 of Algorithm 1.2 in order to simplify the algorithm.

Definition 1.6.4. *We define the* density *of $\bar{\Phi}_l(x, y)$ defined in Proposition 1.4.5 as*

$$\frac{\textit{the number of nonzero } a_{ik} \textit{ in equation (1.22)}}{(l+1)^2 = \textit{ the maximum number of terms of } a_{ik}}.$$

Theorem 1.6.5. *For prime l the density of $\bar{\Phi}_l(x, y)$ is at most* $\frac{1}{8} + \frac{14}{l+1}$.

Proof. Table 1.2 shows the summary of results from Section 1.5. For each possible residue class $l \bmod 8$, it shows necessary conditions that i and k must satisfy if $a_{ik} = 1$. (See Theorems 1.5.6, 1.5.8, 1.5.10 and 1.5.11.)

Let s be the largest integer such that $8s \le l + 1$. Let us consider the 8×8 square

$$Q = [8t, 8t + 7] \times [8u, 8u + 7] \text{ where } t, u \in \{0, 1, \ldots, s - 1\}.$$

By Theorems 1.5.6, 1.5.8, 1.5.10 and 1.5.11, if $a_{8t+v,8u+w} = 1$ and $v, w \in \{0, \ldots, 7\}$, then (v, w) must be one of the pairs listed in Table 1.3. Hence, Q contributes at most 8 nonzero coefficients a_{ij}. The square $[0, l] \times [0, l]$ splits into $s \cdot s = s^2$ squares Q as introduced

$l \equiv 1 \pmod 8$	$l \equiv 3 \pmod 8$	$l \equiv 5 \pmod 8$	$l \equiv 7 \pmod 8$
(0, 2)	(0, 4)	(0, 6)	(0, 0)
(1, 1)	(1, 1)	(1, 1)	(1, 1)
(2, 0)	(2, 6)	(2, 4)	(2, 2)
(3, 7)	(3, 3)	(3, 7)	(3, 3)
(4, 6)	(4, 0)	(4, 2)	(4, 4)
(5, 5)	(5, 5)	(5, 5)	(5, 5)
(6, 4)	(6, 2)	(6, 0)	(6, 6)
(7, 3)	(7, 3)	(7, 3)	(7, 7)

Table 1.3: All possible pairs (v, w) in Theorem 1.6.5

above and two overlapping rectangular regions (outside this big square) that cover $l + 1$ lattice points in one direction and 7 lattice points in the other direction. Because we have $8s \leq l + 1$, we get $8s^2 \leq \frac{(l+1)^2}{8}$. This gives the upper bound on the density of $\bar{\Phi}_l$ as

$$\frac{8s^2 + (2)(7)(l+1)}{(l+1)^2} \leq \frac{1}{8} + \frac{14}{l+1}.$$

□

Asymptotically this gives the upper bound of $\frac{1}{8}$ on the density of $\bar{\Phi}_l(x, y)$.

If Conjecture 1.5.5 holds, then it implies the following result:

Conjecture 1.6.6. *The density of a modular polynomial with prime level l over $\mathbb{Z}_2$ is at most $\frac{1}{16}$ asymptotically.*

Based on data computed for prime $l < 2000$ it appears that the density of the modular polynomial with prime level l settles around the value $\frac{1}{48}$. However, it is not clear if we have enough evidence to make a stronger statement about this. Figure 1.5 shows density of $\bar{\Phi}_l$ where l is a prime between 500 and 2000.

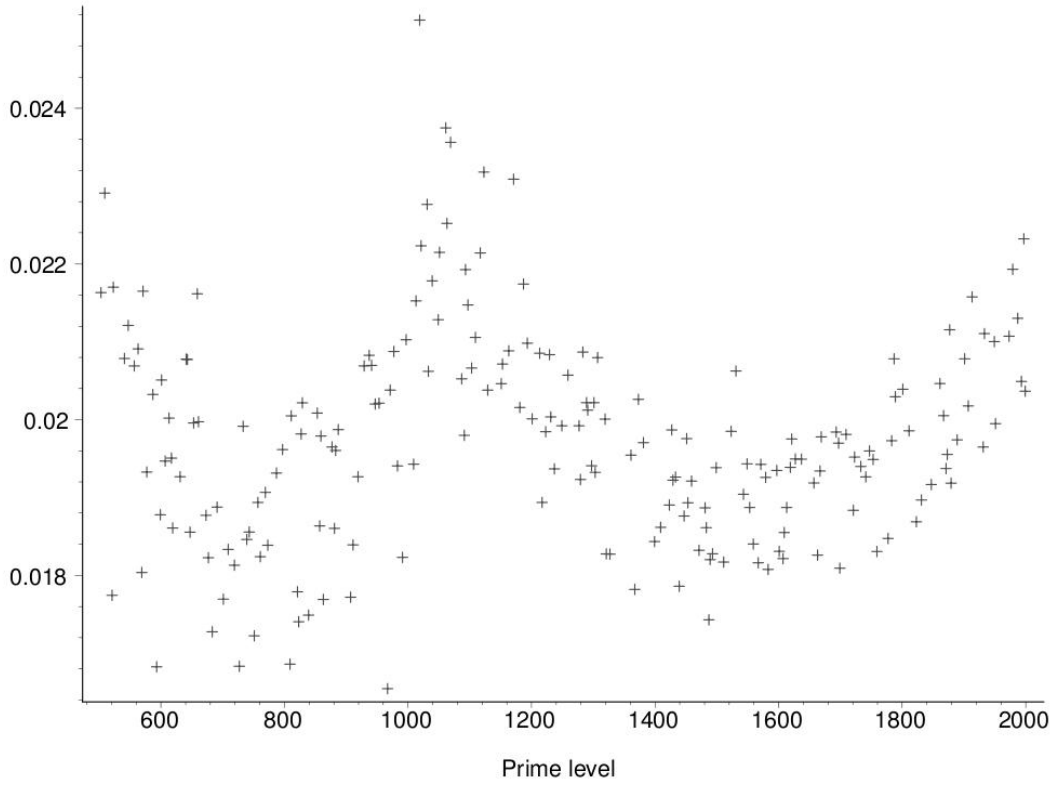

Figure 1.5: Density of $\bar{\Phi}_l$ for $500 < l < 2000$

Chapter 2

Kloosterman Zeros

Kloosterman sums have recently enjoyed much attention. Some of this interest is due to their applications in cryptography and coding theory. For instance, a Kloosterman zero can be used to construct one hyperbent (bent) function in even (odd) characteristic, respectively. Since the bent and hyperbent functions also have to satisfy some other criteria required in applications, it is desired to have a large set of bent functions to choose from. This motivates the task of listing all Kloosterman zeros in the given field.

In this chapter, we recall what Kloosterman zeros are and how we compute one Kloosterman zero and the total number of Kloosterman zeros in the field $\mathbb{F}_{2^m}$ for some fixed m. Moreover, we develop algorithms that list all Kloosterman zeros and all minimal polynomials of binary Kloosterman zeros for this field.

For a nonzero element in $\mathbb{F}_{p^m}$ where $p \in \{2, 3\}$ it is known that there is a relation between Kloosterman sum of this element and the number of points with coordinates in $\mathbb{F}_{p^m}$ on a certain elliptic curve [20, 27].

We developed our algorithms by combining concepts involving Kloosterman sums, elliptic curves and modular polynomials. We use classical modular polynomials which are precomputed in Algorithm 1.2. Applying our new algorithms, we have computed all Kloosterman zeros in $\mathbb{F}_{2^m}$ up to $m \leq 63$. This is a significant improvement because this list is known only up to $m \leq 14$ in the literature (see paper [5] by Charpin and Gong). Since there are no known efficient algorithms for listing all Kloosterman zeros in larger fields, our novel algorithms can be blueprints for listing them.

Throughout this chapter, we let $\mathbb{F}_q$ be the finite field of order

$$q = p^m$$

where p is a prime and m is a positive integer. Let $\mathbb{F}_p$ be the subfield of $\mathbb{F}_q$ of order p and $\mathbb{F}_q^* = \mathbb{F}_q \backslash \{0\}$.

2.1 Background and Definitions

In this section, we give definitions and background information which will be used in later sections. We recommend [28] as an accessible reference for these topics.

Throughout this section, we let $a \in \mathbb{F}_q$ and use n to denote the smallest positive integer such that $a^{p^n} = a$ (so that $a^{p^i} \neq a$ for $1 \leq i < n$). Notice that n divides m because $a^{p^m} = a$.

Lemma 2.1.1. *We have*

$$a^{p^j} \neq a^{p^k} \text{ for } 1 \leq j < k < n.$$

Proof. Suppose $a^{p^j} = a^{p^k}$ and $1 \leq j < k < n$. We have

$$a = a^{p^n} = a^{p^{k+(n-k)}} = (a^{p^k})^{p^{n-k}} = (a^{p^j})^{p^{n-k}} = a^{p^{j+n-k}}.$$

This contradicts the minimality condition for n because $1 \leq j + n - k < n$. □

Definition 2.1.2. *We define the* minimal polynomial $f(x) \in \mathbb{F}_p[x]$ *of* a *over* $\mathbb{F}_p$ *to be the monic polynomial of the smallest degree over* $\mathbb{F}_p$ *such that* $f(a) = 0$.

We know that the minimal polynomial of a over $\mathbb{F}_p$ is unique [28].

Theorem 2.1.3. *[28] Any polynomial* $g(x) \in \mathbb{F}_p[x]$ *with* $g(a) = 0$ *is a multiple of the minimal polynomial of* a *over* $\mathbb{F}_p$.

Definition 2.1.4. *We define the* algebraic conjugate(s) *of* a *over* $\mathbb{F}_p$ *to be the root(s) of the minimal polynomial of* a *over* $\mathbb{F}_p$.

Notice that conjugate(s) may not in $\mathbb{F}_p$.

Definition 2.1.5. *The* Frobenius automorphism *on* $\mathbb{F}_q$ *is the map* $\phi : \mathbb{F}_q \to \mathbb{F}_q$ *defined by*

$$\phi(a) = a^p \quad \text{for all } \ a \in \mathbb{F}_q. \tag{2.1}$$

Proposition 2.1.6. *The Frobenius automorphism $\phi : \mathbb{F}_q \to \mathbb{F}_q$ is a field automorphism.*

Proof. For all $a, b \in \mathbb{F}_q$, we have

$$\begin{aligned} \phi(0) &= 0^p = 0, \\ \phi(1) &= 1^p = 1, \\ \phi(a+b) &= (a+b)^p = a^p + b^p = \phi(a) + \phi(b) \quad \text{and} \\ \phi(ab) &= (ab)^p = a^p b^p = \phi(a)\phi(b). \end{aligned}$$

Now we show ϕ is bijection.

Let $a, b \in \mathbb{F}_q$ be arbitrary. Suppose $\phi(a) = \phi(b)$. Then $a^p = b^p$ and $(a^p)^{p^{m-1}} = (b^p)^{p^{m-1}}$. Thus $a = b$. Therefore, ϕ is one-to-one.

Let $b \in \mathbb{F}_q$ be arbitrary. Let $a = b^{p^{m-1}} \in \mathbb{F}_q$. Then $\phi(a) = \phi(b^{p^{m-1}}) = b^{p^m} = b$. Therefore ϕ is onto. Thus ϕ is bijection.

Therefore, ϕ is a field automorphism. □

Corollary 2.1.7. *[28] All conjugates of a over $\mathbb{F}_p$ are obtained through repeating the Frobenius automorphism on $\mathbb{F}_q$.*

Proof. Let $f(x) \in \mathbb{F}_p[x]$ be the minimal polynomial of a over $\mathbb{F}_p$ with degree l. Then, we have $f(a) = 0$. We can write

$$f(x) = c_l x^l + c_{l-1} x^{l-1} + \cdots + c_1 x + c_0 \text{ where } c_i \in \mathbb{F}_p.$$

Because $(\alpha + \beta)^p = \alpha^p + \beta^p$ for all $\alpha, \beta \in \mathbb{F}_q$, we have

$$0 = f(a)^p = c_l^p (a^p)^l + c_{l-1}^p (a^p)^{l-1} + \cdots + c_1^p (a^p) + c_0^p.$$

Since $c_i \in \mathbb{F}_p$ and $c_i^p = c_i$, we have

$$0 = c_l (a^p)^l + c_{l-1} (a^p)^{l-1} + \cdots + c_1 (a^p) + c_0 = f(a^p).$$

Thus, a^p is a root of f.

By definition of n, we have $a^{p^i} \neq a^{p^j}$ for $1 \leq i < j < n$ and $a^{p^n} = a$. We conclude that $a, a^p, \cdots, a^{p^{n-1}}$ are distinct roots of f by applying Frobenius automorphism $n - 1$ times.

We need to show that

$$f(x) = \prod_{i=0}^{n-1}(x - a^{p^i}).$$

Let $g(x) = \prod_{i=0}^{n-1}(x - a^{p^i})$. It is clear that $g(x)|f(x)$ over $\mathbb{F}_q$ by above. We can write

$$g(x) = d_0 + d_1x + \cdots + d_{n-1}x^{n-1} + x^n$$

where $d_i \in \mathbb{F}_q$. Then, we have

$$(g(x))^p = \prod_{i=0}^{n-1}(x - a^{p^i})^p = \prod_{i=0}^{n-1}(x^p - a^{p^{i+1}}) = \prod_{i=0}^{n-1}(x^p - a^{p^i}) = g(x^p).$$

On the other hand, we have

$$\begin{aligned}(g(x))^p &= (d_0 + d_1x + \cdots + d_{n-1}x^{n-1} + x^n)^p \\ &= d_0^p + d_1^p x^p + \cdots + d_{n-1}^p x^{p(n-1)} + x^{pn},\end{aligned}$$

and

$$g(x^p) = d_0 + d_1x^p + \cdots + d_{n-1}x^{p(n-1)} + x^{pn}.$$

Since $(g(x))^p = g(x^p)$, all $d_i \in \mathbb{F}_p$. Therefore, $g(x)$ is a polynomial over $\mathbb{F}_p$. Since both $f(x)$ and $g(x)$ are both monic polynomials, $g(x)$ is the minimal polynomial of a over $\mathbb{F}_p$ by definition of the minimal polynomial. □

Theorem 2.1.8. *The minimal polynomial $f(x)$ of a over $\mathbb{F}_p$ can be written as*

$$f(x) = \prod_{i=0}^{n-1}(x - a^{p^i}). \tag{2.2}$$

Proof. Proof follows from the Corollary 2.1.7. □

Definition 2.1.9. *We define $Tr : \mathbb{F}_q \to \mathbb{F}_p$, the* absolute trace *function, as*

$$Tr(x) = x + x^p + x^{p^2} + \cdots + x^{p^{m-1}}. \tag{2.3}$$

Proposition 2.1.10. *[22] For all $\alpha, \beta \in \mathbb{F}_q$, we have*

$$\begin{aligned}Tr(-\alpha) &= -Tr(\alpha), \\ Tr(\alpha^p) &= Tr(\alpha), \\ Tr(\alpha + \beta) &= Tr(\alpha) + Tr(\beta),\end{aligned}$$

Theorem 2.1.11. *[22] For each $\gamma \in \mathbb{F}_p$, there are exactly p^{m-1} elements $u \in \mathbb{F}_q$ such that $Tr(u) = \gamma$.*

Definition 2.1.12. *Let $q = p^m$ where p is prime. The* Kloosterman sum *on $\mathbb{F}_q$ is the mapping $K_q : \mathbb{F}_q \to \mathbb{R}$ defined by*

$$\mathcal{K}_q(a) = 1 + \sum_{x \in \mathbb{F}_q^*} \zeta_p^{Tr(x^{-1}+ax)} \tag{2.4}$$

where $\zeta_p = e^{2\pi i/p}$ is a primitive p-th root of unity.

In some references, the same mapping is defined by $\mathcal{K}_q(a) = \sum_{x \in \mathbb{F}_q} \zeta_p^{Tr(x^{-1}+ax)}$ with the proviso that $Tr(0^{-1}) = 0$.

For all $x \in \mathbb{F}_q^*$, we have

$$Tr((-x)^{-1} + a(-x)) = -Tr(x^{-1} + ax).$$

Then, we have for all $x \in \mathbb{F}_q^*$

$$Tr(x^{-1} + ax) = b \in \mathbb{F}_p \text{ if and only if } Tr((-x)^{-1} + a(-x)) = -b.$$

Since $\zeta_p^b + \zeta_p^{-b} \in \mathbb{R}$, we have

$$\mathcal{K}_q(a) = 1 + \sum_{x \in \mathbb{F}_q^*} \zeta_p^{Tr(x^{-1}+ax)} \in \mathbb{R}.$$

Definition 2.1.13. *We say a is a* zero of the Kloosterman sum $\mathcal{K}_q$, *or a* Kloosterman zero *if $a \in \mathbb{F}_q^*$ and $\mathcal{K}_q(a) = 0$.*

Proposition 2.1.14. *We have*

$$\mathcal{K}_q(0) = 0 \text{ for all } q.$$

$$\mathcal{K}_{p^m}(a) \in \mathbb{Z} \text{ where } p \in \{2, 3\}.$$

Proof. We have

$$\begin{aligned} \mathcal{K}_q(0) &= \sum_{x \in \mathbb{F}_q} \zeta_p^{Tr(x^{-1})} \text{ with } Tr(0^{-1}) = 0 \\ &= p^{m-1}(\zeta_p^0 + \zeta_p^1 + \cdots + \zeta_p^{p-1}) = 0. \end{aligned}$$

by Theorem 2.1.11. We have $\zeta_2 = -1$ and $\zeta_3 = \frac{-1+\sqrt{3}i}{2}$. For $p = 2$, we have

$$\mathcal{K}_{2^m}(a) = 1 + \sum_{x \in \mathbb{F}_{2^m}^*} (-1)^{Tr(x^{-1}+ax)} \in \mathbb{Z}.$$

For $p = 3$, we have

$$\zeta_3^{Tr(x^{-1}+ax)} = \zeta_3 \text{ if and only if } \zeta_3^{Tr((-x)^{-1}+a(-x))} = \zeta_3^2.$$

We know $\zeta_3 + \zeta_3^2 = -1$. Thus,

$$\mathcal{K}_{3^m}(a) = 1 + \sum_{x \in \mathbb{F}_{3^m}^*} \zeta_3^{Tr(x^{-1}+ax)} \in \mathbb{Z}.$$

□

Applying the Frobenius automorphism with the properties of the trace map, we can derive following lemma and corollary.

Lemma 2.1.15. *We have*

$$\mathcal{K}_q(a) = \mathcal{K}_q(a^p) \text{ for all } a \in \mathbb{F}_q. \tag{2.5}$$

Proof. We know $x \mapsto x^p$ is bijective on $\mathbb{F}_q$ and let $y = x^p \in \mathbb{F}_q$. Then, we have

$$Tr(x^{-1} + ax) = Tr((x^{-1} + ax)^p) = Tr((x^p)^{-1} + a^p x^p) = Tr(y^{-1} + a^p y).$$

Therefore, $\mathcal{K}_q(a^p) = \mathcal{K}_q(a)$ by definition. □

Corollary 2.1.16. *We have*

$$\mathcal{K}_q(a) = \mathcal{K}_q(a^p) = \mathcal{K}_q(a^{p^2}) = \cdots = \mathcal{K}_q(a^{p^{m-1}}) \text{ for all } a \in \mathbb{F}_q. \tag{2.6}$$

Theorem 2.1.17. *If a is a Kloosterman zero, then a^{p^i} is also a Kloosterman zero for $1 \leq i \leq m-1$.*

Proof. Proof immediately follows from Corollary 2.1.16. □

Kloosterman zeros have a significant role in cryptography because they are used in the construction of highly nonlinear functions. Binary Kloosterman zeros are used in the construction of a distinguished family of bent functions, which have some additional favorable properties (high algebraic degree).

Dillon conjectured [8] the existence of Kloosterman zeros in all fields of characteristic 2 except for $\mathbb{F}_2$, and he showed the construction of a bent function using a Kloosterman zero. Dillon's conjecture was proved by Lachaud and Wolfmann in [20]. Furthermore, Youssef and Gong [33] and Carlet and Gaborit [3] have shown that the bent functions constructed by Dillion in fact are hyperbent functions. A complete list of binary Kloosterman zeros in $\mathbb{F}_{2^m}$ for $m \leq 14$ is presented in Table 1 in [5]. We will give a more detailed discussion of binary Kloosterman zeros later in this chapter.

A *Boolean* function is any function from $\mathbb{F}_{2^n}$ to $\mathbb{F}_2$. The Hadamard transform of a Boolean function f is defined as

$$\hat{f}(a) = \sum_{x \in \mathbb{F}_{2^n}} (-1)^{f(x)+Tr(ax)}$$

where $a \in \mathbb{F}_{2^n}$. A Boolean function f is a *bent* function if $|\hat{f}(a)| = 2^{n/2}$ for all $a \in \mathbb{F}_{2^n}$; notice that n must be even for binary bent functions. The extended Hadamard transform of f is defined as

$$\hat{f}(a,k) = \sum_{x \in \mathbb{F}_{2^n}} (-1)^{f(x)+Tr(ax^k)}$$

where $a \in \mathbb{F}_{2^n}$ and $\gcd(k, 2^n - 1) = 1$. A Boolean function f is a *hyperbent* function if $|\hat{f}(a,k)| = 2^{n/2}$ for all $a \in \mathbb{F}_{2^n}$ and $\gcd(k, 2^n - 1) = 1$. Note that any hyperbent function must be bent (take $k = 1$).

Define a Boolean function from $\mathbb{F}_{2^n}$ to $\mathbb{F}_2$ to be

$$f_\lambda(x) = Tr(\lambda x^{2^m-1}), \quad \lambda \in \mathbb{F}_{2^m}^*.$$

Dillion [8] proved that f_λ is bent if and only if $\mathcal{K}_{2^m}(\lambda) = 0$. Moreover, Charpin and Gong [5] proved that f_λ is hyperbent if and only if $\mathcal{K}_{2^m}(\lambda) = 0$.

Kononen, Rinta-aho and Väänänen [19] proved that no Kloosterman zero exists in a field of characteristic greater than 3. Moreover, Lisoněk and Moisio [25] proved that there is no Kloosterman zero in $\mathbb{F}_q$ that belongs to a proper subfield of $\mathbb{F}_q$ except when $q = 16$

(this case occurs only if $p = 2$ and $m = 4$) with $\mathcal{K}_{16}(1) = 0$. These results give the following corollary and theorem.

Corollary 2.1.18. *Let $q = p^m \neq 16$. If a is a Kloosterman zero in $\mathbb{F}_q$, then a^{p^i} are distinct Kloosterman zeros in $\mathbb{F}_q$ for $1 \leq i \leq m-1$.*

Theorem 2.1.19. *Let $q = p^m \neq 16$. Let a be a Kloosterman zero in $\mathbb{F}_q$. The degree of the minimal polynomial $f(x)$ of a over $\mathbb{F}_p$ is m, and we can obtain $f(x)$ as*

$$f(x) = \prod_{i=0}^{m-1}(x - a^{p^i}).$$

2.1.1 Root Finding

For the case of finding all zeros in $\mathbb{F}_q$ of a given polynomial $f \in \mathbb{F}_q[x]$, we only need to consider linear factors of f over $\mathbb{F}_q$. Thus, it is enough to first compute $g = \gcd(x^q - x, f)$. We only apply the equal-degree factorization algorithm to g instead of the whole distinct-degree decomposition of f, see [12].

The following theorem shows the running time of finding root.

Theorem 2.1.20. *[12, Corollary 14.16] The cost of finding all roots in $\mathbb{F}_q$ for a polynomial of degree n with coefficients in $\mathbb{F}_q$ is $O(M(n)\log n \log(nq))$ or $O^{\sim}(n \log q)$ where $M(n)$ is the number of operations in the field needed for multiplying two polynomials of degree less than n over $\mathbb{F}_q$. (See Definition 8.26 in [12] for the meaning of $M(n)$.)*

2.2 Elliptic Curves and Modular Polynomials

We follow concepts and ideas from [23]. Throughout this section, we use standard definitions and results on elliptic curves over finite fields and on the Abelian groups associated with them. We recommend [1], [26] as accessible references for these topics.

An elliptic curve $\mathcal{E}$ over $\mathbb{F}_{p^m}$ is a non-singular curve given by an equation of the general form

$$\mathcal{E} : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.7}$$

where $a_i \in \mathbb{F}_{p^m}$. We denote by $\mathcal{E}(\mathbb{F}_{p^m})$ the set of points $(x, y) \in \mathbb{F}_{p^m}^2$ on the curve $\mathcal{E}$, together with one extra point which is called "the point at infinity," which we will denote by

$\mathcal{O}$. The set $\mathcal{E}(\mathbb{F}_{p^m})$ is called the set of $\mathbb{F}_{p^m}$*-rational points* on $\mathcal{E}$. It is well known that on the set $\mathcal{E}(\mathbb{F}_{p^m})$ we can define a binary commutative operation, usually called "point addition" and denoted $P + Q$ where $P, Q \in \mathcal{E}(\mathbb{F}_{p^m})$. Then it is well known that $(\mathcal{E}(\mathbb{F}_{p^m}), +)$ is an Abelian group associated with the ellitpic curve $\mathcal{E}$ over $\mathbb{F}_{p^m}$, with the identity element $\mathcal{O}$. Throughout this chapter we denote this group simply by $\mathcal{E}(\mathbb{F}_{p^m})$. The group law (definition of the $+$ operation) can be found in all books on this topic, for example in [1].

We denote by $\#\mathcal{E}(\mathbb{F}_{p^m})$ the order of the group $\mathcal{E}(\mathbb{F}_{p^m})$, that is, the number of $\mathbb{F}_{p^m}$-rational points on $\mathcal{E}$.

The *order of a point* P on an elliptic curve is the smallest r such that

$$rP = P + P + \cdots + P \text{ (r times)} = \mathcal{O}$$

(so that $sP \neq \mathcal{O}$ for $0 < s < r$).

For this thesis, we only consider characteristic 2. However, we include the terminology and results in characteristic 3 because Kloosterman sums in both characteristics have similar interesting properties, and bent functions can also be constructed from Kloosterman zeros in characteristic 3. The property of elliptic curves that is the most interesting to our applications is that the number of $\mathbb{F}_{p^m}$-rational points on a certain elliptic curve is related to the Kloosterman sum $\mathcal{K}_{p^m}(a)$ for $a \in \mathbb{F}_{p^m}^*$ and $p \in \{2, 3\}$. This is asserted in the next two theorems.

Theorem 2.2.1. [20] *Let $a \in \mathbb{F}_{2^m}^*$ and let $\mathcal{E}_2^a$ be the elliptic curve*

$$\mathcal{E}_2^a : \quad y^2 + xy = x^3 + a.$$

Then $\#\mathcal{E}_2^a(\mathbb{F}_{2^m}) = 2^m + \mathcal{K}_{2^m}(a)$.

Theorem 2.2.2. [27] *Let $a \in \mathbb{F}_{3^m}^*$ and let $\mathcal{E}_3^a$ be the elliptic curve*

$$\mathcal{E}_3^a : \quad y^2 = x^3 + x^2 - a.$$

Then $\#\mathcal{E}_3^a(\mathbb{F}_{3^m}) = 3^m + \mathcal{K}_{3^m}(a)$.

Example 2.2.3. *When the field is $\mathbb{F}_4 = \{0, 1, \beta, \beta^2\}$, we have $\#\mathcal{E}_2^a(\mathbb{F}_4) = 4 + \mathcal{K}_4(a)$. Note that β is a primitive element in $\mathbb{F}_4$. Table 2.1 shows $\mathcal{K}_4(a)$ and $\#\mathcal{E}_2^a(\mathbb{F}_4)$ for each a. For*

a	0	1	β	β^2
$\#\mathcal{E}_2^a(\mathbb{F}_4)$	–	8	4	4
$\mathcal{K}_4(a)$	0	4	0	0

Table 2.1: $\mathcal{K}_4(a)$ and $\#\mathcal{E}_2^a(\mathbb{F}_4)$ for each $a \in \mathbb{F}_4$

instance, when $a = \beta$:

$$\begin{aligned} \mathcal{E}_2^\beta &: y^2 + xy = x^3 + \beta \\ \mathcal{E}_2^\beta(\mathbb{F}_4) &= \{ (\beta, 1), (\beta, \beta^2), (0, \beta^2), \mathcal{O} \} \\ \mathcal{K}_4(\beta) &= 1 + (-1)^1 + (-1)^0 + (-1)^1 = 0 \end{aligned}$$

Notice that the direct computation of Kloosterman sum from Definition 2.1.12 is not practical for larger fields. Instead of that, we compute $\mathcal{K}_q(a)$ using Theorems 2.2.1 and 2.2.2 in combination with fast point counting algorithms [21] that count the number of rational points on elliptic curves over finite fields, such as variants of the *Schoof-Elkies-Atkin (SEA)* algorithm. The *SEA* algorithm [1] has complexity $O(\log^8 q) = O(m^8)$ assuming no fast multiplication routines are used and $O(\log^{5+\epsilon} q) = O(m^{5+\epsilon})$ with fast multiplication routines. Note that Magma contains an efficient implementation of the *SEA* algorithm for computing the number of points on an elliptic curve over a finite field [2].

The discriminant $\Delta(\mathcal{E})$ and j-invariant $j(\mathcal{E})$ for any elliptic curve $\mathcal{E}$ are defined, for example, at pages 30 and 31 in [1]. They are given by formulas in terms of the coefficients a_i of $\mathcal{E}$ defined in equation (2.7). $\Delta(\mathcal{E})$ is a polynomial in a_i and $j(\mathcal{E})$ is a rational function in a_i.

Lemma 2.2.4. *Let $\mathcal{E}_2^a$ and $\mathcal{E}_3^a$ be the curves defined in Theorems 2.2.1 and 2.2.2. We have*

$$\begin{aligned} \Delta(\mathcal{E}_2^a) &= a, \\ j(\mathcal{E}_2^a) &= a^{-1}, \\ \Delta(\mathcal{E}_3^a) &= a^3 \quad \textit{and} \\ j(\mathcal{E}_3^a) &= a^{-3}. \end{aligned}$$

Definition 2.2.5. *We define the* trace of Frobenius at q *of $\mathcal{E}$ to be the integer t where*

$$\#\mathcal{E}(\mathbb{F}_q) = q + 1 - t.$$

Thus, an equivalent characterization of a Kloosterman zero $a \in \mathbb{F}_q^*$ is that the corresponding elliptic curve has trace of Frobenius at q equal to $t = 1$, that is, $\#\mathcal{E}_p^a(\mathbb{F}_q) = q$ where $p \in \{2, 3\}$.

Isogeny is a relation between the groups of rational points on elliptic curves. A detailed description is beyond the scope of this thesis, however we will give precise references for the statements that we will use. We recommend to read section III.6 and III.8 of [1] for more detailed information. We will now list the statements that are important to us.

Lemma 2.2.6. *[1, Lemma III.12] Two isogenous elliptic curves over a finite field have the same number of rational points.*

Recall that Φ_n is the n-th classical modular polynomial, which was defined for the case of prime n in Section 1.2. These polynomials can be defined for all positive integers n. We did not pursue Φ_n for non-prime n, as the computation of it is then more complicated, and the prime values of n are sufficient for our applications in this chapter and in the next chapter.

Lemma 2.2.7. *[1, page 51] There is an isogeny of degree n from $\mathcal{E}'$ to $\mathcal{E}''$ if and only if*

$$\Phi_n(j(\mathcal{E}'), j(\mathcal{E}'')) = 0.$$

Notice that the modular polynomials $\Phi_l(x, y)$ are symmetric and have integer coefficients, thus it can be interpreted over any field. Recall $\bar{\Phi}_l(x, y) = \Phi_l(x, y) \bmod 2$.

Corollary 2.2.8. *[24] Let $q = 2^m$ and a be a Kloosterman zero in $\mathbb{F}_q$.*
Suppose $\bar{\Phi}_n(a^{-1}, b^{-1}) = 0$ for some positive integer n and some $b \in \mathbb{F}_q^*$. Then b is also a Kloosterman zero in $\mathbb{F}_q$.

Proof. By Theorem 2.2.1, $\mathcal{K}_q(c) = \#\mathcal{E}_2^c(\mathbb{F}_q) - q$. By assumption $\#\mathcal{E}_2^a(\mathbb{F}_q) = q$, and we need to show that $\#\mathcal{E}_2^b(\mathbb{F}_q) = \#\mathcal{E}_2^a(\mathbb{F}_q)$. By Lemma 2.2.4, $j(\mathcal{E}_2^a) = a^{-1}$ and $j(\mathcal{E}_2^b) = b^{-1}$. By the assumption $\bar{\Phi}_n(a^{-1}, b^{-1}) = 0$ and Lemma 2.2.7, the curves $\mathcal{E}_2^a$ and $\mathcal{E}_2^b$ are isogenous. We now apply Lemma 2.2.6 to finish the proof. □

Corollary 2.2.8 is a key ingredient for our algorithm that lists Kloosterman zeros. It tells us that, given an arbitrary Kloosterman zero a, we may produce more Kloosterman zeros in the same field by substituting $y = a^{-1}$ in $\Phi_n(x, y)$ and finding root(s) in $\mathbb{F}_q$ of

the resulting univariate polynomial in x. This is a novel idea by combining concepts from modular polynomials, elliptic curve groups and Kloosterman sums.

Now we are interested in the number of roots in $\mathbb{F}_q$ of this univariate polynomial. Happily, there is a formula for this number of roots:

Proposition 2.2.9. *[1, Proposition VII.2] Let $p \in \{2, 3\}$, $q = p^m$. Let ℓ be a prime. Let $\mathcal{E}$ be an elliptic curve over $\mathbb{F}_q$ such that $j(\mathcal{E}) \neq 0$ and the trace of Frobenius of $\mathcal{E}$ at q is t. Consider $\Phi_\ell(x, j(\mathcal{E})) \in \mathbb{F}_q[x]$. The number of roots of $\Phi_\ell(x, j(\mathcal{E}))$ in $\mathbb{F}_q$ is:*

$$\begin{array}{ll} 1 \text{ or } \ell + 1 & \text{if } \ell \text{ divides } t^2 - 4q, \\ 2 & \text{if } t^2 - 4q \text{ is a non-zero square modulo } \ell, \\ 0 & \text{if } t^2 - 4q \text{ is a non-square modulo } \ell. \end{array}$$

By the remark after Definition 2.2.5, we have $t = 1$ for the trace of Frobenius. Applying Proposition 2.2.9 with $t = 1$, we summarize the relation between the Legendre symbol $(\frac{1-4q}{l})$ and the number of root(s) of $\bar{\Phi}_l(x, z^{-1})$ in $\mathbb{F}_q$ as follows:

$$\left(\frac{1-4q}{l}\right) = \begin{cases} 0, & \text{there are 1 or } l+1 \text{ root(s) of } \bar{\Phi}_l(x, z^{-1}) \text{ in } \mathbb{F}_q \\ 1, & \text{there are 2 roots of } \bar{\Phi}_l(x, z^{-1}) \text{ in } \mathbb{F}_q \\ -1, & \text{there are no roots of } \bar{\Phi}_l(x, z^{-1}) \text{ in } \mathbb{F}_q. \end{cases}$$

Recall that the Legendre symbol $(\frac{u}{l})$ represents the quadratic character modulo a prime number l for $u \in \mathbb{Z}$. Notice that an integer u is called a quadratic residue modulo l if it is congruent to a perfect square modulo l, i.e., if there exists an integer v such that $v^2 \equiv u \pmod{l}$. Otherwise, u is called a quadratic non-residue modulo l.

The algorithm that lists all Kloosterman zeros in a given binary field needs the total number of them as a stopping condition. This number is given in Theorem 2.2.10 below. First we need some definitions. For an integer $\delta < 0$ such that $\delta \equiv 0, 1 \pmod{4}$, the *class number for discriminant* δ is an integer denoted by $h(\delta)$. This function is computed by the `ClassNumber` function in Magma [2]. We refer to [30] for a detailed presentation on the class number. By $H(\delta)$ we denote the Kronecker class number for δ, which can be computed as follows [30, Proposition 2.2]:

$$H(\delta) = \sum_d h\left(\frac{\delta}{d^2}\right)$$

where the summation range consists of those positive integers d for which $d^2|\delta$ and $\delta/d^2 \equiv 0, 1 \pmod 4$. Magma code for computing $H(\delta)$ is given in Appendix A.4.

Theorem 2.2.10. *[20, Proposition 9.1] Let $q = 2^m$. The number of Kloosterman zeros in $\mathbb{F}_q$ is $H(1-4q)$.*

The statements given in Sections 2.1 and 2.2 play a significant role in computing all Kloosterman zeros in a given field $\mathbb{F}_{2^m}$ (Algorithm 2.2), and computing all minimal polynomials over $\mathbb{F}_2$ of Kloosterman zeros in a given field $\mathbb{F}_{2^m}$ (Algorithm 2.3).

2.3 Computation of Binary Kloosterman Zeros

Throughout this section, let

$$q = 2^m \text{ where } m \neq 4$$

because of Corollary 2.1.18 and Theorem 2.1.19. Let a be an arbitrary Kloosterman zero in $\mathbb{F}_{2^m}$. For prime l let $\bar{\Phi}_l$ be the classical modular polynomial of level l over $\mathbb{F}_2$ as defined in Section 1.2. By Theorem 2.1.19 the minimal polynomial $f(x)$ of a Kloosterman zero a over $\mathbb{F}_2$ is

$$f(x) = \prod_{i=0}^{m-1} (x - a^{2^i}) \;\; m \neq 4.$$

2.3.1 Algorithm for One Kloosterman Zero

Our approach to listing all Kloosterman zeros in a given field is to obtain one intial Kloosterman zero in that field and then repeatedly apply Corollary 2.2.8 until all Kloosterman zeros are listed. Thus, first we need an algorithm that finds *one (arbitrary)* Kloosterman zero in a given field. In this section, we introduce such an algorithm. This method is sketched (without giving details) in [23].

We have mentioned that we can use the *SEA* algorithm to compute the value of Kloosterman sum. However, since in this thesis we are only interested in Kloosterman zeros, we are only interested in deciding whether $\mathcal{K}_{2^m}(a) = 0$ or $\mathcal{K}_{2^m}(a) \neq 0$. For this, we do not need to run the complicated *SEA* algorithm. In this section we provide a yes-biased Monte Carlo probabilistic algorithm that tests whether an element of $\mathbb{F}_{2^m}$ is a Kloosterman zero.

This algorithm was just sketched in [23] but we provide pseudocode and the complexity analysis. We show that this algorithm is faster than the *SEA* algorithm.

Recall that $\#\mathcal{E}(\mathbb{F}_q)$ denotes the number of $\mathbb{F}_q$-rational points on $\mathcal{E}$. By Theorem 2.2.1, for $a \in \mathbb{F}_{2^m}^*$, the elliptic curve $\mathcal{E}_2^a$ defined by $\mathcal{E}_2^a : y^2 + xy = x^3 + a$ satisfies $\#\mathcal{E}_2^a(\mathbb{F}_{2^m}) = 2^m + \mathcal{K}_{2^m}(a)$. Hence a is a Kloosterman zero if and only if $\#\mathcal{E}_2^a(\mathbb{F}_{2^m}) = 2^m$.

We start by giving statements needed in our algorithm.

Theorem 2.3.1. *Let G be a cyclic group of order 2^m written additively. Let P be a generator of G and let k be a positive integer. Then kP is a generator of G if and only if k is odd. Hence there are 2^{m-1} distinct generators in G.*

Proof. Let k be odd. Then $\gcd(k, 2^m) = 1$. By the extended Euclidean algorithm, there exist $s, t \in \mathbb{Z}$ such that

$$sk + t2^m = 1.$$

Then we have

$$P = (sk + t2^m)P = skP = s(kP).$$

This shows $P \in \langle kP \rangle$ and $\langle P \rangle \subseteq \langle kP \rangle$. Thus, kP is a generator of G. There are 2^{m-1} possible odd values for k, and $\langle kP \rangle \neq G$ if k is even. Therefore, there are 2^{m-1} generators in G. □

Theorem 2.3.2. *[1, page 36] For any elliptic curve $\mathcal{E}$, the group $\mathcal{E}(\mathbb{F}_{p^m})$ is isomorphic to $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2}$ with $n_1 | n_2$ and $n_1 | p^m - 1$ and $\#\mathcal{E}(\mathbb{F}_{p^m}) = n_1 n_2$.*

Corollary 2.3.3. *Assume $\mathcal{K}_{2^m}(a) = 0$ where $a \in \mathbb{F}_{2^m}^*$. Then $\mathcal{E}_2^a(\mathbb{F}_{2^m})$ is isomorphic to $\mathbb{Z}_{2^m}$ and $\mathcal{E}_2^a(\mathbb{F}_{2^m})$ has 2^{m-1} generators.*

Proof. Let a be a Kloosterman zero. Then $\#\mathcal{E}_2^a(\mathbb{F}_{2^m}) = 2^m$. Let n_1 and n_2 be as in Theorem 2.3.2. Since we have $n_1 n_2 = 2^m$ and $n_1 | 2^m - 1$ by Theorem 2.3.2, we have $n_1 = 1$ and $n_2 = 2^m$. Thus, $\mathcal{E}_2^a(\mathbb{F}_{2^m})$ is isomorphic to $\mathbb{Z}_{2^m}$. Since $\mathbb{Z}_{2^m}$ is a cyclic group with order 2^m, there are 2^{m-1} distinct generators in $\mathcal{E}_2^a(\mathbb{F}_{2^m})$ by Theorem 2.3.1. □

Theorem 2.3.4. *[1, Theorem III.3] (Hasse Interval) We have*

$$q + 1 - 2\sqrt{q} \leq \#\mathcal{E}(\mathbb{F}_q) \leq q + 1 + 2\sqrt{q}.$$

Theorem 2.3.5. *Let $m \geq 3$. If $\mathcal{E}(\mathbb{F}_{2^m})$ contains a point of order 2^m, then $\#\mathcal{E}(\mathbb{F}_{2^m}) = 2^m$.*

Proof. We have

$$2^m + 1 - 2\sqrt{2^m} \leq \#\mathcal{E}(\mathbb{F}_{2^m}) \leq 2^m + 1 + 2\sqrt{2^m}$$

by Theorem 2.3.4. Suppose there is an element of $\mathcal{E}(\mathbb{F}_{2^m})$ whose order is 2^m. By Lagrange's Theorem $2^m | \#\mathcal{E}(\mathbb{F}_{2^m})$. Hence we have $\#\mathcal{E}(\mathbb{F}_{2^m}) = s2^m$ for some positive integer s, thus

$$\begin{aligned} 2^m + 1 - 2\sqrt{2^m} \leq \quad & s2^m & \leq 2^m + 1 + 2\sqrt{2^m} \\ 1 - 2\sqrt{2^m} \leq \quad & (s-1)2^m & \leq 1 + 2\sqrt{2^m}. \end{aligned}$$

When $m \geq 3$, the only s satisfying the last two inequalities is $s = 1$. Thus, the order of the group is 2^m. □

As there is little known about characterizing Kloosterman zeros by means other than their very definition (see Chapter 4 for more details on this), for getting the first Kloosterman zero in the given field we choose $a \in \mathbb{F}_q^*$ randomly and subject it to the test $\mathcal{K}_q(a) = 0$. Our algorithm needs a point on the associated curve $\mathcal{E}_2^a$. As we choose random a, we may as well start from a random point $P = (x, y) \in \mathbb{F}_q^2$ and compute $a = x^3 + xy + y^2$ from the curve equation to avoid solving the equation.

Please consult the description of Algorithm 2.1 now. As mentioned earlier this algorithm was only sketched in [23] without details. We provide the pseudocode and the correctness of the Algorithm as well as the complexity analysis. Notice that in lines 3 and 8, the symbol $+$ denotes the group operation in $\mathcal{E}_2^a(\mathbb{F}_{p^m})$ (point addition operation).

We now show the correctness of Algorithm 2.1. If the algorithm returns $True$, then the order of P is 2^m, and $\#\mathcal{E}_2^a(\mathbb{F}_{2^m}) = 2^m$ for $m \geq 3$ by Theorem 2.3.5. Thus, $\mathcal{K}_{2^m}(a) = 0$ by Theorem 2.2.1. By a contrapositive, if $\mathcal{K}_{2^m}(a) \neq 0$, then the algorithm returns $False$.

Now let us look at what happens in the case when $\mathcal{K}_{2^m}(a) = 0$. With probability $\frac{1}{2}$ the point P is a generator for $\mathcal{E}_2^a(\mathbb{F}_{2^m})$ because there are 2^{m-1} generators by Corollary 2.3.3. The algorithm returns $True$ exactly if P is a generator for $\mathcal{E}_2^a(\mathbb{F}_{2^m})$.

We use Algorithm 2.1 to compute one Kloosterman zero in $\mathbb{F}_{2^m}$ because it is much faster than the *SEA* algorithm. (As was mentioned earlier, the *SEA* algorithm returns the number of $\mathbb{F}_{p^m}$-rational points on elliptic curve $\mathcal{E}$ over $\mathbb{F}_{p^m}$, which is 2^m if and only if a is a Kloosterman zero.) Notice that Algorithm 2.1 only uses the elliptic curve point addition

Algorithm 2.1 Testing a Kloosterman Zero in $\mathbb{F}_{2^m}$

Input: A random point $P = (x, y)$ where $x, y \in \mathbb{F}_{2^m}$ ($m \geq 3$) and an element $a \in \mathbb{F}_{2^m}$ such that $a = x^3 + xy + y^2$.

Output: If $\mathcal{K}_{2^m}(a) \neq 0$, return $False$.

If $\mathcal{K}_{2^m}(a) = 0$, return $True$ with probability $\frac{1}{2}$, and return $False$ with probability $\frac{1}{2}$.

$T \leftarrow P$
for $i = 1$ to $m - 1$ **do**
  $T \leftarrow T + T$
  if $T = \mathcal{O}$ **then**
    return $False$
  end if
end for
$T \leftarrow T + T$
if $T = \mathcal{O}$ **then**
  return $True$
else
  return $False$
end if

	Our probabilistic algorithm	The SEA algorithm
Output	$\mathcal{K}_q(a) = 0?\ (T/F)$	$\mathcal{K}_q(a)$
Type	probabilistic with error probability $\frac{1}{2}$	deterministic
Implementation	easy	complicated
Complexity	$O(m^3)$ $O(m^{2+\epsilon})$	$O(m^8)$ $O(m^{5+\epsilon})$

Table 2.2: Our probabilistic algorithm versus the SEA algorithm

operation which only involves a constant number of addition, multiplication and division operations in $\mathbb{F}_{2^m}$. (The formulas can be found, for example, in Section III.3.2 of [1].) Assuming classical arithmetic operations in $\mathbb{F}_{2^m}$, their time complexity is $O(m^2)$. As there are $O(m)$ iterations of the **for** loop in the algorithm, the overall complexity of Algorithm 2.1 is $O(m \cdot m^2 + m^2) = O(m^3)$ with classical methods and this can be improved to $O(m^2 \log m)$ when fast arithmetic operations are used. On the other hand, the complexity of the *SEA* algorithm is $O(m^8)$ with classical methods and $O(m^{5+\epsilon})$ with fast multiplication routines. This information is summarized in Table 2.2.

Theorem 2.3.6. *Suppose that a is a Kloosterman zero in $\mathbb{F}_{2^m}$. The expected number of runs of Algorithm 2.1 (with different inputs P) to prove that a is a Kloosterman zero is* 2.

Proof. The number of runs is k if and only if the algorithm outputs $False$ $k-1$ times and then it outputs $True$ in the k-th run. Thus the expected number of runs is

$$\sum_{k=1}^{\infty} k \left(\frac{1}{2}\right)^{k-1} \left(\frac{1}{2}\right) = 2.$$

□

Notice that for computing one Kloosterman zero using Algorithm 2.1 we only run once for each choice of the random point P. If a certain choice of P does not produce a Kloosterman zero, then we choose another point P which in general will correspond to a different field element a.

By Theorem 2.2.10, there are $H(1-4\cdot 2^m)$ Kloosterman zeros in the field $\mathbb{F}_{2^m}$. For the range $m \leq 63$, we can find using Magma that $0.51 \cdot 2^{m/2} < H(1-4\cdot 2^m) < 2.38 \cdot 2^{m/2}$.

Appendix A.2 is our Magma implementation of Algorithm 2.1. Using Magma with $m = 32$, it took 33.32 seconds to find one Kloosterman zero. Notice that for larger fields, we need to implement this algorithm in C++ using *NTL* library [31].

For each $m \leq 63$, we obtained one Kloosterman zero from Lisoněk using his C++ implementation of Algorithm 2.1 and this Kloosterman zero became the initial Kloosterman zero for our later algorithms.

2.3.2 Algorithm for All Kloosterman Zeros

Up to now, there has been no known efficient method to list all Kloosterman zeros in fields of very large order. Moreover, the known listings of them are in [5] and they are only up to $m \leq 14$ for $\mathbb{F}_{2^m}$. In this section, we develop novel algorithms that compute all Kloosterman zeroes and their minimal polynomials in $\mathbb{F}_q$. We have used these algorithms to compute lists of Kloosterman zeros and minimal polynomials over $\mathbb{F}_2$ of Kloosterman zeros for all fields $\mathbb{F}_{2^m}$ up to $m \leq 63$. This is significant improvement compared to the previous known results in [5]. Some statistics (including timings) for our computations are shown in Section 2.3.3.

In order to proceed to the algorithms, we require m where

$$q = 2^m \text{ for } m \geq 6,$$

one precomputed Kloosterman zero (which in our computations is found using Algorithm 2.1, which is applied to random field elements) and the total number of Kloosterman zeros in $\mathbb{F}_q$ as the inputs. We set $m \geq 6$ because small cases are easy to work out, and avoiding the small cases simplifies our algorithm.

Recall that $\bar{\Phi}_l(x, y)$ is the l-th classical modular polynomial over $\mathbb{F}_2$, which for prime l is defined in Section 1.4. By Lemma 2.2.4, the j-invariant of $\mathcal{E}_2^a$ is a^{-1} for all $a \in \mathbb{F}_q^*$. Listing all Kloosterman zeros can be done by listing the inverses of all Kloosterman zeros in $\mathbb{F}_q$.

Theorem 2.3.7. *If z is a Kloosterman zero in $\mathbb{F}_q$, then $b \in \mathbb{F}_q^*$ is also a Kloosterman zero if $\bar{\Phi}_l(b^{-1}, z^{-1}) = 0$ for some prime l.*

Proof. Proof immediately follows from Corollary 2.2.8. □

Consequently, our algorithms apply the root finding algorithm on evaluated modular polynomials $\bar{\Phi}_l(x, z^{-1})$ to obtain the inverses of other Kloosterman zeros. Notice that $\bar{\Phi}_l(x, z^{-1})$ is a univariate polynomial over $\mathbb{F}_q$ in x whose degree is $l + 1$.

We observed through experiments that most of $\bar{\Phi}_l(x, z^{-1})$ ended up having only one root when the case $(\frac{1-4q}{l}) = 0$ occurs as prime l varies. This gives options whether we restrict to the case $(\frac{1-4q}{l}) = 0$ or 1 or only the case $(\frac{1-4q}{l}) = 1$. We choose $(\frac{1-4q}{l}) = 0$ or 1 because searching Kloosterman zeros when we restrict to the case $(\frac{1-4q}{l}) = 1$ requires higher prime levels l for $\bar{\Phi}_l$ and it costs on evaluation and root finding of $\bar{\Phi}_l(x, z^{-1})$. However, searching Kloosterman zeros in $(\frac{1-4q}{l}) = 1$ could possibly reduce the number of root finding iteration steps because it ensures $\bar{\Phi}_l(x, z^{-1})$ has exactly 2 roots over $\mathbb{F}_q$. We compared the overall time and we found that the case $(\frac{1-4q}{l}) = 0$ or 1 is faster than the other case.

Recall that our algorithms require one Kloosterman zero a and the total number of Kloosterman zeros in $\mathbb{F}_{2^m}$, which is given by Theorem 2.2.10.

We suggest to read the pseudocode for Algorithms 2.2 and 2.3 at this time. These algorithms compute all Kloosterman zeros and all minimal polynomials of Kloosterman zeros for a certain field $\mathbb{F}_{2^m}$ for $m \leq 63$. The procedure is as follows: We let $invKLZ$ to be the container where the inverses of Kloosterman zeros are stored during the iterations of the while loop and $invKLZ$ contains a^{-1} when it is initialized. We let $\#invKLZ$ to be the number of the inverses of Kloosterman zeros stored in $invKLZ$. First, we set the prime level l to be the smallest prime such that $(\frac{1-4q}{l}) \neq -1$. Then, we find b^{-1} by solving $\bar{\Phi}_l(x, a^{-1}) = 0$ as a univariate polynomial in x by Theorem 2.3.7 and store b^{-1} to $invKLZ$. Keep finding the inverse(s) of Kloosterman zero(s) by finding root(s) of the univariate polynomial $\bar{\Phi}_l(x, z^{-1})$ where $z^{-1} \in invKLZ$. As soon as $\#invKLZ$ is equal to the total number of Kloosterman zeros, the algorithm terminates. If $\#invKLZ$ is less than the total number of Kloosterman zeros when we use all z^{-1} in $invKLZ$, then we set l to be the next prime number greater than l such that $(\frac{1-4q}{l}) \neq -1$.

On line 2 of Algorithms 2.2 and 2.3, $foundPoly$ is the number of minimal polynomials found through iterations and n is the number of minimal polynomials before we increase prime level l. We use n to distinguish the inverses of Kloosterman zeros into those on which we need to perform the root finding algorithm *only* for $\bar{\Phi}_l(x, jinv)$ and those on which we have to perform the root finding algorithm for $\bar{\Phi}_{ll}(x, jinv)$ for all prime $ll \leq l$.

Algorithm 2.2 Computing All Kloosterman Zeros in $\mathbb{F}_q$

Input: m where $q = 2^m, m \geq 6$, a Kloosterman zero a, $totalKLZ$ the total number of Kloosterman zeros over $\mathbb{F}_q$, and all modular polynomials $\bar{\Phi}_l$ upto some bound B computed by Algorithm 1.2.

Output: Listing of all Kloosterman zeros in $\mathbb{F}_q$.

1: $invKLZ[i] \leftarrow (a^{-1})^{2^i}, 0 \leq i \leq m-1$
2: $foundKLZ \leftarrow m$, $foundPoly \leftarrow 1$, $n \leftarrow 1$, $jinv \leftarrow a^{-1}$, $totalPoly \leftarrow \frac{totalKLZ}{m}$
3: Set H be a table whose size is the next prime number greater than $2 * totalPoly$
4: Choose the max bit-string among $(a^{-1})^{2^i}, 0 \leq i \leq m-1$ and insert it into H
5: Set l to be the smallest prime such that $(\frac{1-4q}{l}) \neq -1$
6: **while** $((\frac{1-4q}{l}) \neq -1)$ and $(foundPoly < totalPoly)$ **do**
7: $k \leftarrow 0$
8: **while** $(k < foundPoly)$ **do**
9: **if** $(k < n)$ **then**
10: $ll \leftarrow l$
11: **else**
12: Set ll be the smallest prime such that $(\frac{1-4q}{ll}) \neq -1$
13: **end if**
14: **while** $ll \leq l$ **do**
15: Compute b^{-1} from the root finding algorithm applied to $\bar{\Phi}_{ll}(x, jinv)$ as a univariate polynomial in x
16: **if** the max bit-string among $(b^{-1})^{2^i}, 0 \leq i \leq m-1$ is not in H **then**
17: $invKLZ[foundKLZ + i] \leftarrow (b^{-1})^{2^i}, 0 \leq i \leq m-1$
18: $foundPoly \leftarrow foundPoly + 1$, $foundKLZ \leftarrow foundKLZ + m$
19: **end if**
20: Set ll to be the next prime number greater than ll such that $(\frac{1-4q}{ll}) \neq -1$
21: **end while**
22: $k \leftarrow k + m$, $jinv = invKLZ[k]$
23: **end while**
24: $n \leftarrow foundPoly$, l to be the next prime number greater than l such that $(\frac{1-4q}{l}) \neq -1$
25: **end while**
26: **return** $(invKLZ[index])^{-1}, 0 \leq index < totalKLZ$

Using the algorithms we have found all Kloosterman zeros in $\mathbb{F}_{2^m}$ up to $m \leq 57$ on a Pentium(R) D 3.00GHz with 2GB memory. We are strongly positive that with a larger memory machine, we can go to higher m. In fact, we have computed all Kloosterman zeros for $m \leq 63$ and some of Kloosterman zeros up to $m \leq 70$ on Intel Core i7 CPU at 2.6 GHz with 12 GB RAM. The polynomials $\bar{\Phi}_l$ with prime levels $l \leq 61$ were sufficient for all $m \leq 63$.

Theorem 2.3.8. *To list all Kloosterman zeros in $\mathbb{F}_{2^m}$ using Algorithm 2.2, the polynomials $\bar{\Phi}_l$ with prime levels $l \leq 61$ are sufficient for all $m \leq 63$.*

Algorithms 2.2 and 2.3 are implemented in C++ with *NTL* library [31]. Algorithm 2.2 computes all Kloosterman zeros in $\mathbb{F}_q$ and Algorithm 2.3 lists all minimal polynomials of Kloosterman zeros in $\mathbb{F}_q$. Both of them compare a newly computed minimal polynomial of a Kloosterman zero to the previously found minimal polynomials (stored in memory) in order to determine whether the new polynomial needs to be stored or not.

Before developing this version, we first stored Kloosterman zeros in the memory. However, the total number of Kloosterman zeros becomes larger as we increase m, which causes the memory shortage problem. Moreover, the number of minimal polynomials is exactly the number of Kloosterman zeros divided by m since each polynomial represents m distinct Kloosterman zeros by Corollary 2.1.18 and Theorem 2.1.19. For this reason, we decided to store minimal polynomials of Kloosterman zeros to utilize memory more efficiently.

We use the balanced binary tree, provided by standard C library *search*, to store the minimal polynomials in memory to achieve better performance. The data type of the node of the tree is either 64-bit integer (or 128-bit integer if $m > 64$) as a bit-string of coefficients. However we found that using binary tree consumed too much memory space. Thus to make the algorithm more efficient in terms of space, we choose to use the hash table. Using the hash table we consumed only half of memory space compared to the binary tree.

Then we realize that instead of minimal polynomials, we can simply compare the inverses of Kloosterman zeros. First, we compute all inverses of conjugates of a new Kloosterman zero, which is obtained by line 15. Then, choose the one whose bit-string value is the largest and store it in the hash table. Using this method, each minimal polynomial is computed only once when it is output. Moreover, the minimal polynomial over $\mathbb{F}_2$ of the inverse of a Kloosterman zero is the reciprocal polynomial of this Kloosterman zero over

$\mathbb{F}_2$. This implies we simply output the minimal polynomial of z over $\mathbb{F}_2$ by the reciprocal polynomials of z over $\mathbb{F}_2$. Because using this method saves the multiplication and inverse operations, we have improved the overall running time about $25 - 35\%$ from the version that stores minimal polynomials on the binary tree.

Algorithm 2.3 Computing All Minimal Polynomials of Kloosterman Zeros in $\mathbb{F}_q$

Input: m where $q = 2^m, m \geq 6$, a Kloosterman zero a,
$totalKLZ$ the total number of Kloosterman zeros over $\mathbb{F}_q$, and
all modular polynomials $\bar{\Phi}_l$ upto some bound B computed by Algorithm 1.2.

Output: List of all minimal polynomials of Kloosterman zeros in $\mathbb{F}_q$.

1: $invKLZ[0] \leftarrow a^{-1}$, $recipPoly[foundPoly] \leftarrow \prod_{i=0}^{m-1}(x - (a^{-1})^{2^i})$
2: $foundKLZ \leftarrow 1$, $foundPoly \leftarrow 1$, $n \leftarrow 1$, $jinv \leftarrow a^{-1}$, $totalPoly \leftarrow \frac{totalKLZ}{m}$
3: Run line 3 through line 26 of Algorithm 2.2 with modification of lines 17, 18, 22, 26.
Line 17: $invKLZ[foundPoly] \leftarrow b^{-1}$, $recipPoly[foundPoly] \leftarrow \prod_{i=0}^{m-1}(x-(b^{-1})^{2^i})$
Line 18: $foundKLZ \leftarrow foundKLZ + 1$
Line 22: $k \leftarrow k + 1$
Line 26: **return** $recipPoly[index], 0 \leq index < totalPoly$

Note that the termination of the algorithms is not obvious from the pseudocode. This requires a deeper work in the theory that is beyond the scope of our thesis. Bounds on the largest prime l that is needed for a given field $\mathbb{F}_{2^m}$ require assumptions such as the Generalized Riemann Hypothesis.

More positively, we note that our computations (see Section 2.3.3) already cover most fields $\mathbb{F}_{2^m}$ for which our algorithms can run in the memory of today's computers ($m \leq 63$), and we give the precise values of l needed for those fields in the tables below. Interestingly, we only need primes $l \leq 53$ when $m \leq 63$ and $m \neq 49$ from Table 2.4 and 2.6 for $30 \leq m \leq 63$. The modular polynomials $\bar{\Phi}_l$ up to prime level $l \leq 59$ are available in Magma [2]. Hence anyone who wishes to repeat these computations does not need to compute the modular polynomials.

Appendix A.3 is our C++ implementation of Algorithm 2.3. This implementation returns a set S where $S[i] = (KLZ[i], recipPoly[i])$ for $0 \leq i < totalPoly$, where KLZ and $recipPoly$ are introduced and computed through Algorithm 2.3.

m	Running Time in Seconds	m	Running Time in Seconds	m	Running Time in Seconds	m	Running Time in Seconds
30	1	31	4	32	5	33	7
34	5	35	20	36	15	37	128
38	24	39	38	40	17	41	737
42	2	43	340	44	871	45	173
46	738	47	211	48	199	49	12457
50	6933	51	7	52	4353	53	16172
54	7748	55	34224				

Table 2.3: Running time (in seconds) on Intel Core i7 CPU at 2.6 GHz to obtain one Kloosterman zero in $\mathbb{F}_{2^m}$ using Algorithm 2.1

2.3.3 Timings and Results

In Table 2.3 we show the running time to obtain one Kloosterman zero in $\mathbb{F}_{2^m}$ using Algorithm 2.1 for $30 \leq m \leq 55$ on Intel Core i7 CPU at 2.6 GHz. These timings are provided by Lisoněk.

In Tables 2.4, 2.5 and 2.6 we show results of Algorithm 2.3. Tables 2.4 and 2.5 show the running times and results for Algorithm 2.3 for $30 \leq m \leq 57$ on a Pentium(R) D 3.00GHz with 2GB memory running C++ implementation that store the inverse of Kloosterman zeros on the hash table. Table 2.6 shows those for $58 \leq m \leq 63$ on an Intel Core i7 CPU at 2.6 GHz with 12 GB RAM running C++ implementation that store the minimal polynomials of Kloosterman zeros on the binary tree. Thus the overall running time for listing Kloosterman zeros for $58 \leq m \leq 63$ will be faster $25 - 35\%$ than those given in Table 2.6.

In Tables 2.4 and 2.5, 2.6, *Roots per second* represents how many Kloosterman zeros are found in one second by Algorithm 2.3 on average, which are computed as follows:

$$Roots\ per\ second = \frac{Total\ number\ of\ roots\ that\ are\ found}{Running\ Time}. \tag{2.8}$$

Moreover, *Repetition factor* represents how many times Algorithm 2.3 discovers a minimal polynomial on average. We have

$$Repetition\ factor = \frac{Total\ number\ of\ roots\ that\ are\ found}{Total\ number\ of\ minimal\ polynomials}. \tag{2.9}$$

m	Number of Kloosterman Zeros	Number of Minimal Polynomials	Primes l used	Running Time in Seconds	Roots per Second	Repetition Factor
30	42240	1408	{3, 5, 7, 23, 41, 47}	12	704	6.00
31	55056	1776	{5, 7, 11}	3	2047	3.46
32	63424	1982	{3, 13}	4	1483	2.99
33	57024	1728	{7, 31, 53}	20	292	3.39
34	243712	7168	{3, 5, 7, 11, 13, 17}	37	1103	5.70
35	213780	6208	{5, 11}	3	2715	1.33
36	354888	9858	{3, 7}	19	1555	3.00
37	278832	7536	{7, 17, 23, 31}	42	564	3.15
38	687040	18080	{3, 5, 11, 17, 19}	42	1399	3.25
39	951600	24400	{5, 7}	20	2928	2.40
40	1202400	30060	{3, 7, 23, 31}	97	1009	3.26
41	1179816	28776	{11, 13, 19, 23}	115	909	3.64
42	3384192	80576	{3, 5, 7, 19, 23}	355	1021	4.50
43	3558336	82752	{5, 7, 13, 31, 37, 41}	1128	439	5.99
44	3532496	80284	{3, 11}	122	1974	3.00
45	6751620	150036	{7, 17}	488	1269	4.13
46	19942656	433536	{3, 5, 7, 11, 17, 19}	1775	1190	4.87
47	12773754	271782	{5, 19}	640	1132	2.67
48	19184640	399680	{3, 7, 11, 13, 23}	1388	1116	3.88
49	12211584	249216	{7, 29, 47, 61}	4802	181	3.49
50	30566400	611328	{3, 5, 17, 31, 43}	8027	374	4.92
51	63977664	1254464	{5, 7, 11, 47}	22115	382	6.75
52	82353440	1583720	{3, 7, 19, 23}	11415	816	5.88

Table 2.4: Running time (in seconds) on Pentium(R) D 3.00GHz for Algorithm 2.3

m	Number of Kloosterman Zeros	Number of Minimal Polynomials	Primes l used	Running Time in Seconds	Roots per Second	Repetition Factor
53	70065152	1321984	{13, 17, 23, 37, 41}	20083	326	4.96
54	195810048	3626112	{3, 5, 7, 11, 17, 19}	45500	522	6.55
55	251110200	4565640	{5, 7, 11, 17}	21418	1151	5.40
56	274240512	4897152	{3, 11, 13, 19}	15797	966	3.12
57	330426378	5796954	{7, 23}	18101	640	2.00

Table 2.5: Running time (in seconds) on Pentium(R) D 3.00GHz for Algorithm 2.3

m	Number of Kloosterman Zeros	Number of Minimal Polynomials	Primes used	Running Time in Seconds	Roots per Second	Repetition Factor
58	957324800	16505600	{ 3, 5, 7, 11, 13, 17, 37, 41 }	314189	331	6.30
59	740609005	12552695	{ 5 }	31853	789	2.00
60	1336863480	22281058	{ 3, 7 }	88324	757	3.00
61	1147564208	18812528	{ 7, 11, 17, 47 }	242283	307	3.95
62	2020442112	32587776	{ 3, 5, 17, 23, 29, 37 }	781372	308	7.36
63	3685539312	58500624	{ 5, 7, 17 }	309789	539	2.85

Table 2.6: Running time (in seconds) on Intel Core i7 CPU at 2.6 GHz for Algorithm 2.3

Example 2.3.9. *From Table 2.4, for $m = 30$ it takes 12 seconds to discover 42240 Kloosterman zeros and 1408 minimal polynomials and only $l = 3, 5, 7, 23, 41, 47$ are used for $\bar{\Phi}_l$ finding new minimal polynomials. In one second, 704 roots are found and we found new Kloosterman zero in every 6 roots on average.*

By Theorem 2.2.1 we can use the *SEA* algorithm to check whether field elements computed by Algorithm 2.2 are Kloosterman zeros independently. The SEA algorithm is easily accessible in Magma [2] and we performed these checks for all $m \leq 40$.

2.3.4 Optimization Suggestions

We have improved our algorithm by first storing the minimal polynomials of Kloosterman zeros instead of Kloosterman zeros; then we store the inverse of Kloosterman zeros to speed up. Moreover we substitute the hash table for the binary tree to store the inverse of Kloosterman zeros to be more efficient in terms of memory usage.

We figured out that the bottleneck of our algorithm is the root finding. At least 83% of total running time is used in the root finding procedure. If we find an improvement root finding method, we believe that we can fasten overall computation.

Also we have plotted the running time with respect to the percentage of Kloosterman zeros computed by the algorithm. Figures 2.1, 2.2, and 2.3 show these graphs for $m = 50, 51$ and 52 respectively. We can improve the running time if we try to reduce idle time as we increase prime level earlier. For example, for $m = 50$ and 51 there are huge idle times when 50% of Kloosterman zeros are found. Notice that for $m = 52$ it appears that the algorithm is more stable (having not much idle time) than $m = 50$ or 51. We suggest that if we reduce the idle time, then the graphs for $m = 50$ and 51 will be closer to linear.

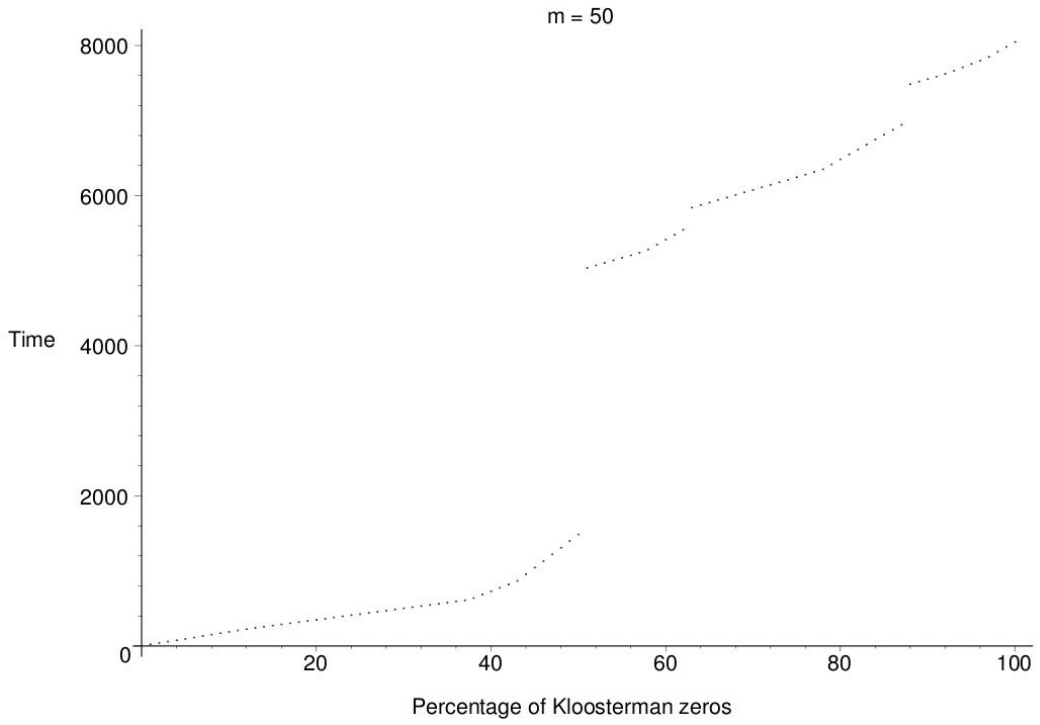

Figure 2.1: The running time respect to the percentage of Kloosterman zeros for $m = 50$

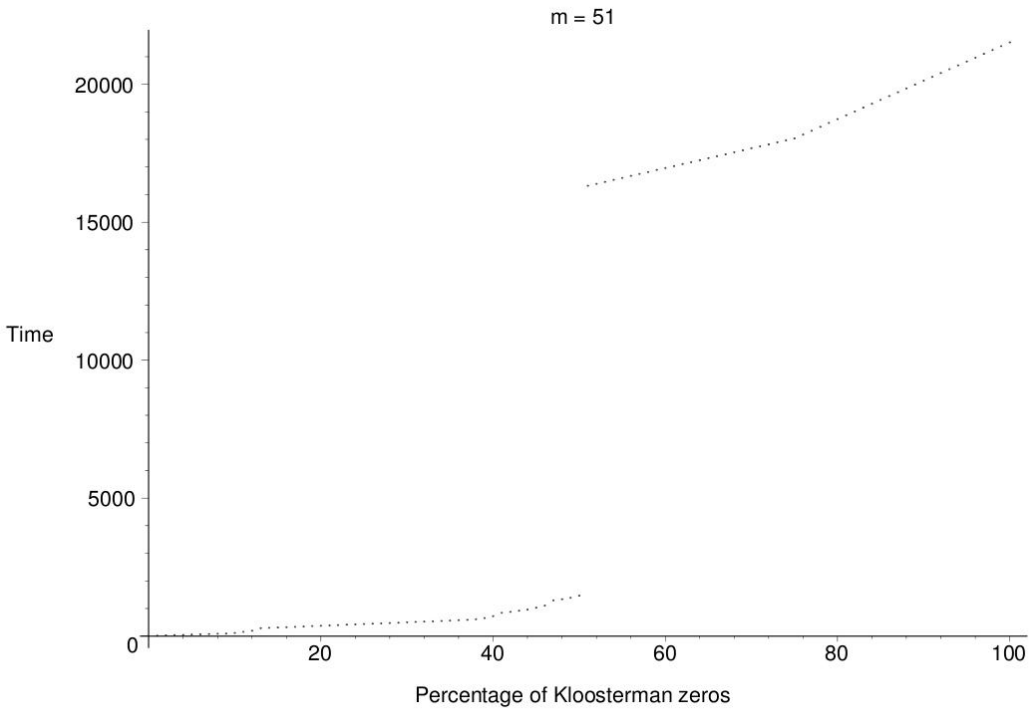

Figure 2.2: The running time respect to the percentage of Kloosterman zeros for $m = 51$

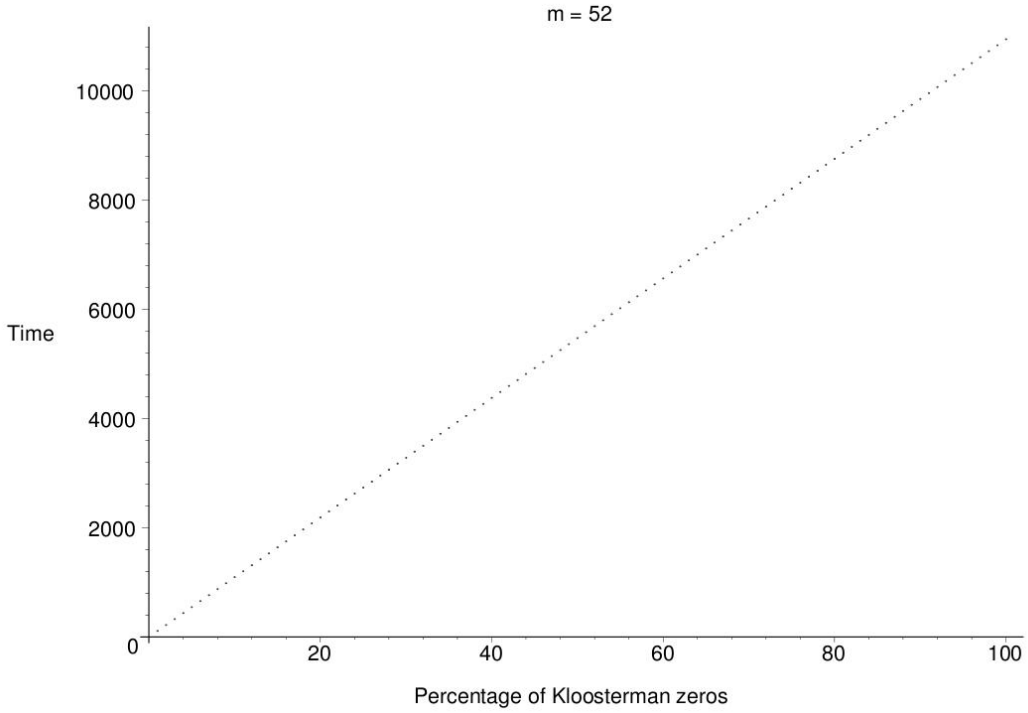

Figure 2.3: The running time respect to the percentage of Kloosterman zeros for $m = 52$

Chapter 3

Coefficients of Characteristic Polynomials of Kloosterman Zeros

Characterization of Kloosterman zeros appears to be very hard. Hence, necessary conditions for them are of interest. Various divisibility conditions for a Kloosterman sum $\mathcal{K}_{p^m}(a) \in \mathbb{Z}$ where $p \in \{2, 3\}$ by small integers can be investigated. The first condition appeared in a paper by van der Geer and van der Vlugt [14]. More such conditions appeared in papers by Helleseth and Zinoviev [18], Charpin, Helleseth, and Zinoviev [7], Lisoněk [23], and Garaschuk and Lisoněk [10], [11]. Then Göloğlu, Lisoněk, McGuire and Moloney [16] introduced divisibility conditions using the coefficients of the minimal polynomials over $\mathbb{F}_2$ of Kloosterman zeros. We conjecture two new relations for coefficients of the minimal polynomials of Kloosterman zeros over $\mathbb{F}_2$ later this chapter.

Since we obtained the minimal polynomials of all Kloosterman zeros in $\mathbb{F}_{2^m}$ for $m \leq 63$ using Algorithm 2.3, we started by verifying the known results from [18], [14] and [16]. We provide two algorithms, one of which finds relations among coefficients and the other algorithm checks whether the relation we have found is implied by previous results or not. We have found two new results which are independent of the known ones and of each other.

Throughout this chapter, let $q = 2^m$. Recall Definition 2.1.12, $\mathcal{K}_q(a)$ denotes the binary Kloosterman sum on $\mathbb{F}_q$:

$$\mathcal{K}_q(a) = 1 + \sum_{x \in \mathbb{F}_q^*} (-1)^{\mathrm{Tr}(x^{-1} + ax)}.$$

In some references the same mapping is defined by $\mathcal{K}_q(a) = \sum_{x\in\mathbb{F}_q}(-1)^{\mathrm{Tr}(x^{-1}+ax)}$ with $Tr(0^{-1}) = 0$.

3.1 Previous Results

Recall that by Corollary 2.1.8, the minimal polynomial $f(x)$ over $\mathbb{F}_2$ of $a \in \mathbb{F}_q$ is

$$f(x) = \prod_{i=0}^{n-1}(x - a^{2^i})$$

where n is the smallest positive integer such that $a^{2^n} = a$.

Definition 3.1.1. *Let $q = 2^m$ and $a \in \mathbb{F}_q$. We define the* characteristic polynomial $g(x) \in \mathbb{F}_p[x]$ *of a over $\mathbb{F}_2$ to be*

$$g(x) = \prod_{i=0}^{m-1}(x - a^{2^i}) = x^m + e_1x^{m-1} + e_2x^{m-2} + \cdots + e_m. \tag{3.1}$$

We set $e_i = 0$ for $i > m$.

We call e_1 the trace of a and e_2 the subtrace (or quadratic trace) of a.

Since $a^{2^m} = a$ for all $a \in \mathbb{F}_{2^m}$, n divides m where n is the smallest positive integer such that $a^{2^n} = a$. Then we have the following corollary.

Corollary 3.1.2. *Let $q = 2^m$ and $a \in \mathbb{F}_q$. Let n be the smallest positive integer such that $a^{2^n} = a$. Then the relation between the minimal polynomial $f(x)$ of a over $\mathbb{F}_2$ and the characteristic polynomial $g(x)$ of a over $\mathbb{F}_2$ is $g(x) = f(x)^d$ where $d = \frac{m}{n}$.*

Hence $e_i \in \mathbb{F}_2$ for all i in Definition 3.1.1, which justifies the part "over $\mathbb{F}_2$" in that definition.

Corollary 3.1.3. *Let $q = 2^m$ and $a \in \mathbb{F}_q$. Let n be the smallest positive integer such that $a^{2^n} = a$. The characteristic polynomial of a over $\mathbb{F}_2$ is equal to the minimal polynomial of a over $\mathbb{F}_2$ if and only if $n = m$.*

Recall that no Kloosterman zero belongs to any subfield of $\mathbb{F}_{2^m}$ except for $m = 4$ [16]. Thus, the characteristic polynomial over $\mathbb{F}_2$ of any Kloosterman zero a in $\mathbb{F}_{2^m}$ is the minimal polynomial of a over $\mathbb{F}_2$ except for $m = 4$ by Corollary 3.1.3.

Definition 3.1.4. *Let $a \in \mathbb{F}_q$. We define the* $\mathbb{Z}$-characteristic polynomial *of a over $\mathbb{Z}$ to be*

$$\prod_{i=0}^{m-1}(x - a^{2^i}) = x^m + \bar{e}_1 x^{m-1} + \bar{e}_2 x^{m-2} + \cdots + \bar{e}_m \tag{3.2}$$

where $\bar{e}_i \in \{0, 1\}$ considered as integers.

We restrict the integers $\bar{e}_i$ to the set $\{0, 1\}$ is so that we can identify $\bar{e}_i^2$ with $\bar{e}_i$. This restriction allows us to eliminate exponents and reduce the length of certain expressions in $\bar{e}_i$ that will follow later in this chapter.

Example 3.1.5. *Let $a = x + x^8 \in \mathbb{F}_{2^{10}}$ and $\mathcal{K}_{2^{10}}(a) = 0$. We have*

$$\prod_{i=0}^{9}(x - a^{2^i}) = x^{10} + x^7 + 1.$$

Thus $\bar{e}_3 \equiv \bar{e}_{10} \equiv 1 \pmod 2$ and $\bar{e}_i \equiv 0 \pmod 2$ for all i and $i \notin \{3, 10\}$.

In this section we rephrase the known results. Let $a \in \mathbb{F}_q$. These results are evaluation of $\mathcal{K}_q(a)$ mod 8, mod 16, mod 32, mod 64 and mod 128 and in terms of $\bar{e}_i$, the coefficients of the $\mathbb{Z}$-characteristic polynomials of a (see [18], [14], [16]). There are also known divisibility conditions of the values $\mathcal{K}_q(a)$ by 3 and 24 (see [10], [7]). All these conditions are necessary but not sufficient for $\mathcal{K}_q(a) = 0$. Checking these conditions is faster than computing the value of $\mathcal{K}_q(a)$.

The first result is a congruence for Kloosterman sums mod 8. A congruence for Kloosterman sums mod 8 is usually attributed to Helleseth and Zinoviev [18]. It also can be derived from an earlier paper by van der Geer and van der Vlugt [14].

Theorem 3.1.6. *[18], [14] Let $q = 2^m$, $q \geq 8$ and $a \in \mathbb{F}_q$. Let $\bar{e}_1, \ldots, \bar{e}_m \in \{0, 1\}$ be the coefficients of the $\mathbb{Z}$-characteristic polynomial of a as given in equation (3.2). Then we have*

$$\mathcal{K}_q(a) \equiv 4\bar{e}_1 \pmod 8.$$

Corollary 3.1.7. *[16] Let $q = 2^m$, $q \geq 8$ and $a \in \mathbb{F}_q$. Let $e_1, \ldots, e_m \in \mathbb{F}_2$ be the coefficients of the characteristic polynomial of a over $\mathbb{F}_2$ as given in equation (3.1). Then we have $\mathcal{K}_q(a) \equiv 0 \pmod 8$ if and only if*

$$e_1 = 0.$$

We have a congruence for Kloosterman sums modulo 16.

Theorem 3.1.8. *[17] Let $q = 2^m$, $q \geq 16$ and $a \in \mathbb{F}_q$. Let $\bar{e}_1, \ldots, \bar{e}_m \in \{0, 1\}$ be the coefficients of the $\mathbb{Z}$-characteristic polynomial of a as given in equation (3.2). Then we have*

$$\mathcal{K}_q(a) \equiv 12\bar{e}_1 + 8\bar{e}_2 \pmod{16}.$$

Corollary 3.1.9. *[16] Let $q = 2^m$, $q \geq 16$ and $a \in \mathbb{F}_q$. Let $e_1, \ldots, e_m \in \mathbb{F}_2$ be the coefficients of the characteristic polynomial of a over $\mathbb{F}_2$ as given in equation (3.1). Then $\mathcal{K}_q(a) \equiv 0 \pmod{16}$ if and only if*

$$e_1 = 0 \text{ and } e_2 = 0.$$

We can see divisibility conditions for Kloosterman sums by 24 in [7].

All following results are proved by Göloğlu, Lisoněk, McGuire and Moloney in [16]. Now, we present a congruence for Kloosterman sums modulo 32.

Theorem 3.1.10. *[16] Let $q = 2^m$, $q \geq 32$ and $a \in \mathbb{F}_q$. Let $\bar{e}_1, \ldots, \bar{e}_m \in \{0, 1\}$ be the coefficients of the $\mathbb{Z}$-characteristic polynomial of a as given in equation (3.2). Then we have*

$$\mathcal{K}_q(a) \equiv 28\bar{e}_1 + 8\bar{e}_2 + 16(\bar{e}_1\bar{e}_2 + \bar{e}_1\bar{e}_3 + \bar{e}_4) \pmod{32}.$$

Corollary 3.1.11. *[16] Let $q = 2^m$, $q \geq 32$ and $a \in \mathbb{F}_q$. Let $e_1, \ldots, e_m \in \mathbb{F}_2$ be the coefficients of the characteristic polynomial of a over $\mathbb{F}_2$ as given in equation (3.1). Then $\mathcal{K}_q(a) \equiv 0 \pmod{32}$ if and only if*

$$e_1 = 0,\ e_2 = 0 \text{ and } e_4 = 0.$$

Next, we present a congruence for Kloosterman sums modulo 64.

Theorem 3.1.12. *[16] Let $q = 2^m$, $q \geq 64$ and $a \in \mathbb{F}_q$. Let $\bar{e}_1, \ldots, \bar{e}_m \in \{0, 1\}$ be the coefficients of the $\mathbb{Z}$-characteristic polynomial of a as given in equation (3.2). Then we have*

$$\begin{aligned}\mathcal{K}_q(a) \equiv\ & 28\bar{e}_1 + 40\bar{e}_2 + 16(\bar{e}_1\bar{e}_2 + \bar{e}_1\bar{e}_3 + \bar{e}_4) \\ & + 32(\bar{e}_1\bar{e}_4 + \bar{e}_1\bar{e}_5 + \bar{e}_1\bar{e}_6 + \bar{e}_1\bar{e}_7 + \bar{e}_2\bar{e}_3 + \bar{e}_2\bar{e}_4 + \bar{e}_2\bar{e}_6 \\ & + \bar{e}_3\bar{e}_5 + \bar{e}_1\bar{e}_2\bar{e}_3 + \bar{e}_1\bar{e}_2\bar{e}_4 + \bar{e}_8) \pmod{64}.\end{aligned}$$

Corollary 3.1.13. *[16] Let $q = 2^m$, $q \geq 64$ and $a \in \mathbb{F}_q$. Let $e_1, \ldots, e_m \in \mathbb{F}_2$ be the coefficients of the characteristic polynomial of a over $\mathbb{F}_2$ as given in equation (3.1). Then $\mathcal{K}_q(a) \equiv 0 \pmod{64}$ if and only if the conditions of Corollary 3.1.11 are satisfied, and furthermore,*

$$e_8 = e_3 e_5.$$

We have a congruence for Kloosterman sums modulo 128.

Theorem 3.1.14. *[16] Let $q = 2^m$, $q \geq 128$ and $a \in \mathbb{F}_q$. Let $\bar{e}_1, \ldots, \bar{e}_m \in \{0, 1\}$ be the coefficients of the $\mathbb{Z}$-characteristic polynomial of a as given in equation (3.2). Then we have*

$$\begin{aligned}
\mathcal{K}_q(a) \equiv\ & 92\bar{e}_1 + 40\bar{e}_2 + 16(\bar{e}_1\bar{e}_2 + \bar{e}_4) + 80\bar{e}_1\bar{e}_3 + 32(\bar{e}_1\bar{e}_2\bar{e}_3 + \bar{e}_1\bar{e}_7 + \bar{e}_2\bar{e}_6 + \bar{e}_8) \\
& + 96(\bar{e}_1\bar{e}_2\bar{e}_4 + \bar{e}_1\bar{e}_4 + \bar{e}_1\bar{e}_5 + \bar{e}_1\bar{e}_6 + \bar{e}_2\bar{e}_3 + \bar{e}_2\bar{e}_4 + \bar{e}_3\bar{e}_5) \\
& + 64(\bar{e}_1\bar{e}_2\bar{e}_3\bar{e}_4 + \bar{e}_1\bar{e}_2\bar{e}_3\bar{e}_5 + \bar{e}_1\bar{e}_2\bar{e}_5 + \bar{e}_1\bar{e}_2\bar{e}_6 + \bar{e}_1\bar{e}_2\bar{e}_{10} + \bar{e}_1\bar{e}_2\bar{e}_{11} + \bar{e}_1\bar{e}_2\bar{e}_{12} \\
& + \bar{e}_1\bar{e}_3\bar{e}_7 + \bar{e}_1\bar{e}_3\bar{e}_{11} + \bar{e}_1\bar{e}_4\bar{e}_6 + \bar{e}_1\bar{e}_4\bar{e}_7 + \bar{e}_1\bar{e}_4\bar{e}_8 + \bar{e}_1\bar{e}_4\bar{e}_{10} + \bar{e}_1\bar{e}_5\bar{e}_7 + \bar{e}_1\bar{e}_5\bar{e}_9 \\
& + \bar{e}_1\bar{e}_6\bar{e}_8 + \bar{e}_1\bar{e}_8 + \bar{e}_1\bar{e}_9 + \bar{e}_1\bar{e}_{10} + \bar{e}_1\bar{e}_{11} + \bar{e}_1\bar{e}_{12} + \bar{e}_1\bar{e}_{13} + \bar{e}_1\bar{e}_{14} + \bar{e}_1\bar{e}_{15} + \bar{e}_2\bar{e}_3\bar{e}_5 \\
& + \bar{e}_2\bar{e}_3\bar{e}_8 + \bar{e}_2\bar{e}_3\bar{e}_9 + \bar{e}_2\bar{e}_4\bar{e}_5 + \bar{e}_2\bar{e}_4\bar{e}_6 + \bar{e}_2\bar{e}_4\bar{e}_8 + \bar{e}_2\bar{e}_5\bar{e}_7 + \bar{e}_2\bar{e}_7 + \bar{e}_2\bar{e}_8 + \bar{e}_2\bar{e}_{10} \\
& + \bar{e}_2\bar{e}_{12} + \bar{e}_2\bar{e}_{14} + \bar{e}_3\bar{e}_4\bar{e}_5 + \bar{e}_3\bar{e}_4\bar{e}_6 + \bar{e}_3\bar{e}_4 + \bar{e}_3\bar{e}_7 + \bar{e}_3\bar{e}_{10} + \bar{e}_3\bar{e}_{13} + \bar{e}_3 + \bar{e}_4\bar{e}_6 \\
& + \bar{e}_4\bar{e}_8 + \bar{e}_4\bar{e}_{12} + \bar{e}_5\bar{e}_6 + \bar{e}_5\bar{e}_{11} + \bar{e}_6\bar{e}_{10} + \bar{e}_7\bar{e}_9 + \bar{e}_{16}) \pmod{128}.
\end{aligned}$$

Corollary 3.1.15. *[16] Let $q = 2^m$, $q \geq 128$ and $a \in \mathbb{F}_q$. Let $e_1, \ldots, e_m \in \mathbb{F}_2$ be the coefficients of the characteristic polynomial of a over $\mathbb{F}_2$ as given in equation (3.1). Then $\mathcal{K}_q(a) \equiv 0 \pmod{128}$ if and only if the conditions of Corollary 3.1.13 are satisfied, and furthermore,*

$$e_{16} = e_3 + e_3(e_7 + e_{10} + e_{13}) + e_5(e_6 + e_{11}) + e_6 e_{10} + e_7 e_9.$$

3.2 An Algorithm for Discovering Relations among Coefficients of the Characteristic Polynomial

In this section, we look for relations of the form

$$0 = \sum c_i e_i + \sum c_{ij} e_i e_j + \cdots \tag{3.3}$$

by plugging in e_i values for the minimal polynomials obtained in Chapter 3 and solving for unknown coefficients $c_i, c_{ij}, \ldots$ over $\mathbb{F}_2$.

From Chapter 3, we have obtained lists of minimal polynomials over $\mathbb{F}_2$ of Kloosterman zeros over $\mathbb{F}_{2^m}$ for $m \leq 63$ using Algorithm 2.3. We have invented an algorithm that discovers relations among the coefficients of characteristic polynomials of Kloosterman zeros over $\mathbb{F}_2$. This algorithm is the topic of the present section. Our algorithm utilizes the *M4RI* library [29]. This library is also used in Sage, but we use it as a standalone library. In its simple form, this algorithm requires an $r \times m$ matrix C over $\mathbb{F}_2$ where $q = 2^m$. The matrix C stores values of $e_1, \ldots, e_m$ of r distinct minimal polynomials over $\mathbb{F}_2$ of Kloosterman zeros in $\mathbb{F}_{2^m}$ where e_i is defined in equation (3.1). Each row of C represents one minimal polynomial of a Kloosterman zero over $\mathbb{F}_2$ and each column of C represents one e_i. The entry $C[i][j]$ equals e_j of the i-th minimal polynomial. Notice that the number of rows r can be varied.

Definition 3.2.1. *The* kernel *of a matrix A is the vector space consisting of precisely those vectors x for which $Ax = 0$.*

Algorithm 3.1 requires an $nrows \times m$ matrix C over $\mathbb{F}_2$ (C stores the coefficients e_i of $nrows$ minimal polynomials for Kloosterman zeros in $\mathbb{F}_{2^m}$ as its rows). Moreover, our algorithm needs a positive integer n which is the number of the e_i that will be used (That is, relations among $e_1, e_2, \ldots, e_n$ and their products are sought.)

Our algorithm returns "No Relations Found" if there are no relations among the e_i and their products; otherwise, return a basis for conjectured relations.

Our algorithm computes a basis for the kernel of a matrix A over $\mathbb{F}_2$. The matrix A is obtained by setting its column vectors to be column vectors of C or product of column vectors of C. Since the kernel of A represents the relations among the coefficients and we only want to find independent relations, we decide to compute X a basis of the kernel of A. Since we need to find the relations among e_i and products e_ie_j and so on, we need to store "e_i" and "e_ie_j" as string values in vector N which are labels for the columns of A. For example, we have $N[12] =$ "e_3e_6" if and only if the 12-th column of A stores e_3e_6 for each minimal polynomial used in the computation.

At line 25 of Algorithm 3.1, the rows of matrix X give a basis for the kernel of matrix

A and vector N is as defined above. For example, if the algorithm finds

$$X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and } N = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} \quad \text{then } XN = \begin{bmatrix} e_1 \\ e_2 \\ e_4 \end{bmatrix}$$

which means that we have discovered three linearly independent relations $e_1 = 0, e_2 = 0$ and $e_4 = 0$. (See Corollary 3.1.11.)

We denote n to be the number of e_is. Then we can set $maxCols$ for given n as follows

$$maxCols = \sum_{j=1}^{k} \binom{n}{j}$$

where $k = 1$ if we are interested in finding relations with single e_i and $k = 2$ if we search for relations of single e_i and products $e_i e_j$ and so on.

We observe that $maxCols$ is an upper bound of the number of columns of A. To obtain new relations, we have to remove the relations that are implied by previous results. For example, we have $e_8 = e_3 e_5$ by Corollary 3.1.13. Then we get $e_3 e_8 = e_3 e_5$ and $e_5 e_8 = e_3 e_5$ by multiplying by e_3 and e_5, respectively. We have

$$e_8 = e_3 e_5 = e_3 e_8 = e_5 e_8. \tag{3.4}$$

Thus, we have to remove columns that store $e_3 e_5$, $e_3 e_8$ and $e_5 e_8$ from the matrix A in order to prevent rediscovery of these relations (and their multiples) in later runs of the algorithm. It is clear that the number of columns of A becomes less than $maxCols$.

Let us consider how many rows of A are needed for the success of the algorithm. (This number is stored in the $nrows$ variable in the description of the algorithm.) If we set $nrows$ too low, we may get bogus relations. For instance, if we set $nrows = maxCols$, then A becomes a square matrix and the probability of a square matrix over $\mathbb{F}_2$ to be singular is high (in fact it approaches $0.288...$ for large square matrices).

The probability of a random $r \times c$ binary matrix to have rank s where $s \leq \min(r, c)$ denoted by $P_{r,c}(s)$ is given in [15]:

$$P_{r,c}(s) = 2^{s(r+c-s)-rc} \prod_{i=0}^{s-1} \frac{(1 - 2^{i-r})(1 - 2^{i-c})}{1 - 2^{i-s}}. \tag{3.5}$$

Note that equation (3.5) holds for random matrices. In order to avoid bogus relations with high probability, we use the following heuristic. We will use an $r \times c$ matrix A where $r > c$ if $P_{r,c}(c)$ is close 1. That, is a random matrix A would produce a bogus relation with probability close to 0.

For $r = c = s = n$, we have

$$P_{n,n}(n) = \prod_{i=0}^{n-1}(1 - 2^{i-n}),$$

which approaches $0.288788...$ as n grows to infinity.

Appendix A.5 is our C++ implementation for Algorithm 3.1 with the *M4RI* library [29] with single e_i and products e_ie_j for $1 \leq i < j \leq 32$ on $\mathbb{F}_{2^{70}}$.

After running our C++ implementation for Algorithm 3.1 on several fields, we have obtained Results 3.2.2 through 3.2.6, which have been proved in the literature (see Section 3.1) and Conjectures 3.2.7 and 3.2.8. Note that we have no proofs but only conjectures.

Result 3.2.2. *Let $a \in \mathbb{F}_q^*$ and $\mathcal{K}_q(a) = 0$. Let $e_1, \ldots, e_m \in \mathbb{F}_2$ be the coefficients of the characteristic polynomial of a over $\mathbb{F}_2$ as given in equation (3.1) for $m \geq 4$. We have*

$$\begin{aligned} 0 &= e_1 \\ 0 &= e_2 \\ 0 &= e_4 \end{aligned}$$

as expected from Corollary 3.1.11.

We insert a random bit on the e_1, e_2 and e_4 columns of A to remove Result 3.2.2.

After removing Result 3.2.2, we could not find any linear relations of e_i for $i \leq 70$. Thus, we decided to include the products e_ie_j for $1 \leq i < j \leq 8$. We have the following result.

Result 3.2.3. *Let $a \in \mathbb{F}_q^*$ and $\mathcal{K}_q(a) = 0$. Let $e_1, \ldots, e_m \in \mathbb{F}_2$ be the coefficients of the characteristic polynomial of a over $\mathbb{F}_2$ as given in equation (3.1) for $m \geq 8$. After removing Result 3.2.2, we have*

$$\begin{aligned} 0 &= e_3e_5 + e_8 \\ 0 &= e_3e_5 + e_3e_8 \\ 0 &= e_3e_8 + e_5e_8 \end{aligned}$$

Algorithm 3.1 Discovering New Relations among e_is

Input: An $nrows \times m$ matrix C over $\mathbb{F}_2$ and the number of e_is, n.

Output: "No Relations Found" or a basis for conjectured relations.

Set A to be $nrows \times maxCols$
Set N to be a vector whose length is $maxCols$, which will contain the names of columns
$ncols \leftarrow 1$
for $j = 1$ to n **do**
  for $i = 1$ to $nrows$ **do**
    $A[i][ncols] \leftarrow C[i][j]$
  end for
  $N[ncols] \leftarrow$ "e_j", $ncols \leftarrow ncols + 1$
end for
for $j = 1$ to n **do**
  for $k = j + 1$ to n **do**
    for $i = 1$ to $nrows$ **do**
      $A[i][ncols] \leftarrow C[i][j] \cdot C[i][k]$
    end for
    $N[ncols] \leftarrow$ "$e_j e_k$", $ncols \leftarrow ncols + 1$
  end for
end for
{We use the same approach for $e_i e_j e_k$ and $e_i e_j e_k e_l$ if desired.}
Set A to be the submatrix of A from $A[1, 1]$ to $A[nrows, ncols]$ so that A becomes a $nrows \times ncols$ matrix
Row-echelonize A
if $rank(A) = ncols$ **then**
  return No Relation Found
else
  $X \leftarrow$ basis for the kernel of A
  Set N to be $N[1] \ldots N[cols]$; then the size of N becomes $ncols$
  return XN
end if

as expected from Corollary 3.1.13.

We have shown that $e_8 = e_3e_5$ implies $e_3e_5 = e_3e_8$ and $e_3e_8 = e_5e_8$ by equation (3.4).

We also provide an algorithm that checks whether a relation is independent of a set of relations. This is given as Algorithm 3.2.

Algorithm 3.2 Checking Dependence of Relations

Input: $F = \{f_1, \dots f_r\}$ where $f_i \in \mathbb{F}_2[e_1, \dots, e_n]$ and $g \in \mathbb{F}_2[e_1, \dots, e_n]$

Output: If $\forall e \in \mathbb{F}_2^n \ ((f_1(e) = 0) \ \& \ (f_2(e) = 0) \ \& \dots \& \ (f_r(e) = 0)) \Rightarrow (g(e) = 0)$, return $True$. Otherwise return $False$.

1: Set $KnownVariety$ be the set of vectors $e \in \mathbb{F}_2^n$ that satisfy $f_i(e) = 0$ for each $i = 1, \dots, r$
2: Set $NewVariety$ be the set of vectors $e \in \mathbb{F}_2^n$ that satisfy $g(e) = 0$
3: **if** $KnownVariety \subseteq NewVariety$ **then**
4: **return** $True$
5: **else**
6: **return** $False$
7: **end if**

Example 3.2.4. *From Result 3.2.3, we have 3 relations* $0 = e_3e_5 + e_8$, $0 = e_3e_5 + e_3e_8$, *and* $0 = e_3e_8 + e_5e_8$. *We will show* $0 = e_3e_5 + e_8$ *implies* $0 = e_3e_5 + e_3e_8$ *by Algorithm 3.2. We have* $F = \{e_3e_5 + e_8\}$ *and* $g = e_3e_5 + e_3e_8$. *Without loss of generality, we can only consider coordinates* e_3, e_5, e_8 *for the vectors* e. *Then we have*

$$KnownVariety = \{(0,0,0), (0,1,0), (1,0,0), (1,1,1)\},$$

$$NewVariety = \{(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,1,1)\}.$$

We have $KnownVariety \subseteq NewVariety$, *thus* $0 = e_3e_5 + e_8$ *implies* $0 = e_3e_5 + e_3e_8$.

There is also Magma code for Example 3.2.4 running Algorithm 3.2 in Appendix A.6.

We deleted the columns e_3e_5, e_3e_8 and e_5e_8 from matrix A and successfully removed Result 3.2.3. After removing Result 3.2.3 and running Algorithm 3.1 with single e_i and products e_ie_j for $1 \leq i < j \leq 16$, we have the following result.

Result 3.2.5. *Let $a \in \mathbb{F}_q^*$ and $\mathcal{K}_q(a) = 0$. Let $e_1, \ldots, e_m \in \mathbb{F}_2$ be the coefficients of the characteristic polynomial of a over $\mathbb{F}_2$ as given in equation (3.1) for $m \geq 16$. We have*

$$0 = e_3 + e_3e_7 + e_3e_{10} + e_3e_{13} + e_5e_6 + e_5e_{11} + e_6e_{10} + e_7e_9 + e_{16}$$

as expected from Corollary 3.1.15.

We decided to remove the column of e_3e_7 from matrix A to get rid of this relation. After removing Result 3.2.5, we have tested up to $n \leq 70$; nonetheless, we have not found any new relation in e_i and e_ie_j. We started searching for relations on e_i, e_ie_j and $e_ie_je_k$.

Result 3.2.6. *Let $a \in \mathbb{F}_q^*$ and $\mathcal{K}_q(a) = 0$. Let $e_1, \ldots, e_m \in \mathbb{F}_2$ be the coefficients of the characteristic polynomial of a over $\mathbb{F}_2$ as given in equation (3.1) for $m \geq 16$. Then we have*

$$\begin{aligned}
0 &= e_8 + e_5e_6 + e_5e_{11} + e_5e_{16} + e_7e_8 + e_8e_{10} + e_8e_{13} + e_5e_6e_{10} + e_5e_7e_9 \\
0 &= e_{16} + e_3e_9 + e_3e_{16} + e_5e_6 + e_5e_{11} + e_6e_8 + e_6e_{10} + e_8e_{11} + e_9e_{16} + e_3e_6e_{10} \\
&\quad + e_3e_9e_{10} + e_3e_9e_{13} + e_5e_6e_9 + e_5e_9e_{11} + e_6e_9e_{10} \\
0 &= e_5e_6 + e_5e_{11} + e_5e_{16} + e_6e_8 + e_8e_{11} + e_8e_{16} + e_5e_6e_{10} + e_5e_7e_9 + e_6e_8e_{10} + e_7e_8e_9 \\
0 &= e_6e_{10} + e_7e_9 + e_7e_{16} + e_{10}e_{16} + e_{13}e_{16} + e_5e_6e_7 + e_5e_6e_{10} + e_5e_6e_{13} + e_5e_7e_{11} \\
&\quad + e_5e_{10}e_{11} + e_5e_{11}e_{13} + e_6e_7e_{10} + e_6e_{10}e_{13} + e_7e_9e_{10} + e_7e_9e_{13}.
\end{aligned}$$

We have tested these results by Algorithm 3.2 and it shows the previous results from Corollaries 3.1.13 and 3.1.15 imply Result 3.2.6. We removed all columns that represent multiples of e_5e_6 and e_6e_8 to get rid of these results. We have not found any new relations in e_i, e_ie_j and $e_ie_je_k$ when $n \leq 70$. We started searching for relations in e_i, e_ie_j, $e_ie_je_k$ and $e_ie_je_ke_l$ and we found two new relations. By Algorithm 3.2, we have checked that these new results are independent of known corollaries in Section 3.1 and also independent of each other.

Conjecture 3.2.7. *Let $a \in \mathbb{F}_q^*$ and $\mathcal{K}_q(a) = 0$. Let $e_1, \ldots, e_m \in \mathbb{F}_2$ be the coefficients of the characteristic polynomial of a over $\mathbb{F}_2$ as given in equation (3.1) for $m \geq 32$. Then we have*

$$\begin{aligned}
0 = e_8(&e_{10}e_{11}e_{13} + e_{10}e_{13} + e_{10}e_{12} + e_{10}e_{11}e_{21} + e_{10}e_{12}e_{14} + e_{10}e_{11} + e_{10}e_{12}e_{20} + e_{10}e_{12}e_7 + \\
&e_{10}e_{11}e_{16} + e_{10}e_{13}e_{16} + e_{10}e_{13}e_{19} + e_{13} + e_{10}e_{13}e_7 + e_{10}e_{11}e_{15} + e_{10}e_{14}e_{18} + e_{10}e_{14}e_9 +
\end{aligned}$$

$e_{10}e_{15}e_{17} + e_{23}e_9 + e_{10}e_{16}e_9 + e_{10}e_{17}e_9 + e_{10}e_{18}e_7 + e_{10}e_{19}e_7 + e_{10}e_{17} + e_{10}e_{18} + e_{23} + e_{10}e_{23}e_9 + e_{10}e_{19} + e_{10}e_{20} + e_{10}e_{21} + e_{10}e_{22} + e_{10}e_{23} + e_{11}e_{13}e_9 + e_{10}e_{24} + e_{10}e_{25}e_7 + e_{10}e_{27} + e_{10}e_{29} + e_{10}e_{32} + e_{11}e_{12}e_7 + e_{11}e_{12} + e_{11}e_{16}e_9 + e_{11}e_7e_9 + e_{11}e_{13}e_7 + e_{12}e_{13}e_7 + e_{12} + e_{11}e_{17} + e_{11}e_{21} + e_{11}e_{26} + e_{11}e_7 + e_{11}e_9 + e_{12}e_{13} + e_{12}e_{14} + e_{13}e_{16} + e_{12}e_{16} + e_{12}e_{20} + e_{12}e_7 + e_{13}e_{15} + e_{13}e_{16}e_7 + e_{12}e_{16}e_7 + e_{13}e_{17} + e_{13}e_{19} + e_{13}e_{26} + e_{13}e_7e_9 + e_{13}e_7 + e_{14}e_{18} + e_{14}e_9 + e_{15}e_{16} + e_{15}e_{17} + e_{15}e_7e_9 + e_{15}e_7 + e_{16}e_{17} + e_{16}e_{26} + e_{16}e_9 + e_{17}e_7e_9 + e_{17}e_7 + e_{17}e_9 + e_{17} + e_{18}e_7 + e_{18} + e_{19}e_7 + e_{19} + e_{20} + e_{21} + e_{22} + e_{24} + e_{25}e_7 + e_{26}e_7e_9 + e_{26}e_7 + e_{27} + e_{29} + e_{32})$.

Conjecture 3.2.8. *Let $a \in \mathbb{F}_q^*$ and $\mathcal{K}_q(a) = 0$. Let $e_1, \ldots, e_m \in \mathbb{F}_2$ be the coefficients of the characteristic polynomial of a over $\mathbb{F}_2$ as given in equation (3.1) for $m \geq 32$. Then we have*

$0 = e_{10}e_{11}e_{13}e_3 + e_{10}e_{11}e_{13}e_8 + e_{10}e_{11}e_{15}e_3 + e_{10}e_{11}e_{15}e_5 + e_{10}e_{11}e_{16} + e_{10}e_{11}e_{19}e_5 + e_{10}e_{11}e_{21} + e_{10}e_{11}e_6e_9 + e_{10}e_{11}e_7e_9 + e_{10}e_{11}e_7 + e_{10}e_{12}e_{14}e_3 + e_{10}e_{12}e_{20} + e_{10}e_{12}e_5 + e_{10}e_{12}e_6e_7 + e_{10}e_{13}e_{15}e_3 + e_{10}e_{13}e_{19}e_3 + e_{10}e_{13}e_{19} + e_{10}e_{13}e_5e_7 + e_{10}e_{13}e_5e_9 + e_{10}e_{13}e_6e_7 + e_{10}e_{13}e_6 + e_{10}e_{13}e_7e_8 + e_{10}e_{14}e_{18} + e_{10}e_{14}e_3e_6 + e_{10}e_{14}e_3 + e_{10}e_{14}e_6 + e_{10}e_{14}e_8 + e_{10}e_{14}e_9 + e_{10}e_{15}e_{16} + e_{10}e_{15}e_{17} + e_{10}e_{15}e_3 + e_{10}e_{15}e_5e_7 + e_{10}e_{15}e_7e_9 + e_{10}e_{15}e_8 + e_{10}e_{16}e_{19} + e_{10}e_{16}e_7e_8 + e_{10}e_{16}e_7e_9 + e_{10}e_{16} + e_{10}e_{17}e_3e_6 + e_{10}e_{17}e_3e_9 + e_{10}e_{17}e_3 + e_{10}e_{17}e_5 + e_{10}e_{18}e_7 + e_{10}e_{18}e_8 + e_{10}e_{19}e_3 + e_{10}e_{19}e_7e_9 + e_{10}e_{20}e_3e_6 + e_{10}e_{20}e_3 + e_{10}e_{20}e_6 + e_{10}e_{21}e_8 + e_{10}e_{22}e_5 + e_{10}e_{23}e_3 + e_{10}e_{23}e_9 + e_{10}e_{24}e_8 + e_{10}e_{25}e_7 + e_{10}e_{26}e_3 + e_{10}e_{26}e_6 + e_{10}e_{27}e_5 + e_{10}e_{29}e_3 + e_{10}e_3 + e_{10}e_{32} + e_{10}e_5e_7 + e_{10}e_5e_9 + e_{10}e_5 + e_{10}e_6 + e_{10}e_7e_8 + e_{10}e_8e_9 + e_{10}e_8 + e_{11}e_{12}e_8 + e_{11}e_{13}e_3 + e_{11}e_{15}e_3 + e_{11}e_{15}e_5 + e_{11}e_{16}e_5 + e_{11}e_{16} + e_{11}e_{21} + e_{11}e_6e_9 + e_{11}e_7e_9 + e_{11}e_7 + e_{11}e_8e_9 + e_{11}e_8 + e_{12}e_{13}e_8 + e_{12}e_{14}e_3 + e_{12}e_{16}e_8 + e_{12}e_{20} + e_{12}e_5 + e_{12}e_6e_7 + e_{12}e_7e_8e_9 + e_{12}e_7e_8 + e_{13}e_{15}e_3 + e_{13}e_{19}e_3 + e_{13}e_{19}e_8 + e_{13}e_{19} + e_{13}e_5e_7 + e_{13}e_5e_9 + e_{13}e_6e_7 + e_{13}e_6 + e_{13}e_7e_8e_9 + e_{13}e_8e_9 + e_{14}e_{18} + e_{14}e_3e_6 + e_{14}e_3 + e_{14}e_6 + e_{14}e_8 + e_{14}e_9 + e_{15}e_{16} + e_{15}e_{17} + e_{15}e_3 + e_{15}e_5e_7 + e_{15}e_7e_9 + e_{15}e_8 + e_{16}e_{19}e_5 + e_{16}e_{19} + e_{16}e_5e_7e_9 + e_{16}e_5 + e_{16}e_7e_9 + e_{16}e_8e_9 + e_{16}e_8 + e_{16} + e_{17}e_3e_6 + e_{17}e_3e_9 + e_{17}e_3 + e_{17}e_5 + e_{18}e_7 + e_{18}e_8 + e_{19}e_3 + e_{19}e_5e_7e_9 + e_{19}e_7e_8 + e_{19}e_7e_9 + e_{20}e_3e_6 + e_{20}e_3 + e_{20}e_6 + e_{21}e_8 + e_{22}e_5 + e_{23}e_3 + e_{23}e_9 + e_{24}e_8 + e_{25}e_7 + e_{26}e_3 + e_{26}e_6 + e_{27}e_5 + e_{29}e_3 + e_3 + e_{32} + e_5e_7 + e_5e_9 + e_5 + e_7e_8e_9 + e_8e_9 + e_8$.

Due to memory shortage problem, we only used $n \leq 40$ for $e_ie_je_ke_l$. Later, we can run on a bigger machine up to $n = 63$ or higher.

Notice that Conjecture 3.2.7 and 3.2.8 are not the shortest output. Combining with

other previous known relations, they might be given in a shorter form. However, we did not pursue that task in this thesis.

We have checked that Conjectures 3.2.7 and 3.2.8 hold for Kloosterman zeros in $\mathbb{F}_{2^m}$ for $32 \leq m \leq 42$.

Appendix A

Implementations

A.1 Computing Classical Modular Polynomials over $\mathbb{Z}_2$

```
/* This implimentation computes j-invariant as q-series up to q^(B*B) */
#include <NTL/GF2X.h>
#include <cstdio>
#include <cstdlib>
#include <list>
#include <ctime>

NTL_CLIENT

// series = q^(-v)\sum (q^m)(j)
struct series
{
  long long v;
  long long m;
  GF2X j;
};

// Phi = sum(all e in Entry) X^(first)*Y^(second) + X^(second)*Y^(first)
struct Entry{
  unsigned long long first;
  unsigned long long second;
};
```

```
// B should be even
# define B 2000
# define SIZE (B*B)

long long gcd(const long long &a, const long long &b);
bool isprime(const unsigned long long& l);

series J[B+1];
void j_series();
void compute_jseries();
void update_jlseries(series& x,
const series& a, const unsigned long long& l);

void print_series(const series& S);
void print_reverse(const series& S);

void sqr_series(series& x, const series& a);
void mul_series(series& x, const series& a, const series& b);

void getL(series& L, const series& j, const series& jl,
          const unsigned long long& l);
void updateL(series& L, const series& a);
void updateL2(series& L, const series& jl,
const unsigned long long& l);

int main() {
  unsigned long long l, pointCount;
  long long i;
  lldiv_t ik;
  clock_t start, end, begin;
  double cpu_time_used;

  l = 1501;
  if (l%2 == 0)
    l++;
  while (!(isprime(l)))
    l = l+2;
```

```
start = clock();
SetCoeff(J[0].j, 1); // J[0] = 1;
J[0].v = 0;
J[0].m = 1;

j_series(); // update J[1];
compute_jseries();

while (l<B){
  printf("%llu\n", l);
  // Determine j^i(l\tau) from 0 <= i <= l+1
  series JL[l+2];
  SetCoeff(JL[0].j, 1); // JL[0] = 1;
  JL[0].v = 0;
  JL[0].m = 1;
  for (i=1; i<l+2; i++)
    update_jlseries(JL[i], J[i], l);

  // set L = J[l+1] + JL[l+1]
  // since each powers of all terms of L is (-l*(l+1)) mod 8
  series L, temp;
  getL(L, J[l+1], JL[l+1], l);
  // L = L + J[l]JL[l];
  mul_series(temp, J[l], JL[l]);
  updateL(L, temp);

  list<Entry> A;
  list<Entry>::iterator it;
  Entry ent;
  ent.first = l+1;
  ent.second = 0;
  A.push_back(ent);
  ent.first = l;
  ent.second = l;
  A.push_back(ent);
  pointCount = 3;
```

```
while (deg(L.j) >= 0) {
  long long d;
  d = L.m*deg(L.j)+L.v;
  ik = lldiv(d, l);

  ent.first = ik.quot;
  ent.second = ik.rem;
  A.push_back(ent);

  if (ent.first == ent.second)
    pointCount++;
  else
    pointCount+=2;

  if (ik.rem == 0) {
    updateL(L, J[ik.quot]);
    updateL2(L, JL[ik.quot], l);
  }
  if (ik.quot == 0) {
    updateL(L, J[ik.rem]);
    updateL2(L, JL[ik.rem], l);
  }
  else {
    mul_series(temp, J[ik.rem], JL[ik.quot]);
    updateL(L, temp);
    d = L.m*deg(L.j)+L.v;
    if ((ik.quot != ik.rem)) {
      mul_series(temp, J[ik.quot], JL[ik.rem]);
      updateL(L, temp);
    }
  }
}

printf("%llu\n", pointCount);
for (list<Entry>::iterator it = A.begin(); it != A.end(); it++)
  if ((it->first)==(it->second))
    printf("%lld %lld\n", it->first, it->second);
  else {
```

```
        printf("%lld %lld\n", it->first, it->second);
        printf("%lld %lld\n", it->second, it->first);
      }
    printf("-1\n\n");


    // printf ("Phi[%lld] := ", l);
    // for (list<Entry>::iterator it = A.begin(); it != A.end(); it++)
    //   if ((it->first)==(it->second))
    //         printf("+X^(%lld)*Y^(%lld) ", it->first, it->second);
    //   else
    //         printf("+X^(%lld)*Y^(%lld) + X^(%lld)*Y^(%lld) ",
    //                 it->first, it->second, it->second, it->first);
    // printf(":\n\n");

    l = l+2;
    while (!(isprime(l)))
      l = l+2;
  }
  return 0;
}

void j_series() {
  // later we need to use in mpz_int
  long long n, temp1, temp2;
  GF2X Num(0, 1);
  GF2X Den(0, 1);
  GF2X Jbar;

  n = 1;
  temp1 = 4*n*(3*n+1);
  temp2 = 16*n*(3*n+1);
  while (temp2 <= SIZE) {
    SetCoeff(Num, temp1);
    SetCoeff(Num, temp1-8*n);
    SetCoeff(Den, temp2);
    SetCoeff(Den, temp2-32*n);
    n++;
```

```
    temp1 = 4*n*(3*n+1);
    temp2 = 16*n*(3*n+1);
  }
  temp2 -= 32*n;
  if (temp2 <= B^2)
    SetCoeff(Den, temp2);

  while (temp1 <= SIZE) {
    SetCoeff(Num, temp1);
    SetCoeff(Num, temp1-8*n);
    n++;
    temp1 = 4*n*(3*n+1);
  }
  temp1 -= 8*n;
  if (temp1 <= SIZE)
    SetCoeff(Num, temp1);

  InvTrunc(Jbar, Den, SIZE);
  MulTrunc(Jbar, Jbar, Num, SIZE);

  for (long long i=0; 8*i<=SIZE; i++)
    if (coeff(Jbar, 8*i)==1)
      SetCoeff(J[1].j, i);

  J[1].v = 1;
  J[1].m = 8;
  trunc(J[1].j,J[1].j, (SIZE+J[1].v)/J[1].m + 1);
}

// computed (J(tau))^i, 2 <= i <= B
void compute_jseries() {
  for (long long i=2; i<B; i=i+2) {
    sqr_series(J[i], J[i/2]);
    mul_series(J[i+1], J[i], J[1]);
  }
  sqr_series(J[B], J[B/2]);
}
```

```
// this uses for printing J[i] or JL[i] where 1 <= i <= l+1
void print_series(const series& S) {
  for (long long  i=0; i <= deg(S.j); i++)
    if (coeff(S.j, i) == 1)
      printf (" + q^(%lld)", -(S.v)+S.m*i);
  printf (":\n");
}

// this uses for printing L
void print_reverse(const series& S){
  for (long long i = 0; i<= deg(S.j); i++)
    if (coeff(S.j, i) == 1)
      printf (" + q^(%lld)", -(S.v)-S.m*i);
  printf(":\n");
}

// x = a^2
void sqr_series(series& x, const series& a){
  x.m = a.m;
  x.v = 2*a.v;
  SqrTrunc(x.j, a.j, (SIZE+x.v)/x.m + 1);
}

// x = a*b
void mul_series(series& x, const series& a, const series& b){
  x.v = a.v + b.v;
  x.m = gcd(a.m, b.m);
  MulTrunc(x.j, a.j, b.j, (SIZE+x.v)/x.m +1);
}

// this uses compute JL[i], 1 <= i <= l+1
void update_jlseries(series& x, const series& a,
const unsigned long long& l){
  x.v = l*a.v;
  x.m = a.m;
  for (long long i=0; i< (SIZE+x.v)/(l*x.m) +1; i++)
    if (coeff(a.j, i)==1)
      SetCoeff(x.j, l*i);
```

```
}

// this sets L = J[l+1] + JL[l+1]
void getL(series& L, const series& j, const series& jl,
          const unsigned long long& l){
  series Rev;
  long long diffM;

  L.m = 8;
  L.v = -(-j.v % 8);

  // L = J[l+1]
  reverse(L.j, j.j, j.v/j.m);

  Rev.v = -(-jl.v % 8); // tempRev
  Rev.m = jl.m;
  reverse(Rev.j, jl.j, jl.v/jl.m);
  if ((Rev.v % Rev.m)==0)
    SetCoeff(Rev.j, 0, 0);

  // L = L + J[l+1]
  add(L.j, L.j, Rev.j);
}

// update L <- L + a
void updateL(series& L, const series &a){
  series Rev;
  Rev.v = -((-a.v) % (a.m));
  Rev.m = a.m;
  reverse(Rev.j, a.j, a.v/a.m);
  trunc(Rev.j, Rev.j, deg(L.j)+1);
  if ((Rev.v % Rev.m)==0)
    SetCoeff(Rev.j, 0, 0);
  add(L.j, L.j, Rev.j);
}

// this only uses when i=0 or k=0
// update L <-  L + JL[i] or L <- L + JL[k]
```

```
void updateL2(series& L, const series& jl,
const unsigned long long& l){
  series Rev;
  long long diffM;

  Rev.v = -(-jl.v % jl.m);
  Rev.m = jl.m;
  reverse(Rev.j, jl.j, jl.v/jl.m);
  if ((Rev.v % Rev.m)==0)
    SetCoeff(Rev.j, 0, 0);
  add(L.j, L.j, Rev.j);
}

// return true if l is prime
bool isprime(const unsigned long long& l){
  // finding primes l < B
  for (long long d = 3; (d*d) <= l; d += 2)
    if ((l%d) == 0)
      return false;
  return true;
}

// return gcd(a, b)
long long gcd(const long long &a, const long long &b){
  long long c, d, r;
  c = a;
  d = b;
  while (d != 0) {
    r = c % d;
    c = d;
    d = r;
  }
  return c;
}
```

A.2 Verifing a Kloosterman Zero

```
m:=21;
```

```
F := GF(2^m);
found_KlZ:=false;

while not found_KlZ do
 x := Random(F);
 y := Random(F);
 a := x^3+x*y+y^2;

 while (a eq 0) do
  x := Random(F);
  y := Random(F);
  a := x^3+x*y+y^2;
  end while;

  E := EllipticCurve([1, 0, 0, 0, a]);
  P := E![x, y];

  T := P;
  i := 0;
  while ((i lt m) and not(IsZero(T))) do
   // T = 2^i * P
   T := T+T;
   i := i+1;
  end while;

  if (i eq m) and (IsZero(T)) then
   assert (#E - 2^m) eq 0;
   found_KlZ:=true;
   printf P;
  end if;
end while;
```

A.3 Listing All Minimal Polynomials of Kloosterman Zeros

```
// trying to implement array of pointer that stores (i, j) for Phi_l
// implementing BFS searching lower level l first
```

```
// We assume m >= 6

#include <NTL/ZZ.h>
#include <NTL/GF2X.h>
#include <NTL/GF2XFactoring.h>
#include <NTL/GF2E.h>
#include <NTL/GF2EX.h>
#include <NTL/GF2EXFactoring.h>
#include <NTL/vec_GF2E.h>
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>
#include <list>
#include <algorithm>
#include <ctime>

using namespace std;
NTL_CLIENT

#define lSize sizeof(long)
#define S (sizeof(unsigned long long))

struct Power {
  long powerx;
  long powery;
};

static long n = 98;
static long Primes[50];
static long NumCoeff[50];
static unsigned long long BitMask[64];
typedef Power* ModPoly;

long m;
long minM=51, maxM=60; // mimM <= m <= maxM
ModPoly *Mcoeff;
```

```
// this returns the Legendre Symbol (a/n)
long Legendre(const ZZ& a, const long& n);

// this returns 64-bit unsigned long long
// when IsOne(coeff(z, i)) then
// return value contain ith position contains 1
unsigned long long convertGF2X(const GF2X& z);

// same as convertGF2X
// even if it shows as GF2EX,
// this is in fact GF2X because of the properties of conjugates
unsigned long long convertGF2EX(const GF2EX& z);

// Create Bitmask B[i] represent 2^i
void createBitMask();

// this print Minimal Polynomial whose roots are Kloosterman zeros
void printPoly(const GF2EX& z);
void printrecipPoly(const GF2EX& z) ;

// print GF2X, GF2E, GF2EX,
// the output can be readable by magma
void printGF2X(const GF2X& z);
void printGF2E(const GF2E& a);
void printGF2EX(const GF2EX& z);

unsigned long long FindMaxString(const vec_GF2E& z);
unsigned long long nextprime(unsigned long long x);

unsigned long long hash(unsigned long long x,
unsigned long long p) {
  unsigned long long h;
  h = x % p;
  if (h < 0)
    h+=p;
  return h;
}
```

```
bool insert(unsigned long long x,
unsigned long long *H, unsigned long long p ){
  unsigned long long h;
  h = hash(x,p);
  for(int i=1; H[h]!=0 && H[h]!=x; i++ )
    h = (h+i) %p; // quadratic probe
  if (H[h] == x)
    return false;
  else {
    H[h]=x;
    return true;
  }
}

int main() {
  fstream fin;
  long i, j, c, numPrimes=0;
  ModPoly *Mcoeff = new ModPoly[n/2];
  time_t start, end, mstart, mend, lstart, lend;
  bool finished = false;
  double cpu_time_used;

  createBitMask();

  // store modular polynomial on memory
  fin.open("ModPolyData100.txt", ifstream::in);
  while ((!(fin.eof())) && (!(finished))) {
    fin >> Primes[numPrimes] >> NumCoeff[numPrimes];
    Mcoeff[numPrimes] = new Power[NumCoeff[numPrimes]];
    for (int i=0; i < NumCoeff[numPrimes]; i++)
      fin >> Mcoeff[numPrimes][i].powerx
         >> Mcoeff[numPrimes][i].powery;
    fin >> c;
    numPrimes++;
    if (Primes[numPrimes] == 97)
      finished  = true;
  }
```

```
fin.close();

finished = false;
fin.open("get_IrrPoly_with_Random_KLZ.txt", ifstream::in);

while ((!(fin.eof())) && (!(finished))){
  // IrrPoly is irreducible polynomial degree m
  // so it will generate GF(2^m)
  // totalKLZ = total number of Kloosterman zeros over GF(2^m)
  // totalPoly = total number of minimal polynomials,
  //             which contain m Kloosterman zeros as roots
  // countKLZ = number of Kloosterman zeros that are found
  // countPoly = number of minimal polynomials that are found
  GF2X IrrPoly;
  long totalKLZ, totalPoly, countKLZ = 1, countPoly = 1;

  // ignore m < minM
  fin >> m;
  while (m < minM) {
    for (i=0; i<m; i++)
      fin >> c;
    fin >> totalKLZ;
    for (i=0; i<m; i++)
      fin >> c;
    fin >> c >> m;
  }

  // get Irreducible polynomial from disk
  SetCoeff(IrrPoly, m);
  for (i=0; i<m; i++) {
    fin >> c;
    if (c)
      SetCoeff(IrrPoly, i);
  }
  fin >> totalKLZ;
  totalPoly = totalKLZ/m;
  cout << "#m = " << m << endl;
```

```
// construct hash table
unsigned long long *H;
unsigned long long HashSize;

HashSize = nextprime(2*totalPoly);
H = (unsigned long long *) malloc(
 sizeof(unsigned long long)* HashSize );
for (i = 0; i<HashSize; i++)
  H[i] = 0;

// print magma code
cout << "\n// _____ GF(2^" << m << ") _____\n";
cout << "P<X> := PolynomialRing(GF(2));\n";
cout << "p := ";
printGF2X(IrrPoly);
cout << ";\n";
cout << "F := ext< GF(2) | p >;\n";
cout << "PE<X> := PolynomialRing(F);\n";
cout << "printf \"m = " << m << "\\n\";\n";

// initialize GF(2^m) using GF2 mod IrrPoly
GF2E::init(IrrPoly);
GF2E KLZ, ZERO; // ZERO = 0

// invKLZs will contains the one of roots of minimal polynomials
//         that are inserted into tree
// invzeros contains m inverse of Kloosterman zeros
//          in order to compute its minimal polynomial
// invzeros[i] is assigned as inverse of
//          Kloosterman zero KLZ^(2^(i)), 0 <= i < m
GF2EX recipPoly;
vector<GF2E> invKLZs(totalPoly);
vec_GF2E invzeros;
invzeros.SetLength(m);

// getting  information of one random Kloosterman Zero
// data is m-sequence of 0 or 1's
// if c = 1 then set X^i
```

```
for (i=0; i<m; i++){
  fin >> c;
  if (c)
    SetCoeff(KLZ._GF2E__rep, i);
}
fin >> c; // this is getting -1

// invKLZs[0] = 1/KLZ
inv(invKLZs[0], KLZ);
invzeros[0] = invKLZs[0];

cout << "KLZ" << m << " := { \n";
cout << "< ";
printGF2E(inv(invzeros[0]));
cout << ", ";

// insert first minimal polynomial
// whose roots are (KLZ^(2^i)) 0 <= i < m to tree
// Assume there is no memory allocation error
for (i = 1; i < m; i++)
  sqr(invzeros[i], invzeros[i-1]);

insert(FindMaxString(invzeros), H, HashSize);

// recipPoly = (X-invKLZ)(X-invKLZ^2)...(X-invKLZ^(2^(m-1)))
// recipPoly belongs to GF2X not GF2EX \
// because of the properties of conjugates
BuildFromRoots(recipPoly, invzeros);
printrecipPoly(recipPoly);
cout << " >";

// now getting classical modular polynomial l <= n-2
// LegendValue = t^2 - 4*q = 1 - 2^(m+2)

ZZ LegendValue;
LegendValue = 1;
LegendValue = 1 - (LegendValue << (m+2));
```

```
// x = X in GF2EX
GF2EX x(1, 1);
bool found = false, done = false;

// posKLZ is used for indicate the position of KLZ in invKLZs
// posP is used for indicate the position of current prime
// newLoc is where newKLZ is found after increase of the level.
//        need to start from small prime
long posKLZ, posP = 0, currP, n1, newLoc;
list<unsigned long long> l;
list<unsigned long long>::iterator it;

while ((posP < numPrimes) && (!(done))) {
  long k;
  if (Legendre(LegendValue, Primes[posP]) == -1)
    posP++;
  else {
    posKLZ = 0;
    currP = Primes[posP];
    n1 = currP+2;
    vector<GF2E> JINVPower(n1);
    set(JINVPower[0]);
    // precompute powers of invKLZ
    while (posKLZ < countPoly) {
      JINVPower[1] = invKLZs[posKLZ];
      for (i=2; i<n1; i++)
        if (i & 0x01)
          mul(JINVPower[i], JINVPower[i-1], JINVPower[1]);
        else
          sqr(JINVPower[i], JINVPower[i/2]);

      if (posKLZ < newLoc)
        k = posP;
      else
        k = 0;

      while ((k <= posP) && (!(done))) {
        long r, DEG = Primes[k]+2;
```

```
GF2EX f, h, g; // all f, h, g are initializes as 0
if (Legendre(LegendValue, Primes[k]) != -1) {
  // cout << "Prime = " << Primes[k] << "\n";
  // evaluation
  // COEFF store the coefficients of ModPoly(JINV, X);
  // COEFF[i] = 0, 0 <= i <= Primes[k]+1
  vector<GF2E> COEFF(DEG, ZERO);
  for (i=0; i<NumCoeff[k]; i++)
    COEFF[Mcoeff[k][i].powerx]
    += JINVPower[Mcoeff[k][i].powery];
  for (i=0; i<DEG; i++)
    SetCoeff(f, i, COEFF[i]);

  // applying root finding algorithm
  // h = X^(2^m) mod f
  FrobeniusMap(h, f);
  g = GCD(h-x, f);
  r = deg(g);

  if (r) {
    vec_GF2E Rts;
    FindRoots(Rts, g);
    for (int index = 0;
     ((index < Rts.length()) && (!(done))); index++){
      countKLZ++;
      invzeros[0] = Rts[index];
      for (i = 1; i < m; i++)
        sqr(invzeros[i], invzeros[i-1]);

      if (insert(FindMaxString(invzeros),
         H, HashSize )) {
        invKLZs[countPoly] = Rts[index];
        countPoly++;
        l.push_back(Primes[posP]);
        BuildFromRoots(recipPoly, invzeros);

        cout << ",\n";
        cout << "< ";
```

```
                printGF2E(inv(invzeros[0]));
                cout << ", ";
                printrecipPoly(recipPoly);
                cout << " >";

                if (countPoly == totalPoly) {
                  cout << "\n};\n";
                  free(H);
                  H = NULL;
                  done = true;

                  l.sort();
                  cout << "#l = { ";
                  it = l.begin();
                  long ll = *it;
                  it++;
                  cout << ll;
                  while (it != l.end()) {
                    if (ll != *it) {
                      ll = *it;
                      cout << ", " << ll;
                    }
                    it++;
                  }
                  cout << " };\n";
                }
              }
            }
          }
          k++;
        }
        posKLZ++;
      }
      posP++;
      newLoc = countPoly;
    }
  }
```

```
    if (m==maxM)
      finished = true;
  }
  fin.close();

  for (long i=0; i<numPrimes; i++)
    delete(Mcoeff[i]);
  delete(Mcoeff);

  return 0;
}

void createBitMask(){
  BitMask[0] = 1;
  for (int i=1; i<64; i++)
    BitMask[i] = (BitMask[i-1] << 1);
}

void printGF2X(const GF2X& z) {
  int plus = 0, tms = 0;
  for(int i=0;i<=deg(z);i++)
    if(coeff(z,i)!=0) {
      ++tms;
      if((tms%10)==0)
        printf("\n");
      if(plus==1)
        cout << "+";
      if(i==0)
        cout << "1";
      if(i==1)
        cout << "X";
      if(i>1)
        cout << "X^" << i ;
      plus=1;
    }
}

void printGF2EX(const GF2EX& z) {
```

```
  int plus =0, tms = 0;
  for(int i=0;i<=deg(z);i++) {
    if(coeff(z, i) != 0) {
      ++tms;
      if((tms%20)==0)
        printf("\n");
      cout << "+";
      printGF2E(coeff(z, i));
      if(i==1)
        cout << "*X";
      if(i>1)
        cout << "*X^" << i ;
      plus=1;
    }
    else
      cout << "+F![ 0 ]*X^"<<i;
  }
}

void printGF2E(const GF2E& b) {
  GF2X a;
  long i, da;
  GF2 c;

  a = b._GF2E__rep;
  da = deg(a);

  cout << "F![ ";

  if (da < 0)
    cout <<  0 ;

  for (i = 0; i <= da; i++) {
    c = coeff(a, i);
    if (c == 1)
      cout << "1";
    else
      cout << "0";
```

```
    if (i < da) cout << ", ";
  }
  cout << " ]";
}

void printPoly(const GF2EX& z) {
  //if (IsOne(coeff(z, 0)))
  cout << "1";
  if (IsOne(coeff(z, 1)))
    cout << "+X";
  for (int i=2; i<m; i++)
    if (IsOne(coeff(z, i)))
        cout << "+X^" << i;
  cout << "+X^" << m;
}

void printrecipPoly(const GF2EX& z) {
  //if (IsOne(coeff(z, 0)))
  cout << "1";
  if (IsOne(coeff(z, m-1)))
    cout << "+X";
  for (int i=m-2; i> 0; i--)
    if (IsOne(coeff(z, i)))
        cout << "+X^" << m-i;
  cout << "+X^" << m;
}

long Legendre(const ZZ& a, const long& n) {
  ZZ aa, nn;
  aa = a;
  nn = n;
  aa %= nn;
  return Jacobi(aa, nn);
}

unsigned long long convertGF2X(const GF2X& z) {
  unsigned long long x = 0;
  for (int i=0; i<m; i++)
```

```
    if (coeff(z, i) == 1)
      x ^= BitMask[i];
  return x;
}

unsigned long long convertGF2EX(const GF2EX& z) {
  unsigned long long x = 0;
  for (int i=0; i<m; i++)
    if (IsOne(coeff(z, i)))
      x ^= BitMask[i];
  return x;
}


unsigned long long FindMaxString(const vec_GF2E& z) {
  unsigned long long x, y;
  x = convertGF2X(z[0]._GF2E__rep);
  for (int i=1; i<m; i++) {
    y = convertGF2X(z[i]._GF2E__rep);
    if (y > x)
       x = y;
  }
  return x;

}


unsigned long long nextprime(unsigned long long x) {
  unsigned long long d;
  if( x&1 ) x += 2; else x += 1;
  for(; ; x+=2) {
    for( d=3; d*d < x && x%d != 0; d+=2 );
    if( d*d > x ) return x;
  }
}
```

A.4 Magma Code for the Kronecker Class Number

```
Z:=Integers();

H:=function(m)
Del:=1^2-4*2^m;
d2div := [ d : d in Divisors(-Del) | (Del mod d^2) eq 0 ];
d2div01 := [ d : d in d2div | (Z!(Del/d^2) mod 4) in {0,1} ];
return &+[ ClassNumber( Z!( (1^2-4*2^m)/d^2 ) )
     : d in d2div01 ];
end function;

for m:=1 to 64 do
printf "%o  %o\n",m, H(m);
end for;
```

A.5 Discovering New Relations among e_i and e_ie_j

```
// ei*ei = ei
// e1 = e2 = e4 = 0
// e1ei = e2ei = e4ei = 0
// e8 = e3e5 = e3e8 = e5e8
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <m4ri/m4ri.h>
#include <cmath>

using namespace std;

// en is the number of e_i's used
#define en 32

// max number of cols
#define MAXcols 42000

// max number of rows that can be handled in memory
#define MAXrows  50000
```

```
#define charPolyData_FILE  "charPolyData70.txt"
#define true (0==0)
#define false (!true)

int ncols;
int nrows;
char names[MAXcols][20];
int do_comma;

// this function prints the matrix
// with basis of columns are names[cols]
void print_basis_for_relations(mzd_t* X);

int main() {
  fstream fin;
  unsigned long long m, n, pos, loc;
  unsigned int c;
  int i, j, k, full;
  int j1, j2;
  // m is the degree of the char. polynomials
  // n is the number of the char. polynomials

  // open data file that contains
  // characteristic polynomials of KLZ over GF(2^m)
  fin.open(charPolyData_FILE, fstream::in);

  fin >> m >> n;
  cout << "//     m = " << m << "\n//_________________\n\n\n";
  printf( "en:=%d;\n", en );

  cout << "P<e1" ;
  for(i=2; i<= en ; i++)
    printf( ",e%d" , i );

  cout << ">:=PolynomialRing( GF(2), en );\n\n" ;
  cout << "\n// MAXrows = " << MAXrows << "\n";
  cout << "\n// MAXcols = " << MAXcols << "\n";
```

```
if (n>MAXrows)
  nrows=MAXrows;
else
  nrows=n;

// A will be nrows x m matrix
// nrows = the number of characteristic polynomials used
// m = the degree of the polynomial
// en = the number of e_i's

// A needs to be truncated before calling the kernel routine,
// to reduce cols to ncols.

mzd_t* A = mzd_init(nrows, MAXcols);
for(i=0; i<nrows; i++) {
  for (j=0; j<en; j++) {
    fin >> c;
    mzd_write_bit(A, i, j, c);
  }
  for (j=en; j<m; j++)
    fin >> c;
}
fin.close();

//  Destroy the relations e1=0, e2=0, e4=0.
for(i=0; i<nrows; i++)
  {
    mzd_write_bit(A, i, 0, 1 );
    mzd_write_bit(A, i, 1, rand()%2 );
    mzd_write_bit(A, i, 3, rand()%2 );
  }
sprintf( names[0] , "1" );

// for ei
for (i=1; i<en; i++)
  sprintf( names[i] , "e%d" ,  i+1 );
pos = en;
```

```
// ei*ej
for (j1 = 2; j1 < en; j1++)
  for (j2 = j1+1; j2 < en; j2++)
    if (
         ( !((j1 == 2) && (j2 == 3)) ) // e3e4
      && ( !(j1 == 3) )                // e4
      && ( !((j1 == 2) && (j2 == 4)) ) // e3e5
      && ( !((j1 == 2) && (j2 == 6)) ) // e3e7
      && ( !((j1 == 2) && (j2 == 7)) ) // e3e8
      && ( !((j1 == 4) && (j2 == 5)) ) // e5e6
      && ( !((j1 == 4) && (j2 == 7)) ) // e5e8
      && ( !((j1 == 5) && (j2 == 7)) ) // e6e8
 )
 {
   sprintf( names[pos] , "e%d*e%d" ,  j1+1 , j2+1 );
   for(i=0;i<nrows;i++)
     mzd_write_bit(A, i, pos,
       mzd_read_bit(A,i,j1) * mzd_read_bit(A,i,j2) );
   pos++;
   assert(pos< MAXcols);
 }

ncols = pos;
printf ("// nrows=%d\n",nrows);
printf ("// ncols=%d\n",ncols);
fflush(stdout);

// NOW TRUNCATE A to ncols columns.
A = mzd_submatrix(NULL, A, 0, 0, nrows, ncols);
printf ("// A truncated\n");
fflush(stdout);

int rankA = mzd_echelonize( A , full );
printf ("// rank A = %d\n",rankA);
fflush(stdout);
if (rankA < ncols) {
    mzd_t* X = mzd_transpose(NULL, mzd_kernel_left_pluq(A, 0));
    printf ("// kernel computed\n");
```

```
      mzd_echelonize(X, full);
      printf ("// X echelonized\n");
      do_comma=false;
      print_basis_for_relations(X);
      printf("];\n\n");
      mzd_free(X);
  }
  mzd_free(A);
  return 0;
}

void print_basis_for_relations(mzd_t* X) {
  cout << "basis_for_relations := [ \n";
  for (int i=0; i<X->nrows; i++){
    int count = -1;
    if(do_comma)
      printf(",\n");
    else
      do_comma=true;
    for (int j=0; j<X->ncols; j++) {
      if (mzd_read_bit(X, i, j)) {
        if (count == -1)
          cout << names[j] << " ";
        else
          cout << "+ " << names[j] << " ";
        count++;
        if (count == 10) {
          count = 0;
          cout << "\n";
        }
      }
    }
  }
  cout << "\n";
}
```

A.6 Dependency Test

```
// this is Magma code for dependency test
load "RELATIONS-FOR-THESIS.txt";

rel1 := basis_for_relations[1];
rel2 := basis_for_relations[2];

published_relations := [ e3*e5+e8 ,
// checked against the IEEE IT paper:
e3 + e16 + e3*e7 + e3*e10 + e3*e13
+ e5*e6 + e5*e11 + e6*e10 + e7*e9 ];

E := VectorSpace( GF(2), en );

W := { e : e in E |
( Evaluate( published_relations[1] ,
[ e[i] : i in [ 1 .. en ] ] ) eq 0 )
and
( Evaluate( published_relations[2] ,
[ e[i] : i in [ 1 .. en ] ] ) eq 0 )
};

V1 := { e : e in E | Evaluate( rel1 ,
[ e[i] : i in [ 1 .. en ] ] ) eq 0 }
V2 := { e : e in E | Evaluate( rel2 ,
[ e[i] : i in [ 1 .. en ] ] ) eq 0 }

W subset V1;
W subset V2;

V1 subset V2;
V2 subset V1;
```

Bibliography

[1] Blake, I., Seroussi, G., Smart, N.: Elliptic Curves in Cryptography. Cambridge University Press, 1999

[2] Bosma, W., Cannon, J., Playoust, C.: The Magma algebra system. I. The user language. J. Symbolic Comput. 24, 235–265 (1997)

[3] Carlet, C., Gaborit, P.: Hyper-bent functions and cyclic codes, J. Combin. Theory Ser. A 113, 466–482 (2006)

[4] Carlitz, C.: Kloosterman sums and finite field extensions. Acta Arithmetica XVI, 179–193 (1969)

[5] Charpin, P., Gong, G.: Hyperbent functions, Kloosterman sums and Dickson polynomials. IEEE Trans. Inform. Theory 54, 4230–4238 (2008)

[6] Charpin, P., Helleseth, T., Zinoviev, V.: Propagation characteristics of $x \mapsto x^{-1}$ and Kloosterman sums. Finite Fields Appl. 13, 366–381 (2007)

[7] Charpin, P., Helleseth, T., Zinoviev, V.: The divisibility modulo 24 of Kloosterman sums on $GF(2^m)$, m odd. J. Combin. Theory Ser. A 114, 322–338 (2007)

[8] Dillon, J.F.: Elementary Hadamard Difference Sets. Ph.D. dissertation, University of Maryland, 1974.

[9] Enge, A.: Elliptic Curves and Their Applications to Cryptography: An Introduction. Kluwer Academic Publishers, 1999

[10] Garaschuk, K., Lisoněk, P.: On binary Kloosterman sums divisible by 3. Designs, Codes and Cryptography 49, 347–357 (2008)

[11] Garaschuk, K., Lisoněk, P.: On ternary Kloosterman sums modulo 12. Finite Fields and Their Applications 14, 1083–1090 (2008)

[12] von zur Gathen, J., Gerhard, J.: Modern Computer Algebra. Cambridge University Press, 1999

[13] Geddes, K.O., Czapor, S.R., Labahn, G.: Algorithms for Computer Algebra. Kluwer Academic Publishers, 1992

[14] van der Geer, G., van der Vlugt, M.: Kloosterman sums and the p-torsion of certain Jacobians. Math. Ann. 290, 549–563 (1991)

[15] Fridrich, J., Goljan, M., Lisoněk, P., Soukal, D.: Writing on wet paper. IEEE Transactions on Signal Processing 53, 3923–3935 (2005)

[16] Göloğlu, F., Lisoněk, P., McGuire, G., Moloney, R.: Binary Kloosterman sums modulo 128 and coefficients of the characteristic polynomial. Submitted to IEEE Trans. Inf. Theory (November 2010).

[17] Göloğlu, F., McGuire, G., Moloney, R.: Binary Kloosterman sums using Stickelberger's theorem and the Gross-Koblitz formula. Submitted (2010)

[18] Helleseth, T., Zinoviev, V.: On Z_4-linear Goethals codes and Kloosterman sums. Des. Codes Cryptogr. 17, 269–288 (1999)

[19] Kononen, K., Rinta-aho, M., Väänänen, K.: On integer values of Kloosterman sums. IEEE Trans. Inform. Theory 56, 4011–4013 (2010)

[20] Lachaud, G., Wolfmann, J.: The weights of the orthogonals of the extended quadratic binary Goppa codes. IEEE Trans. Inform. Theory 36, 686–692 (1990)

[21] Lercier, R., Lubicz, D., Vercauteren, F.: Point Counting on Elliptic and Hyperelliptic Curves. In: Cohen, H., Frey, G. (eds.) Handbook of Elliptic and Hyperelliptic Curve Cryptography. Chapman & Hall/CRC, 2006

[22] Lidl, R., Niederreiter, H.: Introduction to Finite Fields and Their Applications. Cambridge University Press, 1994

[23] Lisoněk P.: On the connection between Kloosterman sums and elliptic curves. In: SETA, Lecture Notes in Computer Science, S.W. Golomb, M.G. Parker, A. Pott, and A. Winterhof, Eds., vol. 5203. Springer, 2008, pp. 182–187

[24] Lisoněk P.: Short proofs for Kloosterman sum identities. Submitted (2011)

[25] Lisoněk, P., Moisio, M.: On zeros of Kloosterman sums. Designs, Codes and Cryptography 59, 223–230 (2011)

[26] Menezes, A.: Elliptic Curve Public Key Cryptosystems. Kluwer Academic Publishers, 1993

[27] Moisio, M.: Kloosterman sums, elliptic curves, and irreducible polynomials with prescribed trace and norm. Acta Arith. 132, 329–350 (2008)

[28] Reilly, N: Introduction to Applied Algebraic Systems. Oxford University Press, 2009

[29] SAGE Project: M4RI library. Available at: http://m4ri.sagemath.org/

[30] Schoof, R.: Nonsingular plane cubic curves over finite fields. J. Comb. Theory Ser. A 46, 183–211 (1987)

[31] Shoup, V.: NTL library. Available at: http://www.shoup.net/ntl/

[32] Vercauteren, F.: The SEA algorithm in characteristic 2. (2000)
Available at: http://homes.esat.kuleuven.be/ fvercaut/papers/SEA.pdf. Retrieved 14 June 2011.

[33] Youssef, A.M., Gong, G.: Hyper-bent functions. In: Advances in Cryptology - EUROCRYPT 2001 (Innsbruck). Lecture Notes in Comput. Sci. Berlin: Springer, 2001, vol. 2045, pp. 406–419.