TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TEEMU STENHAMMAR
FACTORIZATION OF BINARY POLYNOMIALS

Master of Science thesis

ABSTRACT

This thesis describes a solution to a cryptographic programming puzzle originally posted by Nintendo [1] in order to gain job applicants. The encryption method turned out to be the same as binary polynomial multiplication which means that the decryption can be done with binary polynomial factorization.

While providing shallow exploration of other options, the main approach in this thesis was to first compute square-free factorization of a polynomial using David Yun's algorithm from 1974 [21] and then to apply slower Elwyn Berlekamp's algorithm [2] on those square-free factors to compute a proper irreducible factorization of the polynomial. In addition to just explaining and implementing algorithms, the details of how to make these computations fast on a computer system have been explained in detail.

One finding that was made during the research was that sometimes researchers in this field bypass the square-free factorization by just citing Yun and moving on. In fact, Yun's algorithm as such does not work for fields with positive characteristic, which often is the case, and Gianni's work [9] that extends square-free factorization to positive characteristic is not even based on Yun's algorithm: it is based on Musser's algorithm [21], which was published 3 years earlier than Yun's.

The binary polynomial factorization translates really efficiently to a computer algorithm where one bit represents one coefficient. Using this fact allowed author of this thesis to efficiently implement the algorithms to solved the puzzle as the 273rd person since the problem was posted on-line.

TIIVISTELMÄ

Tässä työssä kuvataan ratkaisu erääseen kryptografiseen ongelmaan, jonka peliyhtiö Nintendo julkaisi [1] tavoitteenaan tarjota työmahdollisuus ongelman ratkaisseille. Lähemmässä tarkastelussa selvisi, että heidän salausalgoritminsa keskiössä oli binääripolynomien kertolasku ja siten purkualgoritmi, ja siten ongelman ratkaisu vaatii binääripolynomien tekijöihin jakoa.

Itse ratkaisu koostuu kahdesta vaiheesta. Ensin binääripolynomi jaetaan neliöttömiin tekijöihin käyttäen David Yunin algoritmia vuodelta 1974 [21]. Tämän jälkeen neliöttömät tekijät jaetaan alkupolynomeihin käyttäen hieman hitaampaa Elwyn Berlekampin algoritmia [2]. Molemmat algoritmit toteutetaan C++ kielellä modernilla tietokoneella ja tuon toteutuksen tehokkuuteen kiinnitettään työssä erityistä huomiota. Näiden kahden algoritmin kuvaamisen lisäksi työssä esitellään pintapuolisesti muita tapoja jakaa polynomi tekijöihin äärellisen kentän yli tarkoituksena antaa kuva siitä, kuinka alan tutkimus on kehittynyt.

Työn tutkimusvaiheen aikana huomattiin, että alan kirjallisuudessa on epätarkkuuksia ja joskus aiempi tutkimus sivuttiin liian helposti tai viitattiin väärään työhön. Esimerkiksi Yunin algoritmiin viitataan helposti, kun kyse on polynomien jakamisesta neliöttömiin tekijöihin, vaikka Yunin oma työ on tarkoitettu nolla karakteristikan omaaviin kenttiin. Tämä on tärkeää siksi, että binääripolynomien kentän karakteristika on nollasta poikkeava, eikä Yunin algortimi siksi toimi. Kaiken lisäksi Giannin myöhemmin esittämät muutokset [9] eivät pohjaudu Yunin omaan algoritmiin, vaan Musserin algoritmiin [21], jonka Yun vain esitteli samassa julkaisussaan.

Binääripolynomit on hyvin tehokasta esittää tietokoneella niin, että yksi bitti vastaa yhtä kerrointa. Tätä hyväksikäyttäen työssä saatiin aikaiseksi tehokas toteutus, jolla päästiin 273ksi tehtävän suorittaneeksi.

CONTENTS

LIST OF ABBREVIATIONS AND SYMBOLS

AI	Artificial Intelligence
AND	Logical AND operation
BCH	Bose-Chaudhuri-Hocquenghem codes
CLMUL	Carry-less multiplication instruction set on modern microprocessors
ECC	Elliptic curve cryptography
GPU	Graphics processing unit
PC	Personal computer
PID	Principal ideal domain
PIR	Principal ideal ring
RSA	Rivest-Shamir-Adleman, one of the first and most well-known public-key cryptosystem
SETI	Search for extraterrestrial intelligence
UFD	Universal factorization domain
XOR	Logical exclusive OR operation

O	Mathematical notation providing an upper bound on a growth rate of a function
Ω	Mathematical notation providing a lower bound on a growth rate of a function
Θ	Mathematical notation bounding growth of a function both from above and below defining its exact asymptotic behavior
o	Mathematical notation about growth rate of a function. Statement $f(x) \in o(g(x))$ means that $g(x)$ grows much faster than $f(x)$
$\mathbb{Z}_p$	Field of residual classes $\mathbb{Z} \mod p$
$R[x]$	The ring of univariate polynomials over ring R with x as the indeterminate
$char(R)$	Characteristic of a ring R
$deg(p)$	The degree of a polynomial p
$gcd(a, b)$	A greatest common divisor of a and b
$rank(M)$	Rank of a matrix M
$rev(p)$	Reversing coefficients of polynomial p

1.   INTRODUCTION

Factorization of integers to their prime factors is a known problem for which current algorithms are not efficient for large enough integers. Even more, the factorization of integers has a crucial asymmetry: some mathematical computations are fast when prime factors are known, but when they are not, computations are infeasible. This asymmetry is what some encryption schemes, like RSA, are based on and they play a huge part in securing communications today.

It was long thought that the factorization of polynomials is as hard a problem as factorization of integers is, but in 1967 Berlekamp [2] came up with his ingenious algorithm and showed that if the set from which the coefficients come from, is finite, then there exists a deterministic polynomial time algorithm for factorization.

This has led to 50 years of research into different ways these polynomials may be factorized with different sets of restrictions. There have been direct improvements from solving the problem in phases, like Yun's algorithm [21], which finds square-free factors or algorithm by Cantor and Zassenhaus [5] that introduces equal-degree and distinct-degree factorizations. A more recent approach by Kaltoffen and Shoup brought the factorization down to sub-quadratic-time [12]. And even more recently Umans published his sub-quadratic-time algorithm [19], which does not rely on fast matrix multiplication, like Kaltoffen's and Shoup's did.

Presently, factorization of polynomials over finite fields is an important tool for many applications. Cyclic redundancy codes for error correction are based on polynomial rings and the work of Berlekamp in 1968 [11, p. 19]. Two of the most notable of such codes are Bose-Chaudhuri-Hocquenghem codes (BCH), where polynomial factorization over finite fields is used directly [11, p. 22], and Reed-Solomon codes that are basically non-binary BCH codes [11, p. 24].

Another application, elliptic curves, rose to everyone's knowledge when they were used to prove Fermat's last theorem. Particularly elliptic curve cryptography (ECC) is based on elliptic curves over finite fields [8] and relies on the same algebra and factorization of polynomials this thesis presents and is based on. ECC is making

its way to replace RSA in places like Wi-Fi even though its security has not been completely evaluated [8]. However, it seems that ECC can provide the same level of security as RSA, but with shorter keys [8].

The idea for this thesis came when entertainment company Nintendo posted a job-application-like programming challenge to one well-known website [1] for such game related programming puzzles. The problem was about decrypting a given encryption of a laid-out algorithm that looks obfuscated at first glance, but actually just does carry-less multiplication of two binary numbers. This happens to be exactly what multiplication of binary polynomials does, and hence, the link to factorization of binary polynomials is made.

This thesis is built around solving the challenge Nintendo posted. The first chapter is this introduction and it gives an overall picture about the field to which this thesis relates to, and then, chapter 2 introduces all necessary algebraic constructs required for both understanding and proving the algorithms used in the solution. Then, chapter 3 walks through all used algorithms for computing with polynomials and factorizing them. The challenge set by Nintendo, along with its full solution based on the theory and algorithms will be presented in chapter 4. Lastly, chapter 5 concludes the thesis and highlights results and difficulties met during implementation and writing.

2. ESSENTIAL ALGEBRAIC CONCEPTS

Algorithms used for solving the Nintendo challenge [1] rely on algebra of polynomial rings. So, in order to explain those, there needs first to be a clear understanding of monoids 2.1, groups 2.2, rings 2.3 and fields 2.6 with some more in-depth concepts like principal ideal domains 2.4 and universal factorization domains 2.5

Along with the algebra, a good understanding of some properties of polynomials is required. The background for polynomials begin with explanation of polynomial rings in section 2.7. Then, operations on polynomial are presented in section 2.8. Last section, section 2.9, provides a brief summary of the Chinese remainder theorem which is later required to prove an important property of an algorithm.

2.1 Monoid

For a set S a mapping $S \times S \to S$ may be called a *law of composition*. For any $(x, y) \in S$ the law of composition is also called their *product* and the commonly used notation for it is xy or $x \cdot y$. [15, p. 3]

If for all elements $x, y, z \in S$ the relation $x(yz) = (xy)z$ holds the law of composition for S is said to be *associative* [15, p. 3]. Next, if there exists an element $e \in S$ such that for any $x \in S$ it holds that $ex = x = xe$, the element e is called a *neutral element*. The neutral element for a set S is unique, because for any other neutral element e' it would hold that $e = ee' = e'$. With this we can give the following definition.

Definition 2.1.1. A *monoid* is a set G with a law of composition, which is associative, and having a neutral element [15, p. 3]. This definition also means that the set G may not be an empty set for a monoid.

In addition to being associative, a law of composition of a set G can be *commutative*. It means that for all $x, y \in G$ relation $xy = yx$ holds [15, p. 4]. If this is true, G is said to be commutative, or sometimes *abelian*.

2.2   Groups

Given a monoid G, an *inverse* of an element $x \in G$ is an element $y \in G$ such that $xy = e = yx$ [15, p. 7]. The inverse element is usually denoted with superscript -1, i.e. the inverse of an element a is a^{-1}. The inverse element must be unique, because if another inverse $y' \in G$ exists, then

$$y' = y'e = y'(xy) = (y'x)y = ey = y. \, [15, p.7]$$

Definition 2.2.1. A monoid G is a *group* if and only if every element of G has an inverse.

An example of a group would be the set of rational numbers which forms a group under addition a.k.a. *additive group*, where the group operation is denoted as $+$ and the neutral element as 0. If the set was restricted to non-zero rational numbers it would form a group under multiplication, a *multiplicative group*, where the group operation is denoted as $\cdot$ and the neutral element as 1.

Groups may also have *subgroups*. A subgroup H of G is a subset of G which contains the neutral element and for which the law of composition and inverse are closed [15, p. 9], meaning for all $a, b \in H$ it holds that $ab \in H$ and $a^{-1} \in H$. If the neutral element is the only element the subgroup contains, it is called *trivial* [15, p. 9].

Let G be group and S a subset of G. We say S *generates* G, or S is the *generator* of G, if every element of G can be expressed as a product of elements of S or their inverses. A group G is called *cyclic* if it is generated by a set S of only one element. As an example $\mathbb{Z}$ forms a cyclic additive group as it is generated from 1 or -1. $\mathbb{Z}$ has no other generators.

If the law of composition of a group G is commutative, the group G is also said to be commutative or abelian [4, p. 63]. The group may also be either *finite* or *infinite* according to the size of it. All previous examples of groups have been infinite as the size of both integers and rational numbers is infinite. An example of a finite group would be integers modulo some n under addition.

Let $x, y \in \mathbb{Z}$ and n be a positive integer. We define $x \equiv y \mod n$ to mean that $x - y = nq$ for for some $q \in \mathbb{Z}$ [4, p. 64]. This makes $\equiv$ an equivalence relation and we denote the *equivalence class*, or the *residue class*, in which x belongs to with $\overline{x}$. The set of all possible equivalence classes is then denoted with $\mathbb{Z}_n$, or in some

literature $\mathbb{Z}/(n)$. When written out, this set is as follows:

$$\mathbb{Z}_n = \{\overline{0}, \overline{1}, \ldots, \overline{n-1}\}$$

Addition and multiplication operations for integers modulo n may be defined as follows:

$$\overline{x} + \overline{y} = \overline{x+y}$$

$$\overline{x} \cdot \overline{y} = \overline{x \cdot y}$$

The residue class $\overline{0}$ is clearly the neutral element for addition and $\overline{-x}$ is the inverse of $\overline{x}$, hence, $\mathbb{Z}_n$ is a group under addition [4, p. 64]. However, there is no inverse for $\overline{0}$ under multiplication, and therefore when multiplication is considered, $\mathbb{Z}_n$ is just a monoid.

2.3 Rings

A *ring* is a fundamental algebraic concept which consists of a nonempty set R and binary operations for addition and multiplication. Formally a ring is defined as follows [4, p. 159]:

Definition 2.3.1. A system $(R, +, \cdot)$, where R is a nonempty set and $+$ and $\cdot$ are two laws of composition, called addition and multiplication, and for which holds that [15, p. 83]:

1. R is a commutative group under addition.
2. Multiplication is associative and has a neutral element.
3. Multiplication is *distributive* over addition which means that
   for all $a, b, c \in R$: $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(a + b) \cdot c = a \cdot c + b \cdot c$

Some simple examples of rings are $\mathbb{Z}$, the ring of all integers, and $\mathbb{Q}$, the ring of all rational numbers. Both accompanied by the familiar addition and multiplication operations. $\mathbb{Z}_n$ was already established as a group over addition, but when multiplication is added, it forms a ring.

A ring is called *commutative ring* if its multiplicative operation is commutative [4, p. 160]. Furthermore, a ring R which has a zero element 0 and a unit element 1 such that $0 \neq 1$ and for which every non-zero element has an inverse is called a *division ring* [15]. Also, if $ab = 0$ for all $a, b \in R$, the ring R is called *trivial* [4, p. 160].

As an example of a non-commutative ring, let us consider $n \times n$ square matrices with entries from a ring R. Also, let addition and multiplication with such matrices be defined as they usually are for matrices. Now, if

$$a = \begin{bmatrix} x & 0 \\ 0 & 0 \end{bmatrix}, \quad b = \begin{bmatrix} 0 & y \\ 0 & 0 \end{bmatrix}, \quad xy \neq 0,$$

then it is easy to see that $ab \neq ba$, and hence, multiplication is not commutative. However, addition is commutative and multiplication is associative and distributive, hence, $n \times n$ square matrices over R form a ring.

If a ring R contains elements $a, b \neq 0$ such that $ab = 0$, then a and b are called *zero divisors*. Example of such zero divisors would be $2, 4 \in \mathbb{Z}_8$ because $2 * 4 \equiv 0 \mod 8$. When ring has no such zero divisors, it is called an *integral domain*, meaning that for all $a, b \in R, ab = 0 \Rightarrow a = 0 \vee b = 0$.

Multiplication of any element $a \in R$ with an integer n may be defined by using addition operation of the ring and the existence of a unit element. First, we observe that $1a = a$. Second, we may define that $(1 + 1)a = a + a$, and hence $2a = a + a$. And lastly, we can generalize this for any n:

$$na = \sum_{i=1}^{n} a.$$

Another property of a ring is its *characteristic*. It is either zero or a positive integer and tells how many times addition operation must applied to any element of the ring for the sum to be the additive identity 0. There exists different definitions for this property, but the result of each is the same. The following definition is from [4, 169].

Definition 2.3.2. If there exists a positive integer n such that $na = 0$ for each element a of a ring R, the smallest such positive integer is called the characteristic of R. If no such positive integer exists, R is said to have characteristic zero. The characteristic of R is denoted $char(R)$.

One consequence of this definition is that if the ring is of integers modulo n, denoted by $\mathbb{Z}_n$, then the $char(\mathbb{Z}_n) = n$. This follows directly from the fact that $na \equiv 0 \mod n$, when $a, n \in \mathbb{Z}$. As an another example, $\mathbb{Z}$ has a characteristic of 0.

2.4 Principal ideal domain

An *ideal* is a subset S of a ring R for which it holds that [4, p. 179]:

1. For $a, b \in S \Rightarrow a - b \in S$
2. For $a \in S$ and $r \in R$ implies $ar \in S$ and $ra \in S$

For non-commutative rings ideal may also be *right ideal* or *left ideal*, depending on if and which way the second rule holds. So, a right ideal is subset S_R of a ring R for which following holds

1. For $a, b \in S_R \Rightarrow a - b \in S_R$
2. For $a \in S_R$ and $r \in R$ implies $ar \in S_R$

and a left ideal a subset S_L of a ring R for which following rules hold

1. For $a, b \in S_L \Rightarrow a - b \in S_L$
2. For $a \in S_L$ and $r \in R$ implies $ra \in S_L$

All ideals for commutative rings are both left and right ideals due to the commutative nature of multiplication in these rings. These ideals are called *two-sided ideals* [4, p. 179] or just ideals. In every ring R there are two trivial ideals. Namely, 0 and R [4, p. 179]. All other ideals are considered non-trivial.

An example of an ideal would be all even integers. Addition of any two even integers would yield an even integer and multiplying an even integer with any other integer would yield an even integer. Hence, even integers fulfill the rules of an ideal above.

Now, for ring R and element $a \in R$ multiplication aR generates a right ideal of R which is called *principal* [15, p. 86]. Here element a is called the *generator* of that ideal [15, p. 86]. An example of such principal ideal would be all integers divisible by 3, denoted as $3\mathbb{Z}$. All elements of this ideal are generated as a set $I = \{3a | a \in \mathbb{Z}\}$.

The principal ideal in previous example was right principal ideal as the generation happened by multiplying from the right side. As with just ideals, the left principal ideal follows the exact same definition except multiplication happening from the left side. If, however, the principal ideal happens to be two-sided, it is denoted as RaR and defined as the set of all sums $\sum_i x_i a y_i$, where $x_i, y_i \in R$ [15, p. 86]. For commutative rings all three definitions of principal ideals are the same [15, p. 86].

Definition 2.4.1. If all ideals of a ring are principal ideals, the ring is called *principal ideal ring* (PIR) [4, p. 183].

Definition 2.4.2. A commutative integral domain with unity that is a principal ideal ring is called *principal ideal domain* (PID) [4, p. 183].

2.5   Unique factorization domains

Let R be a commutative integral domain ring with unity and $a, b \in R$ two non-zero elements. We say b divides a, denoted as $b|a$, if there exists such an element $c \in R$ that $a = bc$ [4, p. 212]. Another way of expressing the same is to say $a \equiv 0 \mod b$. Furthermore, if there exists a unit $u \in R$ such that $a = ub$, then a and b are called *associates* [4, p. 212].

Now, if $a \in R$ is not a zero or a unit and all divisors of a are either associates or units, it is called *irreducible* over R. Also, an element $p \in R$ is called a *prime element* if p is not a unit and $p|ab \Rightarrow p|a$ or $p|b$, when $a, b \in R$ [4, p. 212]. While it is true that all primes are irreducible for commutative integral domains with unity, the contrary often is not. But, in a case of a principal ideal domain every irreducible element is also a prime [4, p. 213].

From these [4, p. 213] derives a definition for a *unique factorization domain*:

Definition 2.5.1. A commutative integral domain R with unity is called a unique factorization domain (UFD) if following conditions are met:

1. Every non-unit element of R is a finite product of irreducible factors
2. Every irreducible element is a prime

There are three important theorems about UFDs that function as the basis for the algorithms later.

Theorem 2.5.1. *If R is a UFD, then the factorization of any element in R is a finite product of irreducible factors, and that factorization is unique to within order and unit factors.* [4, p. 214]

Theorem 2.5.2. *If R is a UFD and $a, b \in R$, there exists a greatest common divisor of a and b that is uniquely determined to within arbitrary unit factor.* [4, p. 215]

The greatest common divisor in the latter theorem means that for all elements $a, b \in R$ there exists $d \in R$ such that:

1. $d|a$ and $d|b$
2. If $c|a$ and $c|b$ for any $c \in R$ then $d|c$

Theorem 2.5.3. *Every PID is a UFD while not all UFDs are PIDs. [4, p. 216]*

This last theorem is proved in [4, p. 216].

2.6 Fields

A *field* is a commutative division ring [15, p. 84]. In other words it is a nonempty set F accompanied with operations to multiply and add together elements from it. [15, p. 93] defines a field as follows.

Definition 2.6.1. A field F is a commutative ring, such that $0 \neq 1$ and the multiplicative monoid of non-zero elements of F is a group.

Requirement for multiplicative monoid of non-zero elements being a group means that each non-zero element of F must have an inverse. Another direct restriction is that a field must contain at minimum two elements, namely 0 and 1, where 0 is the additive and 1 multiplicative neutral element.

Some such fields would be the systems of rational numbers $\mathbb{Q}$ and real numbers $\mathbb{R}$. They both are examples of *infinite fields* as the size of their respective sets are infinite. However, when the size of the set F is finite the field is called a *finite field*. An example of a finite field would be the set of residue classes $\mathbb{Z}_p$ with p being a prime and addition and multiplication defined as in section 2.2. The proof of this being a proper finite field is in [4, p. 38].

On the other hand, if we consider the set of residue classes $\mathbb{Z}_4 = \{\overline{0}, \overline{1}, \overline{2}, \overline{3}\}$, we quickly notice that $2 \in \mathbb{Z}_4$ does not have an inverse, and hence, $\mathbb{Z}_4$ is not a field.

It was already established in section 2.3 that the characteristic of a ring $\mathbb{Z}_n$, $n \in \mathbb{N}$ is n. Therefore characteristic of the field $\mathbb{Z}_p$, with p being prime, is p. More generally, for fields we have theorem [4, p. 170]:

Theorem 2.6.1. *Let F be a field. The characteristic of F is either 0 or a prime number p.*

Proof: Let $n \neq 0$ be the characteristic of F, so $ne = 0$, where e is the unit element of F and multiplication by n is defined as applying addition operation n

times. Assume n is a composite number and hence can be written as $n = n_1 n_2$. Now $ne = (n_1 n_2)e = (n_1 e)(n_2 e) = 0$, and because F is a field, either $n_1 e = 0$ or $n_2 e = 0$ (if $n_1 e = 0$, then $n_1 a = n_1 ea = 0$ for all $a \in F$). But, because both $n_1 < n$ and $n_2 < n$, characteristic of F cannot be n. $\square$

A field F is called *perfect* if $F^p = F$, where F^p denotes the field of all elements x^p, $x \in F$ [15, p. 252]. Also, all fields with characteristic zero are perfect [15, p. 252]. By applying Fermat's little theorem, $a^p \equiv a \mod p$, it is easy to see that sets of residue classes of a prime are all perfect fields.

2.7   The ring of polynomials

Let R be a ring and consider all polynomials of the form

$$a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n, \quad a_i \in R$$

Here a_i are called *coefficients* of the polynomial. In this polynomial n is the largest integer for which $a_i \neq 0$. As such a_n is called *the leading coefficient* and n *the degree* of the polynomial. The degree of a zero polynomial is defined to be $-\infty$ [4]. When the leading coefficient is 1, the polynomial is said to be *monic*. A part of a polynomial $a_i x^i$ is called a *term*. These polynomials may be added and multiplied together to form a ring in the following manner [4, p. 165]. Let

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_m x^m,$$
$$g(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^n.$$

Now,

$$f(x) + g(x) = (a_0 + b_0) + (a_1 + b_1)x + (a_2 + b_2)x^2 + \ldots,$$
$$f(x)g(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_{m+n} x^{m+n},$$

where

$$c_i = \sum_{j+k=i} a_j b_k, \quad 0 \leq i \leq m + n.$$

As noted in [4] it is now quite easy to see that the operations conform to the definition of a ring. Such a polynomial ring is called *polynomial ring over R* and it is denoted by $R[x]$, where x is called the *indeterminate* or *variable*. When a polynomial has only one indeterminate it is called *univariate*. Although a polynomial may have multiple

indeterminates, those are not of any particular interest for this thesis. Polynomial ring over R is always infinite even if the underlying ring R is finite.

A ring of polynomials may also be an integral domain. For example, a ring of polynomials over $\mathbb{Z}_n$ is an integral domain if n is a prime. For a non-prime n it is easy to show that zero divisors do exist. Take $\mathbb{Z}_8[x]$ and two polynomials $f(x) = 2x$ and $g(x) = 4x \in \mathbb{Z}_n[x]$. Clearly, $f(x)g(x) = 2x * 4x = 0$, because $2 * 4 \equiv 0 \mod 8$. But, if n is a prime, $\mathbb{Z}_n[x]$ is an integral domain. Proof: Let

$$f(x) = \sum_{i=0}^{n} a_i x^i, \quad g(x) = \sum_{i=0}^{m} b_i x^i$$

be non-zero polynomials with leading coefficients $a^n \neq 0$ and $b^m \neq 0$. The result of multiplication of $f(x)$ and $g(x)$ will then have a leading coefficient $a^n b^m$ and $a^n b^m \not\equiv 0 \mod n$, because $a < n$, $b < n$ and n is a prime, and hence, result cannot be a zero polynomial and $\mathbb{Z}_n[x]$ is an integral domain with no zero divisors.

An important theorem regarding polynomial rings is as follows [15, p. 221]:

Theorem 2.7.1. *Let R be a unique factorization domain. Then the polynomial ring $R[x]$ over R is also a unique factorization domain.*

2.8 Polynomials

This section will introduce some basic properties of polynomials. First, there is a brief discussion what factorization of a polynomial over a ring means. After which Euclidean division for integers is explained and further expanded to work for specific polynomial rings. And lastly, the concept of derivative is redefined for polynomials over rings.

2.8.1 Factorization

In section 2.5 irreducibility of an element of a ring was defined to mean that such an element is only divisible by units or associates. This definition translates to polynomials over a ring R so that a polynomial $f(x) \in R[x]$ is irreducible if $deg(f) > 0$ and $f(x)$ cannot be written as a product

$$f(x) = g(x)h(x)$$

where $g, h \in R[x]$, $deg(g) > 0$ and $deg(h) > 0$. [15, p. 175]

Factorization of a polynomial $f(x)$ of a polynomial ring $R[x]$ means finding its decomposition to powers of irreducible polynomials $a_i \in R[x]$ such that

$$f(x) = \prod_i a_i^{e_i},$$

where e_i is the power of an irreducible polynomial a_i. These a_i are also called *factors* of f.

This decomposition is not possible for all polynomial rings, but, if $R[x]$ is a UFD, each element $f \in R[x]$, $f \neq 0$ can be expressed as a unique product of irreducible factors in the form shown earlier. However, if R contains multiple units, the factorization of f is only unique up to unit factors. Polynomial f is called primitive if *gcd* of all its coefficients is a unit [4, p. 221].

As an example, consider field $\mathbb{Z}_2$ and $f(x) \in \mathbb{Z}_2$ such that

$$\begin{aligned} f(x) &= x^5 + x^4 + x^3 + 1 \\ &= (x^2 + 1)^2(x^3 + x^2 + 1). \end{aligned}$$

Here $f(x)$ clearly is not irreducible, but $(x^2 + 1)$ and $(x^3 + x^2 + 1)$ both are. Hence, they form the factorization of $f(x)$.

2.8.2 Euclidean division

The usual and well-known Euclidean division for integers is the following [7, p. 27].

Theorem 2.8.1. *Given integers $a, b > 0$, there exists unique integers $q > 0$ and $0 \leq r < a$ such that $a = bq + r$.*

This famous theorem may easily be translated to a similar one for univariate polynomials over a commutative ring. In [15, p. 173] this is expressed in the form of the following theorem:

Theorem 2.8.2. *Let R be a commutative ring, let $f, g \in R[x]$ be univariate polynomials of degrees ≥ 0 and assume the leading coefficient of g is a unit in R. Then there exist unique polynomials $q, r \in R[x]$ such that $f = gq + r$ and $deg(r) < deg(g)$.*

Proof: To prove that this theorem holds, let

$$f(x) = a_0 + a_1 x + \ldots a_n x^n,$$
$$g(x) = b_0 + b_1 x + \ldots b_m x^m,$$

where $n = deg(f)$, $m = deg(g)$, both $a_n, b_m \neq 0$ and b_m is a unit in R. It is important that b_m is a unit as it guarantees existence of inverse for it even though inverses for other elements do not necessarily exist in R. Next, according to [15, p. 174], induction on n may be used to construct the proof.

As the basis of the induction if $n = 0$ and $deg(g) > deg(f)$, we let $q = 0$ and $r = f$. And further, if $deg(f) = deg(g) = 0$, we let $r = 0$ and $q = a_n b_m^{-1}$.

Next, we assume the theorem is proved for polynomials with degree $< n$. Also, we may assume that $deg(g) \leq deg(f)$, because if this is not true, we just let $q = 0$ and $r = f$. Now we can write

$$f(x) = a_n b_m^{-1} x^{n-m} g(x) + f_1(x),$$

where $deg(f_1) < n$. By using induction, we can find q_1, r and write f as

$$f(x) = a_n b_m^{-1} x^{n-m} g(x) + q_1(x) g(x) + r(x)$$

with $deg(r) < deg(g)$. And finally, to finish the proof we define

$$q(x) = a_n b_m^{-1} x^{n-m} + q_1(x).$$

The previous proves only the existence of q and r, but makes no claim whether they are unique or not. To prove uniqueness [15, p. 174] starts by assuming there exists two instances of q and r such that

$$f = q_1 g + r_1 = q_2 g + r_2,$$

where $deg(r_1) < deg(g)$ and $deg(r_2) < deg(g)$. After reformatting we get

$$(q_1 - q_2) g = r_2 - r_1.$$

Now, the leading coefficient of g was assumed to be a unit in R, so, we can say that

$$deg((q_1 - q_2) g) = deg(q_1 - q_2) + deg(g).$$

But, we know that $deg(r_2 - r_1) < deg(g)$, and that can be only if $deg(q_1 - q_2) = 0$,

which means that $q_1 = q_2$ and therefore $r_1 = r_2$. $\square$

2.8.3   Formal derivative

The usual method for derivation of polynomials does not work on polynomials over rings or fields. Mainly, because the concept of limits does not generalize in such a way that it would be meaningful for such polynomials. Hence, the need for *formal derivative*, which is basically ordinary derivative but defined just for polynomials over commutative rings and without limits. So, let R be a ring and $f(x) \in R[x]$ such that

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n.$$

Formal derivative of f is as follows [15, p. 178]:

$$f'(x) = \frac{d}{dx} f(x) = a_1 + 2a_2 x + \ldots + n a_n x^{n-1}.$$

Here multiplication $m a_i$ should be considered as summing a_i m times.

From this definition it is easy to derive following properties of the formal derivative:

$$(af + bg)' = af' + bg', \qquad a, b \in R \tag{2.1}$$
$$(fg)' = f'g + g'f \tag{2.2}$$

The first property is usually known as *linearity* and the second is called the *product rule*. [15, p. 307]

2.9   Chinese remainder theorem

Chinese remainder theorem is useful when computing with large integers. It allows replacing a computation with known bound in size with similar computations but with smaller integers.

For any two coprime integers $x, y > 1$ and any other integers a and b the theorem guarantees existence of integer N such that N is the solution to following two congruences [7, p. 256]

$$N \equiv a \mod x$$
$$N \equiv b \mod y.$$

In addition to just existence, the theorem also states that the N is unique modulo xy [7, p. 257].

This theorem generalizes to any number of coprime integers $m_1, m_2, \ldots, m_n$, where $m_i > 1$, called the *moduli*, and any other integers $a_1, a_2, \ldots, a_n$. The theorem then states existence of a solution x to the following set of congruences [7, p. 258]

$$x \equiv a_1 \mod m_1$$
$$x \equiv a_2 \mod m_2$$
$$\vdots$$
$$x \equiv a_n \mod m_n.$$

While this theorem does not generalize directly to all PIDs it does so for univariate polynomials over a field F [7, p. 364].

Theorem 2.9.1. *Let F be a field, $a_1(x), a_2(x), \ldots, a_n(x) \in F[x]$ be arbitrary polynomials and $m_1(x), m_2(x), \ldots, m_n(x) \in F[x]$ be pairwise coprime polynomials. Then, there exists a polynomial $f(x) \in F[x]$ such that*

$$f(x) \equiv a_1(x) \mod m_1(x)$$
$$f(x) \equiv a_2(x) \mod m_2(x)$$
$$\vdots$$
$$f(x) \equiv a_n(x) \mod m_n(x).$$

And if $f_1(x)$ and $f_2(x)$ are both solutions, then

$$f_1(x) \equiv f_2(x) \mod \prod_{i=1}^{n} m_i(x).$$

Childs goes on to prove the theorem in [7, p. 365] by constructing a solution. Furthermore, with his constructed solution, he also proves that there exists a unique solution to the previous group of congruences with a degree less than the degree of $\prod_i m_i(x)$ [7, p. 365].

3.   CHOSEN ALGORITHMS

This chapter will explain all major algorithms that the solution to the Nintendo challenge [1] rely upon. These algorithms can be roughly divided into two categories where first category contains algorithms that are meant for implementing operations fast on a PC and the second which actually factorizes polynomials.

The first category includes algorithms for fast multiplication 3.1.1, division 3.1.2 and finding the *gcd* of polynomials 3.1.3.

The second category of algorithms are the main algorithms meant for factorization of polynomials over finite fields. There are several approaches one can take to factorize such a polynomial and the one chosen for this thesis is one of the oldest. It is a two-phased approach where first a polynomial is decomposed to its square-free factors in section 3.2 and then those square-free factors are further factorized with Berlekamp's algorithm in 3.3.

Lastly, in section 3.4 improved approaches to the factorization problem are briefly discussed and their main differences to the chosen one analyzed.

3.1   Fast polynomial operations

This section presents algorithms that aim at speeding up computations with polynomials. Two basic computations for polynomials, multiplication and division, are given algorithms that are faster than naive ones. In addition, a *gcd* algorithm based on the division algorithm described here is presented.

3.1.1   Fast carry-less multiplication

Being able to multiply polynomials fast in a binary field is important for reducing overall complexity of the factorization solution. If this multiplication is done with a naive algorithm based on the definition in 2.7, it is easy to see that the multiplication takes n^2 multiplication operations and $(n-1)^2$ addition operations, where n is the degree of the polynomial. This yields overall quadratic complexity for multiplication.

Another aspect of multiplication of polynomials over field $\mathbb{Z}_2$ is that the operation translates directly to a carry-less multiplication on computer systems. There has been Carry-less Multiplication (CLMUL) instruction set for x86 microprocessors from 2008 which was originally proposed by Intel, and which was later made available in 2010 first in Intel Westmere processors. At the time of writing this thesis there is no simple way of enabling this instruction set for C++, though the algorithm is created with this option in mind.

There are interesting approaches to multiplication available that improve upon the naive algorithm. One of the early ones is from 1963 by Karatsuba [13] who was able to reduce the asymptotic complexity to $O(7n^{1.58} + n)$. This Karatsuba's algorithm has since been improved upon in 2009 by Bernstein [17, p. 317–336]. He got the asymptotic complexity down to $O(6.5n^{1.58} + n)$. While Karatsuba and Bernstein solved the problem with clever division to smaller multiplications, Schönhage and Strassen took different approach in 1971 and based their algorithm on the Fast Fourier Transform achieving asymptotic complexity of $O(n \log n \log \log n)$ [18]. These are not all, or even latest improvements, but they indicate that carry-less multiplication is one option for optimizing the whole algorithm further, should such optimization be needed.

For this thesis Karatsuba's original algorithm was chosen as it is rather simple and it suits the problem quite well. Karatsuba's algorithm [13, p. 595-596] takes two $2n$-bit polynomials F and G as an input. It splits both polynomials to two parts $F = F_0 + F_1 t^n$ and $G = G_0 + G_1 t^n$. Now, the product FG can be calculated as follows:

$$FG = (F_0 + F_1 t^n)(G_0 + G_1 t^n)$$
$$= (1 + t^n)(F_0 G_0) + t^n(F_0 + F1)(G_0 + G1) + (t^n + t^2 n)F_1 G_1$$

By using this decomposition, the multiplication is divided recursively to smaller multiplications until the resulting product fits a 32bit integer. At this level algorithm reverts back to the usual sifting and XORing.

3.1.2 Fast division of polynomials

The Euclidean division, explained in section 2.8.2, is one the key building blocks for the factorization algorithm presented later. Hence, it is important to have a fast implementation for it. A naive one, using the long division of polynomials, will yield an algorithm with asymptotic complexity of $O(deg(a)deg(b))$, where a and b are the

polynomials, which for all practical purposes is the same as $O(n^2)$, where n is the degree of polynomials.

Now, for integers there exists a fast way of implementing Euclidean division by using Newton's method. Let us recall Newton's method and the Euclidean division problem setting from theorem 2.8.1.

Newton's method is a numerical method for finding a root of a real-valued function $f(x)$. It starts by approximating, or just selecting any starting point x_0 and then computing next approximation by forming a tangent line through point $(x_0, f(x_0))$ and using the point where this tangent intersects x-axis as the next approximation. This iterative step can be expressed as

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

Next, when we are given integers $a, b > 0$, we would like to compute integers $q, r \geq 0$ such that $a = bq + r$ and $r < b$. We observe that q may be computed as $q = \lfloor a/b \rfloor$, and when q is known, we can compute $r = a - bq$. So, to determine the value of q, it suffices to get a close enough approximation of $c = b^{-1}$ and then multiply ac and round it down to the closest integer. This we can achieve by using Newton's method on function $f(x) = x^{-1} - b$. With this the iterative step is as follows:

$$x_{i+1} = x_i - \frac{x_i^{-1} - b}{-x_i^{-2}} = 2x_i - bx_i^2.$$

Now, as it turns out, this Newton's method translates to polynomials over commutative rings with unity too. In their publication Zhengjun Cao and Hanyue Cao [6] improve upon some earlier version of this algorithm and provide all missing steps for implementing it. Details of their work are out of the scope of this thesis, but the resulting algorithm will be introduced next.

The algorithm by Cao and Cao relies on first *reversing* coefficients of a polynomial and then computing its modulo with a large power of variable x, doubling that power with each iterative step of Newton's method. Reversing coefficients of a polynomial $f(x)$ is denoted with $rev(f) = rev_{deg(f)}(f)$ and can be achieved with

simply calculating $x^{deg(f)}f(x^{-1})$ [6]. So, as an example:

$$
\begin{aligned}
f(x) &= a_0 + a_1 x + + a_2 x^2 \ldots + a_n x^n \\
rev(f) &= x^n f(x^{-1}) \\
&= x^n(a_0 + a_1 x^{-1} + a_2 x^{-2} + \ldots + a_n x^{-n}) \\
&= a_0 x^n + a_1^{n-1} + \ldots + a_{n-1} x + a_n.
\end{aligned}
\tag{3.1}
$$

We recall theorem 2.8.2, Euclidean division for polynomial, which states: Let R be a commutative ring with unity and $a, b \in R[x]$ two polynomials with degrees > 0 and b being monic. There exists unique, up to unit factors, polynomials $q, r \in R[x]$ such that $a = bq + r$ and $deg(r) < deg(b)$.

Cao and Cao begin then by transforming $a = bq + r$ equation by substituting x with x^{-1} and by multiplying with x^n, with $n = deg(a)$ and later $m = deg(b)$ [6]. We get

$$
x^n a(x^{-1}) = (x^{n-m} q(x^{-1}))(x^m b(x^{-1})) + x^{n-m+1}(x^{m-1} r(x^{-1}))
$$

$$
\Leftrightarrow
$$

$$
rev_n(a) = rev_{n-m}(q) \cdot rev_m b + x^{n-m+1} rev_{m-1}(r),
$$

which implies

$$
rev_n(a) \equiv rev_{n-m}(q) \cdot rev_m(b) \mod x^{n-m+1}.
$$

Next Cao and Cao point out that because b was said to be monic, then $rev_m(b)$ has a constant coefficient of 1, and hence, $rev_m(b)$ is invertible modulo x^{n-m+1} [6]. For $g \in R[x]$ to be invertible $\mod f$, we have to be able to find a polynomial $h \in R[x]$ such that $gh \equiv 1 \mod f$. So, we get

$$
rev_{n-m}(q) \equiv rev_n(a) \cdot rev_m(b)^{-1} \mod x^{n-m+1},
$$

and finally,

$$
q = rev_{n-m}(rev_{n-m}(q)).
$$

This leaves, according to Cao and Cao [6], a problem of finding $g(x) \in R[x]$ such that $fg \equiv 1 \mod x^l$, when $f(x) \in R[x]$ is given, $f(0) = 1$ and $l \in \mathbb{N}$. Furthermore, they observe that when l is a power of two, the iteration step to solve the problem is $g_{i+1} = 2g_i - fg_i^2$ [6]. This leads to the following result [6]:

Theorem 3.1.1. *Let R be a commutative ring with unity and let $f, g_0, g_1, \ldots, \in$*

$R[x]$, *with* $f(0) = 1$, $g(0) = 1$, *and*

$$g_{i+1} \equiv 2g_i - fg_i^2 \mod x^{2^{i+1}}$$

for all i. *Then* $fg_i \equiv 1 \mod x^{2^i}$ *for all* $i \geq 0$.

When all these are combined, we get the following algorithm [6]:

1. If $deg(a) < deg(b)$ return $q = 0$ and $r = a$
2. Let $m = deg(a) - deg(b)$ and $r = \lceil \lg m + 1 \rceil$
3. Let $f = rev(b)$
4. For $i = 1, \ldots, r$ do
   $g_i = (2g_{i-1} - fg_{i-1}^2) \mod x^{2^i}$
5. Let $s = rev(a)g_r \mod x^{m+1}$
6. Return $q = x^{m-deg(s)}rev(s)$ and $r = a - bq$

This algorithm improves asymptotic complexity to $O(M(n))$, where $M(n)$ is the complexity of multiplication of polynomials of degree n [6].

3.1.3   Greatest common divisor

Theorem 2.5.2 stated the existence of a greatest common divisor for any two elements of a ring that is a UFD. However, as there may be multiple unit elements in such a ring, there may exist multiple greatest common divisors [14, p. 424]. So, if w is a greatest common divisor of u and v, then $a \cdot w$ must also be, when a is a unit.

The algorithm for computing a greatest common divisor, often called *Euclid's algorithm*, may be based on Euclidean division provided in subsection 3.1.2. To see how that happens let R be a ring that is a UFD and $a, b \in R[x]$ be two polynomials. Next, let $a = bq + r$ and $w \in R[x]$ be a polynomial that divides both a and b. Therefore, we can write $a = sw$ and $b = tw$ for some $s, t \in R[x]$. Now, as w divides a, it must also divide r because

$$r = a - bq = sw - twq = (s - tq)w.$$

Also, any $v \in R[x]$ that divides both b and r, where $b = s'v$ and $r = t'v$, must divide a as well because

$$a = bq + r = s'vq + t'v = (s'q + t')v.$$

These two show that any common divisor of a and b is also a common divisor of b and r. The algorithm based on this observation is then easy to produce [14].

Let R be a UFD. A greatest common divisor of $u, v \in R[x]$, $gcd(u, v)$, may be found as follows [14, p. 424]:

1. If $v = 0$, then $gcd(u, v) = u$
2. Else $gcd(u, v) = gcd(v, r)$,

where r is the remainder of Euclidean division of u and v from section 2.8.2.

After computing a greatest common divisor, it is customary to multiply it with the inverse of the leading coefficient which results in a monic polynomial. [14, p. 425]. This way there is always a certain result that can be called the greatest common divisor of u and v, while there may be many.

3.2   Square-free factorization

Let F be a UFD and $f \in F[x]$ a polynomial over F. As presented in section 2.8.1, f can be written as a product of irreducible factors such that

$$f(x) = \prod_i a_i^{e_i}, \qquad a_i \in F[x], e_i > 0.$$

The polynomial f is said to be *square-free* if $e_i \leq 1$ for all i. On the other hand, any factor a_k for which corresponding $e_k > 1$ is said to be a *repeated factor*.

The process of decomposing a polynomial to its square-free factors is called *square-free factorization*. Square-free factorization is much faster to perform than a proper factorization algorithm and is therefore used as a preliminary step to speed up the factorization as a whole.

3.2.1   Characteristic zero

Let F be a unique factorization domain with characteristic 0. Yun's algorithm decomposes a polynomial $f \in F[x]$ to $f = a_1^1 a_2^2 \ldots a_k^k$, where $a_k \neq 0$, each a_i is square-free and $gcd(a_i, a_j) = 1, i \neq j \leq k$ [21]. It does this by computing $gcd(f, df/dx)$ and using a clever trick:

Theorem 3.2.1. *If* $f = a_1^1 a_2^2 \ldots a_k^k$ *is primitive in x then* $gcd(f, \frac{df}{dx}) = a_2^1 a_3^2 \ldots a_k^{k-1}$ *[21]*

Proof: Let $a_0 = a_2^1 a_3^2 \ldots a_k^{k-1}$. First we establish using chain rule that

$$\frac{d}{dx} a_i^i = i a_i^{i-1} \frac{da_i}{dx}.$$

Next we calculate the formal derivative of f by applying product rule:

$$\frac{df}{dx} = (\frac{da_1}{dx}a_2^2 a_3^3 \ldots a_k^k) + (2\frac{da_2}{dx}a_1 a_2 a_3^3 \ldots a_k^k) + (3\frac{da_3}{dx}a_1 a_2^2 a_3^2 \ldots a_k^k) + \ldots$$

$$= (\prod_{i=2}^{k} a_i^{i-1})((\frac{da_1}{dx}a_2 a_3 \ldots a_k) + (2\frac{da_2}{dx}a_1 a_3 \ldots a_k) + (3\frac{da_3}{dx}a_1 a_2 \ldots a_k) + \ldots$$

$$= (\prod_{i=2}^{k} a_i^{i-1})(\sum_{i=1}^{k} (i\frac{da_i}{dx}\prod_{j\neq i}^{k} a_j))$$

$$= a_0(\sum_{i=1}^{k} (i\frac{da_i}{dx}\prod_{j\neq i}^{k} a_j))$$

Now, a_0 clearly divides both f and $\frac{df}{dx}$ and $f/a_0 = \prod_{i=1}^{k} a_i$. Next we let s be a divisor of any a_j and notice that s does not divide any $a_i, i \neq j$, because $gcd(a_i, a_j) = 1$ for all $i \neq j$. And, because a_j is square-free and primitive with respect to x, $gcd(a_j, \frac{da_j}{dx}) = 1$ [21], meaning that s does not divide $\frac{da_j}{dx}$. From this it follows that s divides all other terms in the remaining sum but one, and hence, s does not divide the sum. Therefore $gcd(f, \frac{df}{dx}) = a_0$. $\square$

Based on this trick the Yun's algorithm is as follows:

```
1 a[0] = gcd(f, df/dx)
2 c = f/a[0]
3 d = (df/dx)/a[0] - dc/dx
4
5 For i from 1 until c == 1:
6     a[i] = gcd(c, d)
7     c = c/a[i]
8     d = d/a[i] - dc/dx
```

Program 3.1 *Yun's algorithm [21].*

Yun uses two facts to prove correctness of the algorithm, both which are corollaries of theorem 3.2.1.

Corollary 3.2.1.1. *Let* $f' = df/dx$, $a_0 = gcd(f, f')$, *then*
$f'/a_0 - (f/a_0)' = a_1(a_2' a_3 \ldots a_k + 2a_2 a_3' \ldots a_k + \ldots + (k-1)a_2 a_3 \ldots a_{k-1}a_k')$ *[21]*

Proof: By derivation we get following results:

$$f' = a_0(\sum_{i=1}^{k} (ia'_i \prod_{j\neq i}^{k} a_j))$$

$$f'/a_0 = \sum_{i=1}^{k} (ia'_i \prod_{j\neq i}^{k} a_j)$$

$$f/a_0 = a_1 a_2 a_3 \ldots a_k$$

$$(f/a_0)' = \sum_{i=1}^{k} (a'_i \prod_{j\neq i}^{k} a_j).$$

So, we can calculate

$$f'/a_0 - (f/a_0)' = a'_2 a_1 a_3 \ldots a_k + a'_3 a_1 a_2 \ldots a_k + \ldots + a'_k a_1 a_2 \ldots a_{k-1}$$

$$= a_1(a'_2 a_3 \ldots a_k + 2a'_3 a_2 \ldots a_k + \ldots + (k-1)a'_k a_2 \ldots a_{k-1}). \quad \square$$

Corollary 3.2.1.2. $gcd(f/a_0, (f'/a_0) - (f/a_0)') = a_1$ *[21]*

Proof: From earlier we already have $f/a_0 = a_1 a_2 \ldots a_k$ and $(f'/a_0) - (f/a_0)' = a_1 Q$, where $Q = a'_2 a_3 a_4 \ldots a_k + 2a'_3 a_2 a_4 \ldots a_k + \ldots + (k-1)a'_k a_2 a_3 \ldots a_{k-1}$. It is clear that a_1 divides both, but to fully prove the corollary, we need to still show that $gcd(a_2 a_3 \ldots a_k, Q) = 1$ [21]. In order to show this, let $f^* = a_2 a_3^2 \ldots a_k^{k-1}$. But, now theorem 3.2.1 states that $a_0^* = gcd(f^*, f^{*'}) = a_3 a_4^2 \ldots a_k^{k-2}$, where all factors are relatively prime, hence, $gcd(f^*/a_0^*, f^{*'}/a_0^*) = 1$. And last, we just calculate:

$$f^*/a_0^* = a_2 a_3 \ldots a_k$$
$$f^{*'}/a_0^* = (a'_2 a_3^2 a_4^3 \ldots a_k^{k-1} + 2a'_3 a_2 a_3 a_4^3 \ldots a_k^{k-1} + \ldots + (k-1)a'_k a_2 a_3^2 a_4^3 \ldots a_k^{k-2})/a_0^*$$
$$= a'_2 a_3 a_4 \ldots a_k + 2a'_3 a_2 a_4 \ldots a_k + \ldots + (k-1)a'_k a_2 a_3 \ldots a_{k-1}$$
$$= Q$$

This proves both the corollary and the correctness of the algorithm. $\quad \square$

The asymptotic complexity of this algorithm is mostly affected by gcd computations. If the asymptotic complexity of a gcd computation for a polynomial $f \in F[x]$, with $n = deg(f)$, is denoted with $M(n)$, the asymptotic cost of Yun's algorithm is $O(M(n) \log n)$ [20].

3.2.2 Positive characteristic

The Yun's algorithm works well over any universal factorization domain with characteristic zero, but as such does not work when characteristic is positive. This is a problem because a fields of integers modulo prime p have characteristic of p, as stated in section 2.3, and those are important for this thesis.

The problem that arises from the positive characteristic is that the derivative may vanish as all terms of the polynomial $f \in F_p[x]$ which are of the form ax^{ip} have derivatives $ipax^{ip-1} \equiv 0 \mod p$. Also, this means that the derivative of f being zero does not have the usual meaning of a zero derivative from calculus.

Theorem 3.2.2. *Let F_p be a field with a positive characteristic p and $f(x) \in F_p[x]$ with $f'(x) = 0$. There exists polynomial $g(x) \in F_p[x]$ such that $f(x) = g(x^p) = (g(x))^p$ [9].*

Proof: To prove this peculiar property Knuth [14, p. 440] considers two polynomials $u, v \in F_p[x]$ for which

$$(u(x) + v(x))^p = u(x)^p + \binom{p}{1}u(x)^{p-1}v(x) + \ldots + \binom{p}{p-1}u(x)v(x)^{p-1} + v(x)^p$$
$$= u(x)^p + v(x)^p.$$

The last equality is true because a prime number p divides all binomial coefficients $\binom{p}{1}, \ldots, \binom{p}{p-1}$, and from theorem 2.6.1 we know a field can only have characteristic of 0 or a prime. Furthermore, by Fermat's little theorem we know that $a^p = a \mod p$, and hence, if $u(x) = u_0 + u_1 x + \ldots + u_m x^m$, then

$$u(x)^p = (u_0)^p + (u_1 x)^p + \ldots + (u_m x^m)^p$$
$$= u_0 + u_1 x^p + \ldots + u_m x^{mp} = u(x^p). \quad \square$$

From this we get two important results: Firstly, if the derivative of $f \in F_p[x]$ is zero, f is a perfect pth power of some polynomial $g \in F_p[x]$, and secondly, taking pth root of $g \in F_p[x]$, when g is known to be a perfect pth power is achieved by dividing each exponent in g by p.

Formally, when a field has a positive characteristic, theorem 3.2.1 does not hold. This is easy to see if we have a field F_p, $f \in F_p[x]$ and $f(x) = x^p - u$. Here f is clearly irreducible when u does not have pth root in the field, and therefore it must

also be square-free, but $gcd(f, f') = f$. This means that in positive characteristic a polynomial is not necessarily relatively prime with its derivative [9].

To improve upon definition of irreducibility, Gianni introduces a new one: a *separable* polynomial. Polynomial $f \in F_p[x]$ is said to be separable if and only if $gcd(f, f') = 1$ [9]. This makes all separable polynomials square-free, but converse only works for fields with characteristic zero.

To construct an algorithm for square-free factorization of polynomials over fields with positive characteristic Gianni and Trager use Musser's algorithm [9], which is one of three algorithms explained by Yun in [21]. Like all three algorithms, Musser's algorithm is also based on the same property of derivative as Yun's, and it also works only for characteristic zero. However, this algorithm can be improved upon. Musser's algorithm is as follows:

```
1 c[1] = gcd(f, df/dx)
2 b[1] = f/c[1]
3
4 For i from 1 until b == 1:
5     b[i + 1] = gcd(c[i], b[i])
6     c[i + 1] = c[i]/b[i + 1]
7     P[i] = b[i]/b[i + 1]
```

Program 3.2 *Musser's algorithm [9].*

As an input this algorithm takes any $f \in F_p[x]$, where $char(F) = 0$ and it outputs $P_1, P_2, \ldots, P_k \in F_p[x]$ such that $f = \prod P_i^i$. First two lines of the algorithm set up values

$$c_1 = P_2 P_3^2 \ldots P_k^{k-1},$$
$$b_1 = P_1 P_2 \ldots P_k.$$

Within the loop following invariants hold [9]:

$$c_i = P_{i+1} P_{i+2}^2 \ldots P_k^{k-i},$$
$$b_i = P_i P_{i+1} \ldots P_k,$$
$$b_{i+1} = P_{i+1} P_{i+2} \ldots P_k,$$
$$c_{i+1} = P_{i+2} P_{i+3}^2 \ldots P_k^{k-i-1}.$$

To properly incorporate the idea of a portion of polynomials vanishing by derivation, Gianni made the following change to the characterization of the factorization to

square-free factors [9]:

Lemma 3.2.1. *Let F_p be a field with a positive characteristic p and $f \in F_p[x]$. There exists unique, up to unit factors, polynomials $P_i, Q \in F_p[x]$ such that P_i are separable with all i, $gcd(Q, P_i) = gcd(P_i, P_j) = 1$ when $i \neq j$, $dQ/dx = 0$, $i \equiv 0$ mod $p \Rightarrow P_i = 1$ and*

$$f = Q \prod_{i=1}^{k} P_i^i.$$

Proof: Gianni proves this by construction [9]. First we consider irreducible factorization $f = \prod_i f_i^{e_i}$ and sets $S_i = \{j | (e_j = i) \wedge (p \nmid i) \wedge (f_j \neq 0)\}$ and $T = \{j | (p \mid e_j) \vee (f_j = 0)\}$. Now, all we need to do is to set

$$P_i = \prod_{S_i} f_i \qquad Q = \prod_T f_i^{e_i}. \quad \square$$

Next Gianni improves upon Yun's theorem 3.2.1 with following change [9].

Theorem 3.2.3. *Let F_p be a field with a positive characteristic p, $f \in F_p[x]$ and $f = Q \prod_{i=1}^{k} P_i^i$ be the decomposition of f. Then $gcd(f, f') = Q \prod_{i=1}^{k} P_i^{i-1}$.*

Proof: The proof for this theorem follows the steps of Yun's proof. First, we get from the formal derivation that $f' = Q \prod_{i=1}^{k} P_i^{i-1} (\sum_{j=1}^{k} j P_j^i \prod_{i=1, i \neq j}^{k} P_i)$ [9]. Next, by invoking properties of 3.2.1, namely $Q' = 0$, P_i is separable, $i \equiv 0 \Rightarrow P_i = 1$ and Q and P_i are pairwise prime, Gianni obtains the result that $gcd(f, f') = Q \prod_{i=1}^{k} P_i^{i-1}$ [9]. $\quad \square$

Next Gianni notices that when using decomposition $f = Q \prod_{i=1}^{k} P_i^i$ and running Musser's algorithm, later known as *squareFree*, the invariants of that algorithm change to [9]

$$c_1 = Q P_2 P_3^2 \ldots P_k^{k-1},$$
$$b_1 = P_1 P_2 \ldots P_k.$$

and inside the loop to

$$c_i = QP_{i+1}P_{i+2}^2 \ldots P_k^{k-i},$$
$$b_i = P_iP_{i+1} \ldots P_k,$$
$$b_{i+1} = P_{i+1}P_{i+2} \ldots P_k,$$
$$c_{i+1} = QP_{i+2}P_{i+3}^2 \ldots P_k^{k-i-1}.$$

After the algorithm stops, all that is left in $c_k = Q$.

With the previous insight and the theorem 3.2.2 we can finally present an algorithm for square-free factorization of a polynomial $f \in F_p$, when F_p is a perfect field [9].

Step 1. $(q, f_1, f_2, \ldots, f_k) = squareFree(f)$. Then, if $deg(q) = 0$, return.
Step 2. Compute pth root of q to variable h.
Step 3. Recursively decompose h and then merge results with f_i.

3.3 Berlekamp's algorithm

Berlekamp's algorithm is for factorization of polynomials over finite fields to their irreducible factors. It was invented by Elwyn Berlekamp in 1967 [2] and was the dominant algorithm before Cantor and Zassenhaus [5] introduced their algorithm in 1981. Even today Berlekamp's algorithm handles well factorization over fields that are small.

Berlekamp's algorithm was chosen for this thesis because of it being historically important and fast enough for small fields. Improvements to it and more sophisticated algorithms will be discussed from a theoretical point of view later.

While Berlekamp's algorithm works for all polynomials, it is beneficial to use it on square-free polynomials. The first benefit comes from square-free factorization being much faster and the second from overall less complex algorithm. There are special cases to take care of when repeated factors are present in the input, and all those can be avoided by having square-free factorization as a preliminary step.

Berlekamp's algorithm factors polynomial $f \in F_p[x]$ to its irreducible factors. So, let

$$f(x) = \sum_{k=0}^{m} a_k x^k, \ a_i \in F_p$$

The algorithm will produce such irreducible polynomials e_i that:

$$f(x) = \prod_{i=1}^{n} P_i, \ P_i \in F_p[x]$$

The algorithm works by first generating something called the Q *matrix* [2]. This Q matrix represents a set of linear equations that must be partly solved in order to get the information required for factorization. Partial solving here means figuring out the null space of $(Q - I)$, where I is the identity matrix, with appropriate column operations. The third step is to use the base vectors of the null space and Euclid's algorithm for greatest common divisor to figure out one factorization for f. And last, if all factors are not yet irreducible; the algorithm uses further *gcd* computations to find the rest.

3.3.1 Building the Q matrix

The Q matrix is $m \times m$, where m is the largest integer for which the coefficient $a_m \neq 0$. The ith row of the matrix represents $x^{p(i-1)}$ reduced modulo $f(x)$ [2]. So, in other terms:

$$x^{pi} \equiv \sum_{k=0}^{m-1} Q_{i+1,k+1} x^k \quad \mod f(x). \tag{3.2}$$

Berlekamp makes an observation that given any other polynomial $g(x) \in F_p[x]$, where $deg(g) < m$ we can calculate the residue of $(g(x))^p \mod f(x)$ by simply multiplying row vector presentation of $g(x)$ by matrix Q [2]. In other words, when $g(x) = \sum_{k=0}^{m-1} g_i x^i$, row vector representation of $g(x)$ is $[g_0, g_1, \ldots, g_{m-1}]$. And from the composition of the matrix Q follows that:

$$(g(x))^p = g(x^p) = \sum_{i=0}^{m-1} g_i x^{pi} = \sum_{i=0}^{m-1} \left(\sum_{k=0}^{m-1} g_i Q_{i+1,k+1} x^k \right)$$
$$= \sum_{i=0}^{m-1} \left(\sum_{k=0}^{m-1} g_i Q_{i+1,k+1} \right) x^k. \ [2]$$

Similarly the residue of $(g(x))^p - g(x) \mod f(x)$ may be computed by multiplying row vector of $g(x)$ by the matrix $(Q - I)$, where I is the identity matrix over $F_p[x]$. [2]

3.3.2   Base vectors of the null space of the (Q - I) matrix

Next important part of Berlekamp's algorithm is to calculate the *null space*, also known as the *kernel* of the matrix $(Q - I)$. In linear algebra the null space is the set of all vectors $\vec{v}$ such that $\boldsymbol{A}\vec{v} = \vec{0}$, where $\boldsymbol{A}$ is the matrix for which the null space is defined. In other terms, if matrix $\boldsymbol{A}$ is thought as a linear transform, the null space of $\boldsymbol{A}$ is the sub-space which is transformed to the zero vector.

The base vectors defining the null space may be found by applying appropriate column operations on the matrix. From algorithmic point of view this means applying Gaussian elimination on the matrix until rows with just zeros are found.

Now, the important property of the null space of $(Q - I)$ is that every polynomial $g(x)$ that belongs to the null space must satisfy equation

$$(g(x))^p - g(x) \equiv 0 \mod f(x) \text{ [2]}. \tag{3.3}$$

And, each $g(x)$ that satisfy the equation, is also a row vector in the null space of the matrix $(Q - I)$ [2].

3.3.3   Factorization

The last step for the algorithm is to use base vectors $g(x)$ of the null space of the matrix $(Q - I)$ and Euclid's algorithm for finding the greatest common divisor to have a factorization of the polynomial $f(x)$. The factorization is as follows:

$$f(x) = \prod_{s \in F_p} gcd(f(x), g(x) - s) \text{ [2]}. \tag{3.4}$$

From equation 3.3 we know that $f(x)$ divides $(g(x))^p - g(x)$, which again can be written as $\prod_{s \in F_p} (g(x) - s)$. Hence, $f(x)$ also divides $\prod_{s \in F_p} gcd(f(x), g(x) - s)$[2]. But, then again, $gcd(f(x), g(x) - s)$ divides $f(x)$. Now, for all $s \neq t$ and $s, t \in F_p$ it holds that $g(x) - s$ and $g(x) - t$ are relatively prime as are $gcd(f(x), g(x) - s)$ and $gcd(f(x), g(x) - t)$. From there it follows that

$$\prod_{s \in F_p} gcd(f(x), g(x) - s)$$

divides $f(x)$. Now, when both polynomials are assumed to be monic [2], and they both divide the other, they must be equal.

3.3.4   Properties of the Berlekamp's algorithm

Due to the way how the last step of Berlekamp's algorithm creates factorization of polynomial f, it is obvious that it does not always find all, or even irreducible, factors immediately. To see why this happens it is enough to notice that the factorization contains at most $|F_p|$ factors and if the number of irreducible factors is larger than that, not all factors are found and at least one of the resulting factors is further reducible. In order to address this issue either irreducibility test must be created or there needs to be a way to know how many irreducible factors there should be. Berlekamp solves it with the latter approach.

If $f(x) = \prod_i (p_i(x))^{e_i}$, where $p_i(x) \in F_p[x]$ and each $p_i(x)$ is irreducible over $F_p[x]$, then $f(x)$ divides $\prod_{s \in F_p} (g(x) - s)$ if each $(p_i(x))^{e_i}$ divides $g(x) - s_i$ for some $s \in F_p$ [2]. On the other hand, for any set of scalars $s_1, s_2, \ldots, s_n \in F_p$ the Chinese remainder theorem 2.9.1 guarantees that there exists a unique $g(x) \mod f(x)$ such that $g(x) \equiv s_i \mod (p_i(x))^{e_i}$ for all i [2]. Here Berlekamp makes an observation that as there are p^n solutions to equation $(g(x))^p - g(x) \equiv 0 \mod f(x)$, as proven earlier, it follows that the number of irreducible factors of $f(x)$ is equal to the the dimension of the null space of $(Q - I)$ or in other words $rank(Q - I)$.

If the *gcd* computation with the first base vector of the null space $g_1(x)$ did not yield as many unique factors as the rank of the matrix $(Q - I)$, then further *gcd* computations are required. Next *gcd* between each known factor and $(g_2(x) - s)$ is computed, where $g_2(x)$ is the next base vector of the null space [2]. By continuing *gcd* computations for each base vector of the null space all factors are eventually found [2].

Berlekamp himself does not make any estimations of the asymptotic complexity of his algorithm in [2], but subsequent researchers have made. Kaltofen and Shoup compare different algorithms for factorization in their publication [12]. Their proposition is that Berlekamp's algorithm can be implemented in $O(n^\omega + n^{1+o(1)} \log p)$ operations in F_p, where ω is the exponent of matrix multiplication meaning that two $n \times n$ matrices can be multiplied using $O(n^\omega)$ arithmetic operations.

3.4   Other factorization algorithms

There have been many improvements and asymptotically faster algorithms developed for factorization of polynomials over finite fields after Berlekamp first brought the problem to the realm of feasible ones in 1967 [2]. His algorithm works well for fields with small size, but soon becomes too complex when the size of the field is

large. Berlekamp improved upon his own work in 1970 when he published his paper on how to factorize polynomials over large fields.

When Yun introduced his algorithm for finding square-free factors of polynomials over finite fields in 1974 [21], he effectively introduced a preliminary step for Berlekamp's algorithm as Yun's algorithm is much faster than Berlekamp's and basically comes with the complexity of a one *gcd* computation. After Yun's work there was another improvement in a form of introducing yet another step for factorization. Algorithms from Cantor & Zassenhaus (1981), Ben-Or (1981), von zur Gathen & Soup (1992) and Kaltofen & Shoup (1995) all proceed in three steps [20]:

1. Square-free factorization.   This is still a preliminary step for all these algorithms and it is handled with some variation of Yun's algorithm.

2. Distinct degree factorization.   When given a monic square-free polynomial $f \in F_p[x]$, compute its unique decomposition

$$f = \prod_{i=1}^{deg(f)} h_i$$

into monic polynomials $h_1, h_2 \ldots, h_{deg(f)} \in F_p[x]$, where each h_i has only irreducible factors of degree i. Here, h_i are also called *equal-degree polynomials*. [20]

3. Equal degree factorization.   Given integers $r, d \in \mathbb{N}$, $r \geq 0$ and a square-free equal-degree polynomial $f \in F_p[x]$ of order d and degree $n = rd$, compute its r irreducible factors. [20]

In 1994 Niederreiter presented his algorithm which, like Berlekamp's algorithm, sets up a linear system of equations, solves it using Gaussian elimination and extracts the solution from the base vectors of solved system with greatest common divisor computations [16]. In 1999 Roelse showed in his publication [16] that Niederreiter's algorithm parallelizes to multiple processors and was able to, by his own words, make a new "world record" in factorization.

All-in-all, the size of the field, the number of variables and the degree of polynomials influence hugely which algorithm and method should be used. The problem in this thesis is about univariate polynomials in a small field with rather small maximum degree, and hence, Berlekamp's algorithm does not fare significantly worse than

others.

4.   SOLVING THE NINTENDO CHALLENGE

This chapter provides the full solution to the Nintendo challenge [1]. First, section 4.1 gives full presentation of the Nintendo challenge and then section 4.2 expresses it in such a way that the theory given in chapter 2 may be used to solve it. Then, section 4.3 explains why all theory explained applies for the problem. After that, section 4.4 provides comprehensive insight into how polynomials are represented and handled in a target computer system. Sections 4.5 and 4.6 explain the implementation of two main algorithms, square-free factorization and Berlekamp's algorithm respectively. The last section, 4.7 briefly discusses how the factors from the algorithms can be combined to a solution to the problem.

All algorithms and pseudo-code programs were implemented by the author of this thesis in C++ programming language. The implementation with all the helper methods and data structures took little over 700 lines of code. In addition to just implementation there were several hundreds of lines of unit test cases to make sure all code was working properly.

4.1   The Alpha Centaurian encoding

The problem this thesis is to solve originates from www.codingame.com where Nintendo posted it as a sponsored puzzle for participants to solve [1]. A screenshot of the website can be found in Appendix A. At the time of writing this thesis the challenge had been available online for more than three years with only 272 completions during that time. By all measurements this is the hardest puzzle on the site.

The context for the puzzle is that the SETI program has been receiving series of messages from Alpha Centauri, but they have no idea of the content of these messages. What they do know is how Alpha Centaurians are encoding these messages and pseudo-code for this operation is given for participants to use.

```
1 READ size
2 READ size / 16 integers in array a
3 WRITE size / 16 zeros in array b
4
5 For i from 0 to size - 1:
6     For j from 0 to size - 1:
7         b[(i+j)/32] ^= ((a[i/32] >> (i%32)) &
8                   (a[j/32 + size/32] >> (j%32)) & 1) << ((i+j)%32)
9
10 PRINT b
```

Program 4.1 *Alpha Centaurian encoding operation.*

The algorithm seems to be written in this form to make understanding of the operation as hard as possible for programmers not used to reading algorithms. First the algorithm reads something called *size* and then both reads *size*/16 and outputs *size*/16 integers in an array. It is further elaborated in the puzzle context that the *size* is restricted to $0 < size \leq 256$. In reality smallest test value for *size* is 32 and from there it is incremented in multiples of 32 up until 256.

Next subsection, 4.1.1 will explore the algorithm in detail and subsection 4.1.2 provides brute force algorithm for decoding as well as explanation why this approach is not feasible when $size > 64$.

4.1.1   Problem analysis

For analyzing the Centaurian algorithm let $size = 32$. This is the smallest size for which there are tests in the puzzle website and that selection makes understanding the algorithm easier for now. This means that on line 2 there are two 32bit integers to be read in the array a and the same amount of space reserved for the output array b on line 3.

Next, in the nested for loop variables i and j both iterate from 0 to 32. This is to say that whatever is inside the nested for loops will be ran for all $i, j : 0 \leq i < 32,\ 0 \leq j < 32$.

A closer look at lines 7 & 8, the actual operation, reveals a structure like this:

```
1         b[(i+j)/32] ^= (X & 1) << ((i+j)%32)
```

Program 4.2 *Structure of a single iteration of the double loop.*

Here $(X\&1)$ is just selecting the lowest bit of whatever X is and using XOR-equals

operator to XOR it into the array b. The location where the bit is XORred to is combination of the left side and the right side of the line: It is XORred to $(i+j)/32$th integer at position $(i+j)\%32$, which is exactly same as to say array b is just an array of bits and XORring happens to its $(i+j)$th bit because $\lfloor a/n \rfloor + (a \mod n) = a$.

Now, all that is left is to understand what the lowest bit in X stand for. In the nested loop X is as follows:

```
1        (a[i/32] >> (i%32)) & (a[j/32 + size/32] >> (j%32))
```

Program 4.3 *The bit for the xor operation.*

Here two parts are combined by logical AND. Left side uses only variable i while right side uses only variable j. Both sides are again selecting a bit to be the lowest bit using the same logic as before. The bit selected by the left side is simply $\lfloor i/32 \rfloor + (i \mod 32) = i$. Right side is otherwise the same, but it has constant term $size/32$ in it. In the beginning it was stated that the number of integers in the array a is $size/16$, hence $size/32 = size/16/2 = |a|/2$. Altogether this means that the algorithm uses the array a as two separate bit arrays, first half which is indexed with variable i and second half which is indexed with variable j. For now on, the first half of the array a is denoted a_L and second half a_R.

So, to bring it all together: Inside the nested for-loop the Centaurian operation chooses ith bit from a_L and jth bit from a_R. Next it uses logical AND operator to combine these and lastly the result is XORred to the array b in index $(i+j)$.

$$b_0 = a_{L0} \wedge a_{R0}$$
$$b_1 = (a_{L0} \wedge a_{R1}) \oplus (a_{L1} \wedge a_{R0})$$
$$b_2 = (a_{L0} \wedge a_{R2}) \oplus (a_{L1} \wedge a_{R1}) \oplus (a_{L2} \wedge a_{R0})$$
$$b_3 = (a_{L0} \wedge a_{R3}) \oplus (a_{L1} \wedge a_{R2}) \oplus (a_{L2} \wedge a_{R1}) \oplus (a_{L3} \wedge a_{R0})$$
$$\cdots$$
$$b_{2size-3} = (a_{Lsize-2} \wedge a_{Rsize-1}) \oplus (a_{Lsize-1} \wedge a_{Rsize-2})$$
$$b_{2size-2} = a_{Lsize-1} \wedge a_{Rsize-1}$$

Table 4.1 *The output array b of the Centaurian operation*

Now, there are two visual ways for interpreting the operation. First, it can be viewed as a matrix where the value of each cell $c_{ij} = a_{Li} \wedge a_{Rj}$. Next, the bits in the output array b are formed by XORring together cross-diagonal cells such that $b_k = \sum_{i+j=k} c_{ij}$ with addition operation replaced with XOR. Table 4.2 shows an example of this.

	a_{L0}	a_{L1}	$\ldots$	$a_{Lsize-1}$
a_{R0}	$a_{L0} \land a_{R0}$	$a_{L1} \land a_{R0}$		$a_{Lsize-1} \land a_{R0}$
a_{R1}	$a_{L0} \land a_{R1}$	$a_{L1} \land a_{R1}$		$a_{Lsize-1} \land a_{R1}$
$\ldots$				
$a_{Lsize-1}$	$a_{L0} \land a_{Rsize-1}$	$a_{L1} \land a_{Rsize-1}$		$a_{Lsize-1} \land a_{Rsize-1}$

Table 4.2 *Matrix visualization of the Centaurian algorithm*

	a_{R0}	$\land($				$\ldots$	a_{L2}	a_{L1} a_{L0}	$)$
	a_{R1}	$\land($				$\ldots$	a_{L1}	a_{L0}	$)$
	a_{R2}	$\land($				$\ldots$	a_{L0}		$)$
	$\ldots$								
	$a_{Rsize-3}$	$\land($			$a_{Lsize-1}$	$\ldots$			$)$
	$a_{Rsize-2}$	$\land($		$a_{Lsize-1}$	$a_{Lsize-2}$	$\ldots$			$)$
$\oplus$	$a_{Rsize-1}$	$\land($	$a_{Lsize-1}$	$a_{Lsize-2}$	$a_{Lsize-3}$	$\ldots$			$)$
		$b_{2size-2}$	$b_{2size-3}$	$b_{2size-3}$		$\ldots$	b_2	b_1 b_0	

Table 4.3 *Carry-less multiplication visualization of the Centaurian algorithm*

The second way of visualizing the problem is to re-arrange rows of the matrix in a way where previous diagonals are vertical. This creates familiar ladder-like structure which is how polynomial multiplication may be visualized. But, in this case the usual binary addition operation has been changed to XOR-operation. What this effectively does, is, that it now discards carry bits, and hence, this operation is called *carry-less multiplication*. The structure can be seen in the table 4.3.

4.1.2 A brute force solution

One brute force solution can be created by just testing all values for a_L one-by-one. It is enough to consider only a_L and not a_R because $a_R = b/a_L$ when there is no remainder left from the division. The program 4.4 is given as an example.

```
1 a := Array of zeroes
2 b := Encrypted message
3
4 Loop
5     Add 1bit to a
6     If overflow
7         return results
8     q, r := div(b, a)
9     If r = 0
10         results.append(a, q)
```

Program 4.4 *A brute force solution for decrypting Centaurian messages.*

Let n be the number of bits in a. The loop is run $O(2^n)$ times, adding a bit to the array takes at most $O(n)$ and division can be done in $O(n^2)$ operations. Therefore, the total asymptotic complexity of the algorithm is $O(2^n(n^2 + n)) \approx O(2^n)$.

If we take ordinary high-end PC with 3GHz processor with 4 cores and assume it is able to perform a single step of the previous algorithm in each clock cycle, then the 32bit version of the problem will take approximately 0.36s to solve. However, the 64bit version takes almost 49 years and the hardest problem, the 256bit one, takes ridiculous 10^{59} years. It is clear, that the brute force approach is not going to work.

4.2 Reformulation of the problem

Let us consider what the problem would look like if it was given a mathematical notation. Let AND operation be multiplication and XOR operation addition. Next, if we represent the two input bit arrays a_L and a_R from section 4.1.1 as polynomials $f(x)$ and $g(x)$ respectively with coefficients $a_{Li}, a_{R_j} \in \{0, 1\}$, and $n = size - 1$, such that

$$f(x) = a_{L0} + a_{L1}x + a_{L2}x^2 + \ldots + a_{Ln}x^{Ln},$$
$$g(x) = a_{R0} + a_{R1}x + a_{R2}x^2 + \ldots + a_{Rn}x^{Rn}.$$

Now, multiplication of $f(x)$ and $g(x)$ according to section 2.7 would yield following results

$$f(x)g(x) = b_0 + b_1x + b_2x^2 + \ldots + b_kx^k$$
$$b_k = \sum_{i+j=k} a_{Li}a_{Rj},\ 0 \geq k \geq 2n$$
$$b_0 = a_{L0}a_{R0} = a_{L0} \wedge a_{R0}$$
$$b_1 = a_{L0}a_{R1} + a_{L1}a_{R0} = (a_{L0} \wedge a_{R1}) \oplus (a_{L1} \wedge a_{R0})$$
$$\ldots$$
$$b_{2n-1} = a_{Ln-1}a_{Rn} + a_{Ln}a_{Rn-1} = (a_{Ln-1} \wedge a_{Rn}) \oplus (a_{Ln} \wedge a_{Rn-1})$$
$$b_{2n} = a_{Ln}a_{Rn} = a_{Ln} \wedge a_{Rn}$$

From this formulation it is clear that the resulting coefficients from multiplication of $f(x)g(x)$ are exactly the same b_i than the Centaurian algorithm produces in section 4.1.1. So, the original decryption problem translates to one of finding $f(x)$ and $g(x)$ when $f(x)g(x)$, the bit vector b, is given. But what we do not know at this point is whether such an operation is possible and whether an algorithm needed to make such an operation even exists. However, from chapter 3 we do know that an

algorithm for factorization of polynomials over finite fields does exist.

For existence of the decryption algorithm for this formulation we need to ensure that the coefficient set $S = \{0, 1\}$ with two laws of composition, AND for multiplication and XOR for addition, form a finite field, and that polynomials over such a field have a unique factorization. Then, and only then, can all possible $f(x)$ and $g(x)$ be composed of the factors of $f(x)g(x)$ to provide a solution to the original problem.

4.3   Finite field F_2

Let set $S = \mathbb{Z}_2 = \{\overline{0}, \overline{1}\}$ and $+$ and $\cdot$ be two laws of composition over S, called addition and multiplication respectively, be defined as follows:

+	0	1		·	0	1
---	---	---		---	---	---
0	0	1		0	0	0
1	1	0		1	0	1

Here S has 0 as the neutral element for addition, called zero, and 1 as the neutral element for multiplication, called unit, hence, S creates a monoid according to definition 2.1.1 with both addition and multiplication.

With regards to inverse, the unit element of S, 1, is its own inverse for both laws of composition. From the matrices above, it is easy to check that $1 + 1 = 0$ and $1 \cdot 1 = 1$. Therefore S forms a group according to definition 2.2.1 with addition. Also, as the matrices that define the operations are symmetric along its diagonal, both addition and multiplication are commutative.

Multiplication is clearly distributive over addition and with this limited size of S that is easy to see, or even try all combinations out. And, as S is a commutative group under addition, multiplication is associative and because multiplication is distributive over addition, $(S, +, \cdot)$ is a ring according to definition 2.3.1. And, because both operations are commutative and S has no zero divisors, S forms a commutative integral domain.

From theory in section 2.4, it is clear that S has only trivial ideals, 0 and S. And, because unit element 1 generates the whole set S by multiplication $1 \cdot S$, S is a principal ideal. Furthermore, because S is the only principal ideal, hence all ideals of S are principal (except trivial 0), it follows that S is a principal ideal ring as defined in 2.4.1.

As S has been established to be a commutative integral domain and a principal ideal ring, it is by definition 2.4.2 also a principal ideal domain. Furthermore, theorem 2.5.3 states that every PID is also a universal factorization domain, making S a UFD too.

And lastly, as S is a commutative ring, it has $0, 1 \in S$ such that $0 \neq 1$ and as a residue class of a prime number its multiplicative monoid of non-zero elements is a group, S is a field according to definition 2.6.1. As a field S is perfect, because it fulfills the criterion expressed in section 2.6.

4.3.1 Polynomials over F$_2$

According to theorem 2.7.1 a polynomial ring $R[x]$ over ring R that is a UFD is also a UFD. Hence, Polynomial ring $F_2[x]$ over F_2 is a UFD because F_2 has been established as a UFD in the previous section. And consequently, theorem 2.5.1 states that all $f(x) \in F_2[x]$ have a unique factorization to irreducible factors. The theorem claims the factors are unique up to unit factors, but as there is only one unit in F, namely 1, it follows that the factorization is truly unique.

From this we conclude that polynomials over F_2 are in fact factorisable and algorithms presented can be used in order to solve the decryption problem.

4.4 Polynomials operations on computer system

This section will provide information about the representation used in this thesis for polynomials in a computer system. Most of the computations with polynomials represented in this way have their algorithms explained while some trivial helper methods are omitted. Algorithms were implemented in C++, but to remove unnecessary details of the language, they were written in pseudo-code in this thesis.

4.4.1 Representation of polynomials

Polynomials over F_2 may only have a coefficient of 0 or 1, and hence, using one bit in a computer to represent one coefficient would allow a near perfect packing, unless the polynomial is really *sparse*, meaning that if k is the number of non-zero coefficients in $f \in F_2[x]$, then $deg(f) \ggg k$. For this thesis polynomials are considered to be *dense*, which is the opposite of being sparse, and means that $deg(f) \approx 2k$.

With these assumptions the basic building block of the polynomial representation is an unsigned 32bit integer, which is capable of representing a polynomial with a degree < 32. Now, let a be an unsigned integer, a_0 the least significant bit and a_{31} the most significant bit, then the polynomial $f \in F_2[x]$ with $deg(f) \leq 31$ is represented as

$$f = \sum_{i=0}^{31} a_i x^i.$$

As an example, the representation a of a polynomial $f(x) = x^8 + x^5 + x^4 + 1$ would be

$$a = 153_d = 0x99_h = 10011001_b.$$

Next, this representation of polynomials is extended to polynomials with arbitrary degrees by having an array of unsigned 32bit integers, where each integer represents a block of 32 coefficients. So, a polynomial $f \in F_2[x]$ with degree n is represented in an array of unsigned integers b as

$$f = \sum_{i=0}^{n} a_i x^i$$

where a_i is the $(i \mod 32)$th bit of the $\lfloor i/32 \rfloor$th integer in the array b.

Some of the algorithms, like the fast division of polynomials described in section 3.1.2 rely on $\mod x^l$ operations where l is a power of 2, and hence, the representation of polynomials must have a degree of $n = 2^k - 1$ for some $k > 0$. This restriction is realized by first having the smallest representation of polynomials to be one unsigned 32bit integer with maximum degree $n = 2^5 - 1 = 31$. Also, the length of an array representing a polynomial will be made to be a perfect power of 2. The size of the representation array b for $f \in F_2[x]$ with $n = deg(f)$ may be computed as

$$|b| = 2^{\lfloor \lg n \rfloor + 1 - 5}.$$

4.4.2 Derivative

The formal derivative described in section 2.8.3 for polynomials over field F_2 has two effects on a polynomial $f \in F_2[x]$. Firstly, all terms of polynomial f which have an even exponent for the variable x will vanish. To see why this happens we have to first consider a single term with an even exponent $g = x^{2k} \in F[x]$, where $k > 0$, and its derivative:

$$g' = 2kx^{2k-1} \equiv 0 \mod 2.$$

Next, by applying the first rule, linearity, in 2.8.3, all terms of f where the exponent is even, hence can be written in a form $2k$ for some $k > 0$, will vanish from the result of the derivation.

Secondly, all terms with odd exponent will have their exponents reduced by one. This can be seen when we consider a single term with odd exponent $g = x^{2k+1} \in F_2[x]$, where $k > 0$, and its derivative:

$$g' = (2k + 1)x^{2k} = 2kx^{2k} + x^{2k} \equiv x^{2k} \mod 2.$$

And again, the same rule of linearity allows application of this result to all terms with odd exponent in f.

Now, as the representation of polynomials over F_2 is an array of unsigned integers in this thesis, the algorithm for derivation should be fast for them. The following algorithm takes as an input an array a with n 32bit unsigned integers and outputs an array b with the result.

```
1 Function derivative(a)
2     b := []
3     For i from 0 to size(a) - 1:
4         b[i] = (a[i] & 0xAAAAAAAA) >> 1
```

Program 4.5 *Derivation of polynomials over F_2.*

Here the AND operation with $0xAAAAAAAA$ removes all coefficients with an even exponent, taking care of the first effect described earlier, because $0xAA_h = 10101010_b$. Afterwards shifting the result right by one bit will reduce exponents of all odd coefficients by one, handling the second effect. Lastly, we notice that there are no carry-overs from one integer to another as the lowest coefficient in this representation is always even, and hence, will vanish without carrying over.

4.4.3 Square root of a squared polynomial

From theorem 3.2.2 we get two important properties for polynomial $f \in F_2[x]$. We can decide whether f is a square of some $g \in F_2[x]$, meaning $f = g^2$, just by checking that every exponent in f is even. Furthermore, to avoid going through each exponent, we can harness the derivation algorithm, because we know from section 4.4.2 that if $f \in F_2[x]$, $f \neq 0$ and $f' = 0$, all terms in f have an even exponent.

The second important property from 3.2.2 is that to compute g we just have to halve each exponent in f, taken that we first made sure f was a squared polynomial

with only even exponents. The algorithm for computing the square root of a squared polynomial is given next. It takes as an input an array f, which contains coefficients in 32bit unsigned integers and returns an array r containing the result.

```
1 Function squareRoot(f)
2     r := 0
3     For i from 0 to deg(f)
4         If f.at(i) Then
5                 r.set(i/2)
6     return r
```

Program 4.6 *Square root of a squared polynomial over F_2.*

In the algorithm each even bit from f is selected and placed in its proper place in the result array r.

4.4.4   Squaring of a polynomial

Squaring of a polynomial over F_2 is the exact opposite of the algorithm from the previous section. Each exponent of input polynomial f is doubled and the result is returned as polynomial r.

```
1 Function square(f)
2     r := 0
3     For i from 0 to deg(f)
4         If f.at(i) Then
5                 r.set(2i)
6     return r
```

Program 4.7 *Squaring of a polynomial over F_2.*

4.4.5   Reversing bits of an unsigned integer

The faster algorithm for dividing polynomials presented in section 3.1.2, and which is heavily used in Berlekamp's algorithm, requires reversing coefficients of polynomials. And, because polynomials are represented as an array of unsigned integers where each bit represents a single coefficient, reversing the order of bits within an unsigned integer is what has to be done efficiently. A naive approach would reverse bits of an unsigned integer a to b like this:

```
1 Function naiveReversal(a)
2     b := 0
3     For i from 0 to n:
4         b |= ((a >> i) & 1) << (n - 1 - i)
```

Program 4.8 *Naive reversal of bits for unsigned integer of size n.*

However, this leads to complexity of $O(n)$ and it will slow down the main algorithm. Some of this complexity can be avoided by creating a bit reversal lookup table where all 256 8bit sequences have their reversal in the very same index their value points to. While the overall complexity still remains linear with respect to number of bits, it will reduce the constant factor significantly as reversing bits of a 32bit unsigned integer will take 4 lookups to the table instead of 32 iterations of a loop.

```
1 Function reverseBits(n)
2     b := (BIT_REVERSE_TABLE[n & 0xFF] << 24) |
3           (BIT_REVERSE_TABLE[(n >> 8) & 0xFF] << 16) |
4           (BIT_REVERSE_TABLE[(n >> 16) & 0xFF] << 8) |
5           (BIT_REVERSE_TABLE[(n >> 24) & 0xFF])
6     return b
```

Program 4.9 *Reversing bits of a 32bit unsigned integer a with a table.*

To make the reversal of bits to conform with equation 3.1, every 32bit integer in the array representing the polynomial must be reversed and moved to correct place. Also, the whole array of bits must be shifted to proper place. This algorithm is as follows.

```
1 Function rev(a)
2     r := []
3     For i = 0 to size(a) - 1
4         r[i] = reverseBits(a[size(a) - i - 1])
5     r >>= (32 * size(a) - deg(a) - 1)
6     return r
```

Program 4.10 *Reversing an array a of 32bit integers representing a polynomial.*

Some details are omitted from the algorithm. Helper function $size(a)$ returns the number of 32bit integers in an array a, $deg(a)$ returns the degree of polynomial f represented by the array a and $>>=$ is an operator shifting all bits in the array to left by the amount given.

4.4.6 Karatsuba multiplication

The implementation of Karatsuba's algorithm for fast polynomial multiplication presented in section 3.1.1 is quite straight forward. Any polynomials of size 16 or less will be multiplied with the well-known quadratic algorithm for binary multiplication, except instead of addition XOR-operation is used.

Larger polynomials are split up according to Karatsuba's equation and multiplied in smaller parts. Next, those results from smaller multiplications are again combined as the result of larger multiplication. The following algorithm relies on two helper functions which implementations are not included. Namely, *size* and *split*. Former returns the size of the polynomial representation that is always of the form $size = 2^k$, $k > 3$, and latter splits a polynomial to two, half the *size* of the original.

```
1  Function karatsuba(f, g)
2      // Shall we just multiply the usual way
3      If size(f) <= 16 then
4          r := 0
5          For i from 0 to size(f) - 1:
6              r ^= (f * ((g >> i) & 1)) << i
7          return r
8
9      // Calculate FG = (F0 + tnF1)(G0 + tnG1)
10     //  = (1 + tn)F0G0 + tn(F0 + F1)(G0 + G1) + (tn + t2n)F1G1
11     tn := size(f) >> 1
12
13     (f0, f1) := split(f)
14     (g0, g1) := split(g)
15     f0g0 := karatsuba(f0, g0)
16     f1g1 := karatsuba(f1, g1)
17     fg := karatsuba((f0 ^ f1), (g0 ^ g1))
18
19     return f0g0 ^ (f0g0 << tn) ^ (fg << tn)
20             ^ (f1g1 << tn) ^ (f1g1 << (tn << 1))
```

Program 4.11 *Implementation of Karatsuba algorithm for multiplication*

4.4.7 Fast Euclidean division

Fast polynomials division algorithm explained in section 3.1.2 benefits greatly from the field of polynomials being F_2. The most computational heavy operation in the algorithm is the step from the Newton's method $(2g - fg^2)$. Here the first part, $2g \equiv 0 \mod 2$, and the square g^2 can be easily computed with method presented in

section 4.4.4. This leaves only one multiplication, which in turn can be computed with Karatsuba's algorithm from section 4.4.6. Pseudocode implementation of this algorithm is provided next.

```
1  Function fastDivision(a, b)
2      If deg(a) < deg(b) then
3          return (0, a)
4
5      m := deg(a) - deg(b)
6      r := ceil(log2(m) + 1)
7      f := rev(b)
8      g := 1
9
10     For i = 1 to r:
11         g = modExponent(karabatsu(f, square(g)), 1 << i)
12
13     s := modExponent(karabatsu(rev(a), g), m + 1)
14     q := rev(s) << (m - deg(s))
15     rem := karabatsu(b, q) + a
16     return (q, rem)
```

***Program 4.12** Implementation of fast Euclidean division*

This implementation relies on several helper methods that are omitted from this work. Function $deg(f)$ returns the degree of a polynomial f, $ceil(d)$ rounds its parameter d up to the next whole integer and $modExponent(f, e)$ handles modulo computation $f \mod x^e$, which is basically just removing all terms with exponent $\geq e$. Helper methods $square(f)$, $rev(f)$ and $karatsuba(f, g)$ are explained in sections 4.4.4, 4.4.5 and 4.4.6 respectively.

4.4.8   Greatest common divisor

The algorithm for computing gcd is really straight forward according to section 3.1.3 and using the division algorithm from the previous section.

```
1  Function gcd(a, b)
2      If b = 0 then
3          return a
4      Else
5          (q, r) = fastDivision(a, b)
6          return gcd(b, r)
```

***Program 4.13** Implementation of Euclid's algorithm*

4.5 Square-free factorization

Square-free factorization is the preliminary step for factorization. It is relatively cheap computation as explained in section 3.2. Yun's algorithm [21] does not suffice here as section 4.3 established F_2 to have a positive characteristic. Hence, we need to utilize work by Gianni and Trager [9] to implement square-free factorization algorithm.

The algorithm itself consists of two parts. First part is the *basicSquareFree* below in code 4.14 which is the Muller's algorithm [21] and the latter part, in code 4.15, with Gianni's and Trager's [9] proposed way of utilizing Muller's algorithm for fields with positive characteristic.

```
1  Function basicSquareFree(f)
2      P := []
3      c := gcd(f, derivate(f))
4      b := fastDivision(f, c)
5
6      For i = 1 until b = 1
7          tmp = gcd(c, b)
8          c = fastDivision(c, tmp)
9          P[i] = fastDivision(b, tmp)
10         b = tmp
11
12     return (P, c)
```

Program 4.14 *Muller's algorithm for basic square-free computation*

```
1  Function squareFree(f)
2      (P, q) = basicSquareFree(f)
3      If deg(q) = 0
4          return P
5      Else
6          h := squareRoot(q)
7          P2 := squareFree(h)
8          return merge(P, P2)
```

Program 4.15 *Gianni's square-free algorithm*

These algorithms are based on already provided algorithms *derivative* 4.5, *squareRoot* 4.6, *fastDivision* 4.12 and *gcd* 4.13. In addition to these two other omitted helpers are needed, $deg(f)$, which returns the degree of a polynomial f, and *merge*, which combines two arrays of factors to one.

4.6 Berlekamp's algorithm

The Berlekamp's algorithm is the main algorithm that is used to factorize all polynomials that resulted from the square-free factorization. Berlekamp's algorithm described in section 3.3 requires polynomials to be from a field and that was established in section 4.3. The algebra that the algorithm is based on (theory of fields and universal factorization domains) is explained in section 2.5.

4.6.1 The Q matrix

The first phase of the Berlekamp's algorithm is to build the Q matrix. The matrix is formed as an array of rows where each row is a polynomial, and just like any other polynomial, it is represented as an array of 32bit integers.

The equation 3.2 states that an ith row of the Q matrix equals to $x^{pi} \mod f(x)$, where $f(x)$ is the polynomial we are factorizing and $p = 2$ because of the field F_2. But, Berlekamp's algorithm uses matrix $(Q - I)$, hence, the algorithm 4.16 builds that directly instead of just Q.

```
1  Function qmat(f)
2      rows := []
3      p := pol(1)
4      I := pol(1)
5
6      For i = 1 to deg(f)
7          (q, r) := fastDivision(p, f)
8          rows[i] = r ^ I
9          p <<= 2
10         I <<= 1
11     return rows
```

Program 4.16 *Building the Q matrix*

Helper method $fastDivision$ was presented in program 4.12 and $pol(d)$ creates a polynomial representation from integer value d.

4.6.2 Base vectors of the null space

To compute base vectors of the null space of matrix $(Q - I)$, the matrix needs to be triangulated using Gaussian elimination. Basically this means applying Gaussian elimination until the matrix is *upper triangular*, meaning all entries below main

diagonal are 0. While the triangulation takes place, another matrix $A = I$, is produced by duplicating every row operation on it also.

Once the $(Q-I)$ matrix has been triangulated, the base vectors of the null space are the row vectors of A that correspond to 0 rows after triangulation. The algorithm for computing the null space is given next.

```
1  Function nullspace(M)
2      A := Identity matrix
3      currentRow := 1
4      Loop
5          // Find the leading row
6          d := -1
7          leading := -1
8          For r := currentRow to M.n
9              If deg(M.row[r]) > d
10                 d := deg(M.row[r])
11                 leading := r
12
13         If leading < 0
14             break
15
16         // Swap and update all other rows with same degree
17         M.swap(currentRow, leading)
18         A.swap(currentRow, leading)
19         For r := currentRow + 1 to M.n
20             If deg(M.row[r]) = d
21                 M.rows[r] += M.rows[currentRow]
22                 A.rows[r] += A.rows[currentRow]
23
24         currentRow += 1
25
26     np := []
27     For r = 1 to M.n
28         If M.rows[r] = 0
29             np.append(A.rows[r])
30     return np
```

Program 4.17 *Compute null space of the (Q - I) matrix*

4.6.3 Factorization

The last step of Berlekamp's algorithm is to use the base vectors of previously calculated null space to get the factors of f. Here it becomes meaningful that the input for the following algorithm is square-free. The number of base vectors in the

null space tells us how many irreducible factors there are, but it does not tell what is the multiplicity of those factors. But, as the input is square-free, we know that the multiplicity of each factor is 1. The algorithm is presented below and explained right after it.

```
1  Function berlekamp(f)
2      QI := qmat(f)
3      nsp := nullspace(QI)
4      k := nsp.size()
5      factors := []
6
7      // Continue until all factors have been found
8      While !nsp.empty() AND factors.size() < k
9          base := nsp.take()
10
11         If factors.empty() Then
12             p1 := gcd(f, base)
13             p2 := gcd(f, base ^ 1)
14             If deg(p1) > 0 AND p1 != f Then
15                 factors.append(p1)
16             If deg(p2) > 0 AND p2 != f Then
17                 factors.append(p2)
18         Else
19             tmp := factors
20             factors := []
21             For factor in tmp
22                 p1 := gcd(factor, base)
23                 p2 := gcd(factor, base ^ 1)
24                 If deg(p1) > 0 Then
25                     factors.append(p1)
26                 If deg(p2) > 0 Then
27                     factors.append(p2)
28
29     // In case this was irreducible
30     If size(factors) == 0
31         factors.append(f)
32     return factors
```

Program 4.18 *Implementation of the Berlekamp's algorithm*

The implementation begins with computing $(Q - I)$ matrix on line 2 and on the following line by computing its null space. At this point, according to section 3.3.4, we already know that there will be as many irreducible factors as there are base vectors in the null space. The algorithms for computing the $(Q - I)$ matrix and null space were given earlier, 4.16 and 4.17 respectively.

Next, the while loop on line 7 continues until all base vectors have been used, or all factors have been found. Section 3.3.4 guaranteed that all factors will be found. Inside the loop there either are no known factors yet, line 11, when the base vector is used to gain first factorization of f, or there are known factors, line 18, and all known factors are used in order to refine factorization.

Last step before returning factorization is to check whether there were any factors. If the list of factors is empty at this point, the f was irreducible. Without returning f when it is irreducible, line 19 would not be able to collect all irreducible factors.

4.7   Combining factors to result

After running both square-free factorization and Berlekamp's algorithm the resulting factors have to be handled with multiplicities, combined as unique solutions a_L and a_R and ordered alphabetically for output. The algorithm for this has been omitted.

One note-worthy aspect of combining factors to form polynomials a_L and a_R is the size limit. Polynomial f with sufficiently low degree may very well fit into a_L and then solutions $a_L = 1, a_R = f$ and $a_L = f, a_R = 1$ must be added to possibilities. On the other hand, it is also possible to combine factors in a such a way that the result does not fit into a_L. Such a solution, while mathematically being valid, cannot be presented, and hence, must be discarded.

5.   CONCLUSIONS

The Nintendo challenge [1] solved was decrypting messages that were encrypted with an algorithm expressed in plain C++ code. After analyzing the encryption algorithm, it became obvious that it interpreted the input as two binary polynomials, polynomials over F_2, and multiplied them together in order to encrypt the message.

The challenge itself was posted as a really difficult puzzle with an opportunity to apply for a job at Nintendo if solved. While the problem had already been published for over 3 years, there were just 272 accepted submissions for it. The implementation of this thesis was the 273rd.

While the problem of factorizing polynomials over $\mathbb{Z}$ or finding factors of an integer are really hard problems, it turns out that factorization of polynomials over finite fields is not. Berlekamp in 1967 was the first to figure out a deterministic algorithm for doing so [2], and as it works reasonably well for small fields, it was chosen as the backbone of the implementation of this thesis.

The factorization itself was implemented in two steps. First an algorithm, often credited for Yun, was used to find square-free decomposition and after that Berlekamp's algorithm to find all factors from these already square-free polynomials. The preliminary step is included as it is much faster than Berlekamp's algorithm and therefore may reduce the complexity of the problem significantly.

Most of the research concentrated on figuring out the mathematics behind Berlekamp's algorithm to understand why it works. This included enough basic algebra to build up the understanding needed to prove that factorization of polynomials over F_2 is possible (irreducible factors do exist) and to figure out how to achieve the actual factorization.

Another research topic was how to efficiently implement the algorithms with C++ so that the original problem may be solved. If important details were inefficiently implemented, it could render the resulting implementation useless for solving the problem. This is why binary representation and all operations on it were carefully thought out.

There were two sources of confusion during the research phase for this thesis. While the algorithms of Berlekamp and Yun are both quite straight forward and simple, there are variations and minute details that differ according to sources.

Many sources, like on-line course materials available and Wikipedia, refer to Berlekamp's algorithm as one that factorizes square-free polynomials over finite fields. As an example, Harasawa et al. just simply note that "For the factorization of square-free polynomials over finite fields, the Berlekamp algorithm is well known." [10], and this is the often found way of presenting Berlekamp's algorithm. However, Berlekamp's original algorithm [2] makes no mention of input having to be square-free.

Even in his later work for factorization of polynomials over large finite fields [3], where he does mention using the same trick with formal derivation as Yun [21] did, he claims that "Although it may be advisable to eliminate the repeated factors of $f(x)$ immediately, it is not necessary to do so.". However, Berlekamp does mention in a footer text in [3] that he extended the original 1967 algorithm with some computation using $gcd(f(x), f'(x))$, which would indicate utilization of square-free algorithm of some sort.

While Berlekamp's algorithm does not require input polynomial to be square-free, it is understandable why such requirement would be given. As Knuth states in [14, p. 439], finding square-free factors can be done with standard techniques of formal derivative and gcd computation, and it speeds up the process of factorization "nicely". Also, Berlekamp's algorithm [2] includes extra complexity when a factor is repeated, so it does make sense to discard this complexity and add the requirement for input to be square-free, simultaneously making the whole algorithm faster.

Another source of confusion was that research in the field quite easily dismisses square-free factorization as something that can be done with Yun's algorithm [21]. An example of this is a quote "Using the deterministic algorithm of Yun (1976), stage 1 (square-free factorization) can be performed at essentially the cost of one gcd..." by Gathen and Gerhard [20].

The Yun's algorithm does square-free factorization, but only in a field of characteristic zero [21]. In a general case of factorizing over just a finite field F, it would have made sense to not go into detail how square-free factorization is done, but the work of Gather and Gerhard [20] is about factorization over F_2. And, F_2 has positive characteristic, hence, Yun's algorithm would not work.

Furthermore, in his work Yun [21] presents 3 algorithms for square-free factorization from which only one is his. Afterwards, Gianni and Trager [9] extended one of the

algorithms in Yun's paper to work for positive characteristic, but even that was not the one referred to as Yun's algorithm. In fact, it was Musser's algorithm [21] [9] that they based theirs on.

There has been a great number of improvements on factorization of polynomials over finite fields since 1967 when Berlekamp introduced his algorithm. Several new approaches ranging from probabilistic algorithms [12] to using the Fast Fourier Transform have been developed. Computers are being improved by increasing the number of cores available and opening up GPUs, graphical processing units, for parallelizing simple computations. Recent processors have special instruction sets for carry-less multiplication, which when utilized properly, could speed up the factorization algorithms even more.

For the writer of this thesis personally, it would be very interesting to research if Nvidia CUDA, a parallel computing platform, could be used to move some of the factorization algorithms to run on modern day GPUs. Mimicking the way how modern AI computation has already moved there. While this would not reduce the overall asymptotic complexity of the algorithms, it could dramatically lower constant terms making them much faster in practice.

BIBLIOGRAPHY

[1] "Nintendo sponsored contest," https://www.codingame.com/ide/puzzle/nintendo-sponsored-contest, accessed: 2018-11-12.

[2] E. R. Berlekamp, "Factoring polynomials over finite fields," *The Bell System Technical Journal*, vol. 46, no. 8, pp. 1853–1859, Oct 1967.

[3] ——, "Factoring polynomials over large finite fields," *Mathematics of Computation*, vol. 24, no. 111, pp. 713–735, 1970. [Online]. Available: http://www.jstor.org/stable/2004849

[4] P. Bhattachharya, S. Jain, and S. Nagpaul, *Basic Abstract Algebra*. the Press Syndicate of the University of Cambridge, 1994.

[5] D. G. Cantor and H. Zassenhaus, "A new algorithm for factoring polynomials over finite fields," *Mathematics of Computation*, vol. 36, no. 154, pp. 587–592, 1981. [Online]. Available: http://www.jstor.org/stable/2007663

[6] Z. Cao and H. Cao, "On fast division algorithm for polynomials using newton iteration," in *Information Computing and Applications*, B. Liu, M. Ma, and J. Chang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 175–180.

[7] L. N. Childs, *A Concrete Introduction to Higher Algebra*. Springer, New York, NY, 2009.

[8] S. Gajbhiye, M. Sharma, and S. Dashputre, "A survey report on elliptic curve cryptography," *International Journal of Electrical and Computer Engineering*, vol. 1, no. 2, p. 195, 12 2011, copyright - Copyright IAES Institute of Advanced Engineering and Science Dec 2011; Last updated - 2013-09-03. [Online]. Available: https://search-proquest-com.libproxy.tut.fi/docview/1429485698?accountid=27303

[9] P. Gianni and B. Trager, "Square-free algorithms in positive characteristic," *Applicable Algebra in Engineering, Communication and Computing*, vol. 7, no. 1, pp. 1–14, Jan 1996. [Online]. Available: https://doi.org/10.1007/BF01613611

[10] R. Harasawa, Y. Sueyoshi, and A. Kudo, "Improving the berlekamp algorithm for binomials $x^n - a$," in *Arithmetic of Finite Fields*, F. Özbudak and F. Rodríguez-Henríquez, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 225–235.

[11] V. C. d. R. Jr., *Elements of Algebraic Coding Systems*, 2014.

[12] E. Kaltofen and V. Shoup, "Subquadratic-time factoring of polynomials over finite fields," in *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, ser. STOC '95.   New York, NY, USA: ACM, 1995, pp. 398–406. [Online]. Available: http://doi.acm.org/10.1145/225058.225166

[13] A. Karatsuba and Y. Ofman, *Soviet Physics Doklady*, 1963.

[14] D. Knuth, *The Art of Computer Programming*.   Addison-Wesley, 1998.

[15] S. Lang, *Algebra*.   Springer-Verlag, 2002.

[16] P. Roelse, "Factoring high-degree polynomials over f2 with niederreiter's algorithm on the ibm sp2," *Math. Comput.*, vol. 68, pp. 869–880, 1999.

[17] H. S., *Advances in Cryptology - CRYPTO 2009*.   Springer-Verlag Berlin Heidelberg, 2009.

[18] A. . S. V. Schönhage, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, no. 3-4, pp. 281–292, 1971.

[19] C. Umans, "Fast polynomial factorization and modular composition in small characteristic," in *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, ser. STOC '08.   New York, NY, USA: ACM, 2008, pp. 481–490. [Online]. Available: http://doi.acm.org/10.1145/1374376.1374445

[20] J. von zur Gathen and J. Gerhard, "Arithmetic and factorization of polynomial over f2 (extended abstract)," in *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, ser. ISSAC '96.   New York, NY, USA: ACM, 1996, pp. 1–9. [Online]. Available: http://doi.acm.org/10.1145/236869.236882

[21] D. Y. Yun, "On square-free decomposition algorithms," in *Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation*, ser. SYMSAC '76.   New York, NY, USA: ACM, 1976, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/800205.806320
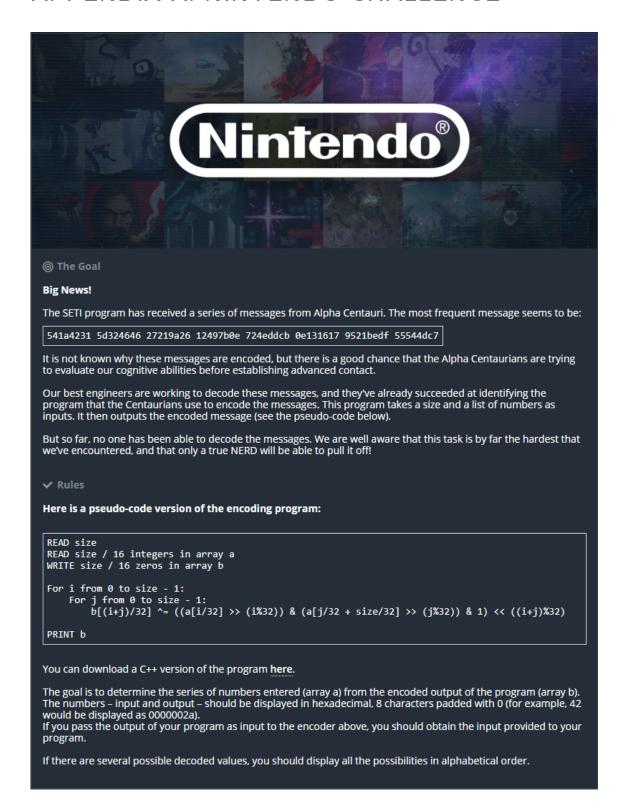
APPENDIX A. NINTENDO CHALLENGE

Figure **1** *Screenshot of the Nintendo challenge webpage. Retrieved 16.11.2018 from https://www.codingame.com/ide/puzzle/nintendo-sponsored-contest*