UNIVERSITÀ DEGLI STUDI DI MILANO
DIPARTIMENTO DI INFORMATICA

Alessandro De Piccoli

Optimized representations in cryptographic primitives

Ph.D. programme in Computer Science
Cycle XXXIV
Section INF/01
Subsection 01/B

Tutor
Prof. Andrea Visconti

Coordinator
Prof. Paolo Boldi

Academic year
2020 – 2021

To my parents, unattainable masters of counting and measuring

Acknowledgements

Although my Ph.D. programme finished on September 30th, 2021, I needed to extend a little that term due to COVID-19 pandemic, so, I consider today my last day as student. This has to be intended formally because I know that life's exams never finish.

First of all, I thank my parents. They are the lighthouse that guides me ashore in a safe harbor whenever I need. I cannot write enough words to thank them.

Beside them, there is the rest of my family, namely the irreplaceable group of uncles, aunts, cousins, great-uncles and great-aunts which is present in my life and makes me feel its friend instead of a simple relative.

I have to thank also my friends. It is not common in nowadays life to find special people whose friendship is real and strong. They are very important and I have learnt a little lesson from each of them, so thank you Elena, Francy, Kiki, Liz, Luca S., Marco, Manu, Miky, Rocco, Sizzy. Among them, a special thank goes to two friends whose support was also technical as well as crucial. Even if I am the only one who reached the position of Ph.D. student in Computer Science, they would have deserved it as much as me and maybe more, so, thank you Lorenzo Cameroni and Luca Casati.

It is time for technical acknowledgements. The most important person is undoubtely my tutor Prof. Andrea Visconti. My dream of a Ph.D. was dead and just a small group of people knew about my dream. One day, he made the magic: he offered me to participate in the selection. He did not know anything about my dream and just know a little about my abilities but he trusted in me and this is one of the most great honors in my life.

In addition to my tutor, I thank: Emanuele Bellini, Michela Ceria, Yousef Hammar, Rusydi Makarim, Sergio Polese, Lorenzo Riva and Martino Tiziani. They are brilliant colleagues and an important part of this work. Moreover, I thank all the people belonging to the LaSER⊕CLUB laboratory, in particular Stefano Cristalli.

At last, I thank the coordinator of my Ph.D. programme Prof. Paolo Boldi. He guided me in many situations since I had some responsibilities being the representative of the Ph.D. students in Computer Science. He was my C lecturer in one of the first courses I took at University, when I did not know very well what a computer is.

Milano – April 4th, 2022

Table of contents

Introduction

Having an optimized representation of any circuit is synonymous of handling it in small size which turns into efficiency when computing the output. This is useful especially when talking about cryptographic primitives that can be described as systems of boolean equations, namely a circuit.

There are many reasons to search for the shortest way of computing the output. First of all, think about Application-Specific Integrated Circuit (ASIC). They are devices dedicated to compute a particular function, e.g. an encryption algorithm. A smaller circuit for that means less costs when producing it and even a reduced power consumption. The latter is very important when talking about cryptographic security, because there exists side channel attacks [15, 45] that exploit the differences in power consumption to guess the secret key in a reduced set. In fact, the optimization is obtained keeping in mind to spend a constant time from an algorithmic point of view, i.e. the same number of operations for every input. Another security aspect affected by the optimization is to measure the best brute-force performance that an attacker could exploit.

Secondly, it is well known that the new Internet of Things (IoT) technology is already widespread. There are many devices in IoT, for instance medical ones, that handle sensitive data and, for this reason, they need a stronger protection. Unfortunately the environment is resource constrained, thus, reducing encryption algorithms could allow the integration of stronger cryptosystems in those devices.

Thirdly, faster algorithms mean reduced latency when communicating over Internet. For instance, think about a video-conference that needs to be secret. The high speed of the algorithm offers an improved experience beside a strong security. Moreover, if the key is compromised, good performances allow a fast restoring of the privacy of the communication. This could be even scaled to the scenario of a server that stores a big amount of keys. If it is optimized, it restores confidential communication in the shortest theorical time.

In this dissertation, we will show two optimizations that improve the state of art allowing better performances from both theoretical and practical point of view.

The first improvement is about polynomial multiplication that is a common operation in modern cryptography. We will deal only with multiplications having two factors of same degree, this means that we can think about an enumeration of multiplications, i.e. there is a one-to-one map between multiplications and degrees. Multiplying two polynomials is an easy task and has a general algorithm called *School-book*, working on any polynomial ring, even for any degree. In particular, it can be defined recursively which means that we can take the advantage of the improvements even from a single degree when studying the

algorithm for higher one. It is well known that, in 1960, Anatolij Karatsuba showed an algorithm [43] that can save some operations. It can be applied for almost all degrees and it is the base for many other improvements. Although the Karatsuba's improvement comes from 60s, we have waited until 2000s for new studies about polynomial multiplication [8], the author was Daniel Julius Bernstein. He proposed a refinement of the Karatsuba's formula and two new algorithms.

- The first one is the *Five-way recursion* whose base is in the Projective Lagrange Interpolation. It is currently the best when speaking about the size of the circuit, but its advantage is taken after the degree 128. This fact besides its high depth makes it less exploitable in real life implementations than other ones.

- The second algorithm is *Two-level seven-way recursion*. It reduces the size for degrees lower than 128 and has a good depth.

We will use the second algorithm as a base to reduce the number of the operations required by some polynomial products especially when the underlying ring has characteristic 2.

All these new algorithms lead to faster cryptographic primitives because they usually require computations in Galois fields and it is for this reason that polynomial product is widely studied. In 2015, Cenk and Hasan [19] proved that asymptotic estimates can be improved by using Projective Lagrange Interpolation as Bernstein did for the finite case. By an inspection of their formula, we will show how better asymptotic estimates can be reached although the cardinality of the field has a non negligible growth.

It is also possible to show that some of these improvements have a real case application. Quantum computers have been already built. Currently, they have not enough qubits to break real instances of RSA cipher, but they are concrete devices and this means that the menace is just a matter of time. Taking this into account, the National Institute of Standards and Technology (NIST) has started a standardization process for the Post-Quantum Cryptography (PQC). Many ciphers have been proposed, one of them is the McEliece cryptosystem. Its implementation take the advantage not only of the usual polynomial product, but also of the product of polynomials whose coefficients are polynomials as well. By replacing the implementation of the *School-book* algorithm with one of the new polynomial product techniques and changing the usage of the memory, the generation time of the McEliece key pair can be considerably decreased.

The second improvement is from a practical point of view and is about the hash function SHA-1. Although the first SHA-1 collision attack has been carried out in 2017 [76], the cryptographic hash function seems still robust against pre-image attacks and it is still widespread [77, 18]. The aim of the second part of this dissertation is to measure how many rounds of SHA-1 can be inverted in practice using SAT solvers. To do so, we will first model the problem of finding a pre-image of SHA-1 as a system of boolean formulas, explicitly describing the procedure to obtain such model. Then, we will configure the model based on different combinations of number of rounds and number of free bits in our target pre-image. Finally, to find a solution of the model, we will use a SAT solver on a server, testing several combinations of restart policies and polarity modes. We analyze and report the number and positions of the pre-image bits that can

be fixed to influence the ability of the SAT solver to find the remaining free bits of the pre-image in a shorter time. In particular, we execute partial pre-image attacks on 64-, 80-, 96-, 112- and 128-bit messages, outperforming the current state-of-the-art records.

Part I

Polynomial product

1. Finite fields

In this chapter, we will review some facts about finite fields. They can be used to understand the underlying mathematical structures in chapter 2. In particular, elements of a generic finite field of characteristic 2 can be represented as polynomials having coefficients in the set $\{0, 1\}$. This leads to a natural representations in hardware as bit-arrays and makes them suitable for cryptographic algorithms. In fact, for instance, McEliece [57] and NTRU [38] are two cryptosystems using algebra from fields of prime characteristic and have reached the third round of the NIST Post-Quantum standardization process.

1.1. Existence and uniqueness

It is well known that when p is a prime number, then $\mathbb{Z}/p\mathbb{Z}$ is a finite field, but there are many more finite fields. We want now to show the general structure of a finite field.

We are going to enunciate three facts that will be useful to depict the structure of a finite field.

- *Characteristic.* We start by defining what is the characteristic of the field. Take a finite field $(F; +, \cdot)$ and consider the ring homomorphism

$$\begin{array}{ccc} \mathbb{Z} & \longrightarrow & F \\ n & \longmapsto & n \cdot 1_F = 1_F + 1_F + \ldots + 1_F. \end{array}$$

  The kernel of the map is different from $\{0\}$, therefore it exists an n such that $n \cdot 1_F = 0$. The smallest positive n must be a prime number p, otherwise F containts two nonzero elements whose product is zero. We call the prime number p the *characteristic* of the field. It generates the kernel, i.e. the principal ideal $p\mathbb{Z}$. Therefore, we have an isomorphism between $\mathbb{Z}/p\mathbb{Z}$ and its image in F.

- *Cardinality.* Note also that the field $\mathbb{Z}/p\mathbb{Z}$ has no automorphisms different from the identity, since 1 maps to itself and additively generates $\mathbb{Z}/p\mathbb{Z}$. At this point, we can see F as a finite vector space whose coefficients are in $\mathbb{Z}/p\mathbb{Z}$. Let n be the dimension of the vector space, the cardinality of the finite field will be p^n.

- *Order of an element.* We need another fact that will help us to contruct a finite field. Let G an abelian finite group. The order of g divides the

cardinality of G for every g in G. To prove it, let $a, b \in G$ and consider the equivalence relation

$$a \ \rho \ b \iff \exists i : g^i \cdot a = b.$$

Every equivalence class has cardinality equal to $\mathrm{ord}(g)$ and the set of equivalence classes is a partition of G.

We are now ready to see the general structure of a finite field.

Theorem 1.1. *For every prime number p and every integer $n \geq 1$, there exists a finite field which order is p^n. It is the splitting field of*

$$f(X) = X^{p^n} - X.$$

Its elements are the roots of this polynomial.

Proof. We basically need to prove that field elements are the roots of the polynomial and there are p^n such roots.

- Take $r \neq 0$ an element of the field. In particular, it is an element of the group $F^\times$ having cardinality $p^n - 1$. As we have said, the order of an element divides the cardinality of the group, therefore $r^{p^n-1} = 1$. We have proved that field elements are the roots of the polynomiali f.
- The derivative of f is

  $$f'(X) = p^n X^{p^n-1} - 1 = -1,$$

  being p the characteristic of the field. Since the derivative has no zeros, f has no multiple roots, thus they are p^n.

At this point, we simply need to prove that the set of the roots is effectively a field.

- *Additive group.* Let r_1 and r_2 two roots of the polynomial f, $r_1 + r_2$ is again a root of the polynomial since

  $$(r_1 + r_2)^{p^n} - (r_1 + r_2) = r_1^{p^n} + r_2^{p^n} - r_1 - r_2.$$

  0 is trivially a root of f. Let r be a root of f, we have

  $$(-r)^{p^n} - (-r) = (-1)^{p^n} r^{p^n} + r.$$

  If p is odd, we have $(-1)^{p^n} = -1$. If p is even, the characteristic is 2, then $r = -r$. In both cases, $-r$ is again a root of f.

- *Multiplicative group.* Let r_1 and r_2 two roots of the polynomial f, $r_1 r_2$ is again a root of the polynomial since

  $$(r_1 r_2)^{p^n} - (r_1 r_2) = r_1^{p^n} r_2^{p^n} - r_1 r_2.$$

  1 is trivially a root of f. Let r be a root of f, we have

  $$(r^{-1})^{p^n} - r^{-1} = (r^{p^n})^{-1} - r^{-1},$$

  showing that r^{-1} is a root of f.

□

We have currently proved that a finite field of cardinality p^n exists, being p a prime and $n \geq 1$ an integer. The next question is: given a prime power, is there a unique finite field? The answer is, somehow, yes.

Proposition 1.1. *Let F a field and $f \in F[X]$. If F_1 and F_2 are both splitting fields for f, then every F-homomorphism $F_1 \to F_2$ is actually an isomorphism. In particular, any two splitting fields for f are F-isomorphic.*

Proof. [60, Proposition 2.7] □

Since a finite field is unique up to an isomorphism, we will use $\mathbb{F}_{p^n}$ to denote a finite field of order p^n. Therefore, given a finite field $\mathbb{F}_{p^n}$, we can see it as a vector space whose coefficients are in $\mathbb{F}_p$. For this reason it is trivial how to perform the addition of two elements, but the multiplication is not so immediate. We need to deepen the multiplicative structure of $\mathbb{F}_{p^n}^\times$.

Proposition 1.2. *Let ϕ the Euler function*

$$\phi = \#\{k : 1 \leq k < n,\ \gcd(k, n) = 1\}$$

and G a group whose cardinality is n. The number of generators of G is $\phi(n)$.

Proof. Let g a generator of G. The number of the generators of G is the number of integers i such that $\mathrm{ord}(g^i) = n$, i.e. n is the smallest integer r such that $(g^i)^r = 1$. Since g is a generator, $g^{ir} = 1$ if and only if n divides ir. In order to have no $r < n$ such that $n \mid ir$, i and n must be coprime. □

Euler ϕ has some properties that can be easily proved:

- if p is prime then $\phi(p) = p - 1$;
- if $\gcd(n, m) = 1$ then $\phi(nm) = \phi(n)\phi(m)$;
- if $n = \prod_i p_i^{m_i}$, then $\phi(n) = n \prod_i (1 - 1/p_i)$;
- $\sum_{d|n} \phi(d) = n$.

We can determine the number of elements whose order is r when r divides $(p^n - 1)$.

Proposition 1.3. *For every integer r such that $r \mid (p^n - 1)$, the number of order r elements in $\mathbb{F}_{p^n}^\times$ is $\phi(r)$. The subgroup of order r in $\mathbb{F}_{p^n}^\times$ is unique.*

Proof. The elements whose order divides r in $\mathbb{F}_{p^n}^\times$ are the roots of $X^r - 1$ belonging to $\mathbb{F}_{p^n}^\times$. Let $a \in \mathbb{F}_{p^n}^\times$ an r order element. The cyclic group generated by a coincides with the set of the roots of $X^r - 1$. In fact, every element of $\langle a \rangle$ is a root of $X^r - 1$ and $\langle a \rangle$ has cardinality equal to r, whereas $X^r - 1$ has r roots at most. Therefore the set of r order elements in $\mathbb{F}_{p^n}^\times$ is the multiplicative group $\langle a \rangle$.

Moreover, by proposition 1.2, $\langle a \rangle$ contains $\phi(r)$ elements whose order is r. This clearly implies that, if an r order element exists in $\mathbb{F}_{p^n}^\times$, then there are exactly $\phi(r)$ such elements. Using the fact that every element's order divides $(p^n - 1)$ and the fact that $\sum_{r|p^n-1} \phi(r) = q - 1$, we get that there are $\phi(r)$ elements whose order is r for every $r \mid (p^n - 1)$. □

We have a direct consequence.

Corollary 1.1. *The multiplicative group $\mathbb{F}_{p^n}^\times$ is cyclic of order $(p^n - 1)$ and there exists $\phi(p^n - 1)$ generators. Every generator is called primitive element.*

Moreover, we need the following to understand how to generate a finite field.

Proposition 1.4. *Every extension of finite fields is simple.*

Proof. Consider $E \supset F$. Then E is a finite subgroup of the multiplicative group of a field, and hence is cyclic. If α generates E as a multiplicative group, then certainly $E = F[\alpha]$. □

1.2. Representation

Let $\mathbb{F}_{p^n}$ a finite field. By corollary 1.1, we can express every element of $\mathbb{F}_{p^n}^\times$ as a generator power. We call this generator α. Therefore, we write $\mathbb{F}_{p^n}^\times = \langle\alpha\rangle$, i.e. $\mathbb{F}_{p^n} = \langle\alpha\rangle \cup \{0\}$.

Moreover, $\mathbb{F}_{p^n}$ can be represented as a vector space whose coefficients are in $\mathbb{F}_p$. We choose the base

$$\{\alpha^0, \alpha^1, \ldots, \alpha^{n-1}\}$$

for $\mathbb{F}_{p^n}$. So that, for every integer k, there exist $c_i \in \mathbb{F}_p$ such that

$$\alpha^k = c_0\alpha^0 + c_1\alpha^1 + \ldots + c_{n-1}\alpha^{n-1}.$$

If we take a nonnegative k, this equation is a polynomial one and this suggests that we have to use the polynomial ring to construct the finite field.

Theorem 1.2. *Let p a prime number and $f(X) \in \mathbb{F}_p[X]$ a degree n irreducible polynomial. The quotient $\mathbb{F}_p[X]/\langle f(X)\rangle$ containing the classes of residues modulo $f(X)$ is a field containing p^n elements.*

Proof. Consider the set F of all the polynomials in $\mathbb{F}_p[X]$ whose degree is at most $(n-1)$. It is a representation of the classes of residues modulo $f(X)$. It is trivial to prove that F is an abelian group with addition modulo $f(X)$ and 0 is the neutral element. Moreover, it can be proven that the product of two elements in F is again in F, 1 is the neutral element and the distributive law is valid.

Therefore, we just need to prove that for every polynomial a different from zero, there exists a polynomial b such that $ab = 1$. This could be done easily by using the Bezout's identity. Since f is irreducible, every polynomial whose degree is at most $(n-1)$ is coprime with f. Thus, there exist two polynomials u and v such that

$$au + fv = 1.$$

We get $au \equiv 1 \pmod f$, i.e. $b \equiv u \pmod f$. □

We can object that it is not guaranteed the existance of a degree n irreducible polynomial whose coefficients are in $\mathbb{F}_p$. The following proposition helps us.

Proposition 1.5. *Let $\mathbb{F}_p$ a finite field. For every positive integer n there exists a degree n irreducible polynomial.*

Proof. From theorem 1.1, the finite field $\mathbb{F}_{p^n}$ exists. By corollary 1.1, the multiplicative group of every finite field is cyclic. If we take a generator α, using proposition 1.4, we have $\mathbb{F}_{p^n} = \mathbb{F}_p[\alpha]$. In particular, the minimal polynomial of α, which is irreducible, must have the degree equal to $[\mathbb{F}_{p^n} : \mathbb{F}_p]$, i.e. n. □

Example 1.1. We show how to construct the field $\mathbb{F}_8$. First of all, we need a degree 3 irreducible polynomial whose coefficients are in $\mathbb{F}_2$. There are 16 degree 3 polynomials whose coefficients are in $\mathbb{F}_2$. We choose $f(X) = 1 + X + X^3$ and it can be easily proved that it is irreducible, in fact $f(0) = f(1) = 1$. We know by theorem 1.2 that the field can be represented as $\mathbb{F}_2[X]/\langle f(X)\rangle$, therefore the elements are polynomials belonging to $\mathbb{F}_2[X]$ whose degree is at most 2.

Usually the notation is simplified using α, a representative for the residue class of X. Using this notation we can represent the elements of $\mathbb{F}_8$ in a double fashion: as α powers or as polynomials. Refer to table 1.1.

Table 1.1: the $\mathbb{F}_8$ elements.

powers	-	α^0	α^1	α^2	α^3	α^4	α^5	α^6
polynomials	0	1	α	α^2	$1+\alpha$	$\alpha+\alpha^2$	$1+\alpha+\alpha^2$	$1+\alpha^2$

We show two computation examples:

- *Addition.* Remember that the field characteristic is 2.

$$(\alpha + \alpha^2) + (1 + \alpha) = 1 + \alpha^2$$

- *Multiplication.*

$$(\alpha + \alpha^2) \cdot (1 + \alpha) = \alpha + \alpha^3 \equiv 1 \pmod{f(X)}$$

Moreover, we could observe that $\mathbb{F}_8^\times$ contains a unique element whose order is 1 and 6 elements whose order is 7, since we respectively have $\phi(1) = 1$ and $\phi(7) = 6$. Therefore, every element different from 0 and 1 is a generator of $\mathbb{F}_8^\times$.

1.3. Polynomial product in cryptography

Many cryptographic algorithms use fields of characteristic 2. It is natural to implement every element of such a field as a bit array. Recall the example 1.1. The underlying field of coefficients is $\mathbb{F}_2$, i.e. the set $\{0, 1\}$. So, addition and multiplication between coefficients in $\mathbb{F}_2$ correspond to XOR (`^`) and AND (`&`) operations for bits respectively. Moreover, recalling table 1.1, we could make a 1-to-1 correspondence between field elements and their representation in computer as bit arrays. The correspondence is given in figure 1.1.

We are going to show a complete example to see how the implementation of a polynomial product algorithm works.

Example 1.2. Suppose you have to multiply two polynomials representing two elements F and G in $\mathbb{F}_8$. They can be represented as

Figure 1.1: the $\mathbb{F}_8$ elements as bit arrays.

polynomials	0	1	α	α^2
bit arrays	0 0 0	1 0 0	0 1 0	0 0 1
polynomials	$1+\alpha$	$\alpha+\alpha^2$	$1+\alpha+\alpha^2$	$1 \quad +\alpha^2$
bit arrays	1 1 0	0 1 1	1 1 1	1 0 1

$$F = f_0 + f_1\alpha + f_2\alpha^2 \qquad \text{and} \qquad G = g_0 + g_1\alpha + g_2\alpha^2.$$

The product is therefore

$$\begin{aligned} &H = F \cdot G = \\ &= f_0g_0 + (f_0g_1 + f_1g_0)\alpha + (f_0g_2 + f_1g_1 + f_2g_0)\alpha^2 + (f_1g_2 + f_2g_1)\alpha^3 + (f_2g_2)\alpha^4. \end{aligned} \tag{1.1}$$

Note that at this point we have to reduce the polynomial product in order to express it as a degree 2 one because of the theory we have developed so far. We discard this reduction step in this dissertation.

To allow a fast check of how many operations are needed to get the result we present the so called Straight Line Program (SLP), using both mathematical and computer science notation.

Mathematics	Computer Science
$h_0 = f_0g_0$	$\mathtt{h_0 = f_0 \,\&\, g_0}$
$t_0 = f_0g_1$	$\mathtt{t_0 = f_0 \,\&\, g_1}$
$t_1 = f_1g_0$	$\mathtt{t_1 = f_1 \,\&\, g_0}$
$h_1 = t_0 + t_1$	$\mathtt{h_1 = t_0 \,\hat{}\, t_1}$
$t_2 = f_0g_2$	$\mathtt{t_2 = f_0 \,\&\, g_2}$
$t_3 = f_1g_1$	$\mathtt{t_3 = f_1 \,\&\, g_1}$
$t_4 = f_2g_0$	$\mathtt{t_4 = f_2 \,\&\, g_0}$
$t_5 = t_2 + t_3$	$\mathtt{t_5 = t_2 \,\hat{}\, t_3}$
$h_2 = t_5 + t_4$	$\mathtt{h_2 = t_5 \,\hat{}\, t_4}$
$t_6 = f_1g_2$	$\mathtt{t_6 = f_1 \,\&\, g_2}$
$t_7 = f_2g_1$	$\mathtt{t_7 = f_2 \,\&\, g_1}$
$h_3 = t_6 + t_7$	$\mathtt{h_3 = t_6 \,\hat{}\, t_7}$
$h_4 = f_2g_2$	$\mathtt{h_4 = f_2 \,\&\, g_2}$

Note that we have used variables t_i to store intermediate results. We can see that

$$H = h_0 + h_1\alpha + h_2\alpha^2 + h_3\alpha^3 + h_4\alpha^4$$

is exactly (1.1), i.e. the result we were searching for.

The algorithm we have presented is the classical *School-book*. In the next chapter, after a brief recall of the state of art, we will present some new algorithms which can improve the state of art. We will prefer the mathematical notation.

2. Improvements to the state of art

In this chapter we are going to see how to devise some improved polynomial multiplication algorithms. Although most of the algorithms could be used for a generic field, we will restrict our attention to polynomials whose coefficients are in $\mathbb{F}_2$. Moreover, we will consider multiplications whose factors have the same degree. This is done for one principal reason: avoid side channel attacks that exploit energy consumption due to different lengths of the inputs. Doing this, we perform always the same amount of operations with almost no differences in energy consumption when executing.

We are going to see that the improvements affect not only degrees they are designed for, but also the ones that rely part of the algorithm on the improved degrees. The improvements are not only about the number of operations but also in depth, allowing an high level of parallelization.

2.1. Algorithms for cryptographic applications

We are going to review the cornerstones when speaking about polynomial multiplication. Some of them will be clear at once, some others need a deeper investigation to be understood.

2.1.1. School-book

The first algorithm we are going to examine is the well known *School-book* one. Pay attention to the technique used for enumerating steps and computing the number of operations since we will implicitly use the same technique for the subsequent algorithms.

Given two n-bit polynomials

$$\begin{aligned} F &= f_0 + f_1 t + \ldots + f_{n-1} t^{n-1} + f_n t^n \\ \text{and} \quad G &= g_0 + g_1 t + \ldots + g_{n-1} t^{n-1} + g_n t^n \end{aligned}$$

let

$$\begin{aligned} F_{n-1} &= f_0 + f_1 t + \cdots + f_{n-1} t^{n-1} \\ \text{and} \quad G_{n-1} &= g_0 + g_1 t + \cdots + g_{n-1} t^{n-1}. \end{aligned}$$

We can rewrite F and G as

$$\begin{aligned} F &= F_{n-1} + f_n t^n \\ \text{and} \quad G &= G_{n-1} + g_n t^n \end{aligned}$$

and resume the *School-book* algorithm as

$$\begin{aligned} F \cdot G &= (F_{n-1} + f_n t^n) \cdot (G_{n-1} + g_n t^n) \\ &= F_{n-1}G_{n-1} + F_{n-1}g_n t^n + G_{n-1}f_n t^n + f_n g_n t^{2n} \end{aligned} \tag{2.1}$$

The steps needed to obtain the result from (2.1) multiplication algorithm are:

1. Multiply F_{n-1} by G_{n-1}. Note that this is a recursive step and it is now clear what we mean when we say that an algorithm for a specific degree take the advantage from the lower ones. We simply denote its cost, i.e. the number of operations, as $M(n)$.

2. Multiply F_{n-1} by g_n. As it is a multiplication of a polynomial by a single coefficient, it takes n operations.

3. Multiply G_{n-1} by f_n. Same as 2.

4. Multiply f_n by g_n. Note that this is a multiplication between coefficients, therefore it takes just one operation.

5. Add $F_{n-1}G_{n-1}$, which is a degree $2n-2$ polynomial and $F_{n-1}g_n t^n$, which is a degree $2n-1$ polynomial. It takes $n-1$ operations. Note that the multiplication by t^n can be considered just a shift of $F_{n-1}g_n$, therefore we do not take into account the shift.

6. Add the result of the previous step to $G_{n-1}f_n t^n$. It takes now n operations. Note that in the previous step we had two polynomials of degree $2n-2$ and degree $2n-1$, instead, we have here two polynomials: the first one whose degree is $2n-1$, i.e. the result coming from the previous step and the second one, $G_{n-1}f_n t^n$, whose degree is $2n-1$.

7. We eventually add $f_n g_n$ to the result coming from the previous step, but, since the power of $f_n g_n$ is $2n$ and the degree of the previuos result is $2n-1$, the operation has no cost.

Summing up all costs, we get the recursion formula for the *School-book* algorithm.

$$M(n+1) \leq M(n) + 4n. \tag{2.2}$$

As we can see, $M(n+1)$, the complexity of the multiplication for degree $n+1$, is computed using $M(n)$, the one for degree n. This clearly shows that any advantage at any degree could become a chain of improvements for higher degrees.

Moreover, let $M(1) = 1$ and consider just the *School-book*. We can write a closed form

$$M(n) \leq 2n^2 - 2n + 1 \tag{2.3}$$

allowing us to predict an upper bound for the algorithm. We will review the upper bounds in section 2.3 and improve the state of art in section 2.4.

2.1.2. Karatsuba

We are now going to review the Karatsuba algorithm [43]. It was first published in 1963, but it is useful nowadays, in fact, some degrees like 6 or 8 still use this improvement [19].

Given two $2n$-bit polynomials

$$\begin{aligned} F &= f_0 + f_1 t + \ldots + f_{2n-1} t^{2n-1} \\ \text{and} \quad G &= g_0 + g_1 t + \ldots + g_{2n-1} t^{2n-1} \end{aligned}$$

let

$$\begin{aligned} F_0 &= f_0 + f_1 t + \ldots + f_{n-1} t^{n-1} \\ \text{and} \quad F_1 &= f_n + f_{n+1} t + \ldots + f_{2n-1} t^{n-1} \end{aligned}$$

doing the same for G, and rewrite F and G as

$$\begin{aligned} F &= F_0 + F_1 t^n \\ \text{and} \quad G &= G_0 + G_1 t^n \end{aligned}$$

We can write the Karatsuba algorithm as:

$$\begin{aligned} F \cdot G &= (F_0 + t^n F_1) \cdot (G_0 + t^n G_1) \\ &= (1 + t^n) F_0 G_0 + t^n (F_0 + F_1)(G_0 + G_1) + (t^n + t^{2n}) F_1 G_1. \end{aligned} \tag{2.4}$$

We repeat what we have done for the *School-book*.

1. Multiply F_0 by G_0, the cost is $M(n)$.
2. Add $F_0 G_0$ to $t^n F_0 G_0$. Remember that $t^n F_0 G_0$ is just $F_0 G_0$ shifted by n positions. Therefore, we get $A_1 = (1 + t^n) F_0 G_0$ with a cost of $n - 1$ operations.
3. Add F_0 to F_1 and G_0 to G_1. We get $F_0 + F_1$ and $G_0 + G_1$ using $2n$ operation.
4. Multiply $(F_0 + F_1)$ by $(G_0 + G_1)$ using $M(n)$ operations.
5. Multiply F_1 by G_1 using $M(n)$ operations again.
6. Add $t^n F_1 G_1$ to $t^{2n} F_1 G_1$. This is the same type of operation performed in step 2, therefore we get $A_2 = (t^n + t^{2n}) F_1 G_1$ with a cost of $n - 1$ operations.
7. Add A_1 to $t^n (F_0 + F_1)(G_0 + G_1)$. We get $A_3 = A_1 + t^n (F_0 + F_1)(G_0 + G_1)$ using $2n - 1$ operations.
8. The final addition $A_3 + A_2$ uses $2n - 1$ operations.

Summing all costs, we get

$$M(2n) \leq 3M(n) + 8n - 4. \tag{2.5}$$

In section 2.3, we are going to see some results that allow us to resolve this recursion in special cases and get closed form as we have done for the *School-book* inequality (2.3).

The Karatsuba algorithm was the first improvement regarding polynomial multiplication, but we had waited until 2008 for having another improvement. Daniel J. Bernstein was the author and firstly wrote his result in [9] and formally announced the idea in [8]. His improvement, which we will call *Refined-Karatsuba* is based on a little manipulation of the equation (2.4). Moreover, we do a little generalization which could have been done also for the original Karatsuba.

Given two $(n+k)$-bit polynomials

$$\begin{aligned} F &= f_0 + f_1 t + \ldots + f_{n+k-1} t^{n+k-1} \\ \text{and} \quad G &= g_0 + g_1 t + \ldots + g_{n+k-1} t^{n+k-1} \end{aligned}$$

having $n/2 \leq k \leq n$, let

$$\begin{aligned} F_0 &= f_0 + f_1 t + \ldots + f_{n-1} t^{n-1} \\ \text{and} \quad F_1 &= f_n + f_{n+1} t + \ldots + f_{n+k-1} t^{k-1} \end{aligned}$$

doing the same for G, and rewrite F and G as

$$\begin{aligned} F &= F_0 + F_1 t^n \\ \text{and} \quad G &= G_0 + G_1 t^n \end{aligned}$$

The *Refined-Karatsuba* can be written as follows.

$$\begin{aligned} F \cdot G &= (F_0 + t^n F_1)(G_0 + t^n G_1) \\ &= (1 + t^n) F_0 G_0 + t^n (F_0 + F_1)(G_0 + G_1) + (t^n + t^{2n}) F_1 G_1 \\ &= (1 + t^n) F_0 G_0 + t^n (F_0 + F_1)(G_0 + G_1) + (1 + t^n) t^n F_1 G_1 \\ &= (1 + t^n)(F_0 G_0 + t^n F_1 G_1) + t^n (F_0 + F_1)(G_0 + G_1) \end{aligned}$$

We describe the estimate of the cost of the algorithm in a concise way as we have described all the details on how it can be devised along the *School-book* and *Karastuba*. Each item in the following shows cost and operations.

1. $M(n) + M(k)$: multiplications $F_0 G_0$ and $F_1 G_1$, respectively
2. $n - 1$: addition $A_1 = F_0 G_0 + t^n F_1 G_1$
3. $2k - 1$: addition $A_2 = (1 + t^n) A_1$
4. $2k$: additions $F_0 + F_1$, $G_0 + G_1$
5. $M(n)$: multiplication $(F_0 + F_1)(G_0 + G_1)$
6. $2n - 1$: addition $A_3 = A_2 + t^n (F_0 + F_1)(G_0 + G_1)$

The total cost of the algorithm is

$$M(n+k) \leq 2M(n) + M(k) + 3n + 4k - 3, \quad \frac{n}{2} \leq k \leq n. \tag{2.6}$$

If we take $k = n$, it is very clear that this refined version improves the original *Karatsuba* estimate (2.5).

2.1.3. Bernstein

In [8], Bernstein further improves the *Refined-Karatsuba* algorithm, presenting the so called *Two-level Seven-way*. Basically, he just consider to split the factors in four parts and apply three times the *Refined-Karatsuba* factoring out $(1+t^n)$.

Formally, given two $(3n+k)$-bits polynomials

$$\begin{aligned} F &= f_0 + f_1 t + \ldots + f_{3n+k-1} t^{3n+k-1} \\ \text{and} \quad G &= g_0 + g_1 t + \ldots + g_{3n+k-1} t^{3n+k-1} \end{aligned}$$

having $n/2 \leq k \leq n$, let

$$\begin{aligned} F_0 &= f_0 + f_1 t + \ldots + f_{n-1} t^{n-1}, \\ F_1 &= f_n + f_{n+1} t + \ldots + f_{2n-1} t^{n-1}, \\ F_2 &= f_{2n} + f_{2n+1} t + \ldots + f_{3n-1} t^{n-1}, \\ \text{and} \quad F_3 &= f_{3n} + f_{3n+1} t + \ldots + f_{3n+k-1} t^{k-1} \end{aligned}$$

doing the same for G, and rewrite F and G as

$$\begin{aligned} F &= F_0 + F_1 t^n + F_2 t^{2n} + F_3 t^{3n} \\ \text{and} \quad G &= G_0 + G_1 t^n + G_2 t^{2n} + G_3 t^{3n}. \end{aligned}$$

The *Two-level Seven-way* can be written as follows.

$$\begin{aligned} F \cdot G &= (F_0 + F_1 t^n + F_2 t^{2n} + F_3 t^{3n})(G_0 + G_1 t^n + G_2 t^{2n} + G_3 t^{3n}) \\ &= (1 + t^{2n})\Big((1 + t^n)(F_0 G_0 + t^n F_1 G_1 + t^{2n} F_2 G_2 + t^{3n} F_3 G_3) \\ &\qquad + t^n (F_0 + F_1)(G_0 + G_1) + t^{3n} (F_2 + F_3)(G_2 + G_3)\Big) \\ &\quad + t^{2n}\Big(F_0 + F_2 + t^n (F_1 + F_3)\Big)\Big(G_0 + G_2 + t^n (G_1 + G_3)\Big). \end{aligned}$$

The cost evaluation is as follows.

1. $3M(n)$: multiplications $F_0 G_0$, $F_1 G_1$, $F_2 G_2$
2. $M(k)$: multiplication $F_3 G_3$
3. $3(n-1)$: additions $A_1 = F_0 G_0 + t^n F_1 G_1 + t^{2n} F_2 G_2 + t^{3n} F_3 G_3$
4. $2n + 2k - 1$: addition $(1 + t^n) A_1$
5. $2n + M(n)$: multiplication $A_2 = (F_0 + F_1)(G_0 + G_1)$
6. $2k + M(n)$: multiplication $A_3 = (F_2 + F_3)(G_2 + G_3)$
7. $4n - 2$: additions $A_4 = (1 + t^n) A_1 + t^n A_2 + t^{3n} A_3$
8. $2n + 2k + M(2n)$: multiplication $A_5 = (F_0 + F_2 + t^n (F_1 + F_3))(G_0 + G_2 + t^n (G_1 + G_3))$
9. $6n + 2k - 2$: additions $(1 + t^{2n}) A_4 + t^{2n} A_5$

The total cost of the algorithm is

$$M(3n + k) \leq M(2n) + 5M(n) + M(k) + 19n + 8k - 8, \quad \frac{n}{2} \leq k \leq n.$$

2.1.4. Find – Peralta

One of the most recent techniques (early 2018) is from Find and Peralta [35]. In this work, the authors devise a method to derive the so called *Karatsuba-like recurrences*. Although this work contains a generalization of the methodology it gives, the same technique was designed for a particular case some years before by Cenk, Hasan and Negre in [20], which is based on [83]. In fact, in [19, Section 2] the *Karatsuba-like improved 3-way split algorithm* is recalled and it could be checked that the recurrence (7) in [19] is exactly the one in [35, Section 4.3].

We are briefly going to review the *Karatsuba-like* algorithm by Find and Peralta and give an example.

Given two kn-bits polynomials

$$\begin{aligned} F &= f_0 + f_1 t + \ldots + f_{kn-1} t^{kn-1} \\ \text{and} \quad G &= g_0 + g_1 t + \ldots + g_{kn-1} t^{kn-1} \end{aligned}$$

let

$$\begin{aligned} F_0 &= f_0 + f_1 t + \ldots + f_{n-1} t^{n-1}, \\ F_1 &= f_n + f_{n+1} t + \ldots + f_{2n-1} t^{n-1}, \\ &\vdots \\ \text{and} \quad F_{k-1} &= f_{(k-1)n} + f_{(k-1)n+1} t + \ldots + f_{kn-1} t^{n-1}, \end{aligned}$$

doing the same for G, and rewrite F and G as

$$\begin{aligned} F &= F_0 + F_1 t^n + \ldots + F_{k-1} t^{(k-1)n} \\ \text{and} \quad G &= G_0 + G_1 t^n + \ldots + G_{k-1} t^{(k-1)n}. \end{aligned}$$

The *Karatsuba-like* algorithm goes as follows.

- First of all, the product $F \cdot G$ can be written as

$$F \cdot G = \sum_{i=0}^{2k-2} U_i t^{in}$$

  where $U_i = \sum_{a+b=i} F_a \cdot G_b$.

- This second step is very important as it fully determines all the subsequent elements and complexities. Let $\mathbf{F} = (F_0, F_1, \ldots, F_{k-1})$, $\mathbf{G} = (G_0, G_1, \ldots, G_{k-1})$ and $\mathbf{U} = (U_0, U_1, \ldots, U_{2k-2})$. The heart of the algorithm is:

$$\mathbf{U} = R \cdot [(T \cdot \mathbf{F}) \circ (T \cdot \mathbf{G})], \tag{2.7}$$

  where $\circ$ is the Hadamard product[1] and T is a matrix having a requirement: the elements of the Hadamard product $(T \cdot \mathbf{F}) \circ (T \cdot \mathbf{G})$ span the target bilinear forms in $\mathbf{U}$. R is derived accordingly to the matrix T.

  T is called top matrix and the heuristic from [12] can be used to compute it. R is called main matrix and it can be derived using the heuristic for small low-depth XOR circuits from [14].

[1] Let two vectors $v = (v_0, \ldots, v_{n-1})$ and $w = (w_0, \ldots, w_{n-1})$, the Hadamard product is simply the vector $v \circ w = (v_0 w_0, \ldots, v_{n-1} w_{n-1})$.

- As one can notice by computing the previous step, some parts of the U_i polynomials overlaps and a clever use of the Hadamard product can avoid a waste of operations. Thus, let

$$\mathbf{P} = (T \cdot \mathbf{F}) \circ (T \cdot \mathbf{G}) = (P_0, \dots, P_{\tau-1}),$$

we do not calculate $R \cdot \mathbf{P}$ as in the equation (2.7), instead, we split the generic polynomial P_i of $\mathbf{P}$ as

$$P_i = (L(P_i), M(P_i), H(P_i)),$$

where $L(P_i)$, $M(P_i)$, $H(P_i)$ are respectively the first $(n-1)$ bits, the middle bit and the last $(n-1)$ bits of P_i. This could be done since a generic P_i is the product $\sum_{i \in S} F_i \cdot \sum_{i \in S} G_i$ for some set S of indices, therefore the number of coefficients of P_i is $2n-1$. At this point, we can perform the products

$$\begin{pmatrix} H[n-1] \\ H[2n-1] \\ \vdots \\ H[(2k-1)n-1] \end{pmatrix} = R \cdot \begin{pmatrix} M(P_0) \\ M(P_1) \\ \vdots \\ M(P_{\tau-1}) \end{pmatrix} \tag{2.8}$$

and

$$\begin{pmatrix} H[0 \,..\, n-2] \\ H[n \,..\, 2n-2] \\ \vdots \\ H[(2k-1)n \,..\, 2kn-2] \end{pmatrix} = E \cdot \begin{pmatrix} L(P_0) \\ \vdots \\ L(P_{\tau-1}) \\ H(P_0) \\ \vdots \\ H(P_{\tau-1}) \end{pmatrix} \tag{2.9}$$

where

$$E = \begin{pmatrix} R_0 & 0 \\ R_1 & R_0 \\ \vdots & \vdots \\ R_{2k-2} & R_{2k-3} \\ 0 & R_{2k-2} \end{pmatrix}$$

and R_i are the rows of the matrix R.

- It's not difficult to show that, sorting the vector H from the previous step according to the indices given by (2.8) and (2.9), we get the coefficients of the product $F \cdot G$.

The computation of the generic recursion representing the estimate of the cost of the multiplication algorithm between two kn-bit polynomials goes as follows.

1. The first important addendum of the cost is the number of operation needed for the XOR circuit represented by matrix T. We generically call it $s(T)$. Remember that we need 2 circuit, i.e. $(T \cdot \mathbf{F})$ and $(T \cdot \mathbf{G})$. Remember also that each element of $\mathbf{F}$ (same for $\mathbf{G}$) is represented by n bits, so, for this first step we need $2n \cdot s(T)$.

2. The second step is the Hadamard product $(T \cdot \mathbf{F}) \cdot (T \cdot \mathbf{G})$. Every element in both vectors $(T \cdot \mathbf{F})$ and $(T \cdot \mathbf{G})$ is an n-bit polynomial, thus, we simply denote this cost as $n_p \cdot M(n)$. A rough estimate of the number of the elements in a vector like $(T \cdot \mathbf{F})$ is trivially $n_p < 2^k$ since 0 cannot be a result.

3. The third step are products in (2.8) and (2.9). Regarding (2.8) there are no additional costs other than the complexity of the circuit as the $M(P_i)$ are one single bit. We denote $s(R)$ the cost for circuit R. Regarding (2.9), remember that each $L(P_i)$ and each $H(P_i)$ is a $(n-1)$-bit vector, so, in order to the estimate the cost, we need to multiply the complexity of the XOR circuit $s(E)$ by the size of the vector's elements. The cost is then $(n-1) \cdot s(E)$.

Summing up, we have the recursion for the cost of *Karatsuba-like* algorithm:

$$M(kn) \leq n_p \cdot M(n) + 2n \cdot s(T) + s(R) + (n-1) \cdot s(E).$$

Note that, likewise the algorithms so far, we can consider $(kn + c)$-bit polynomials and compute the recursion accordingly. Of course, we need $n/2 \leq c \leq n$. For the sake of simplicity, we have just considered kn-bit polynomials.

Recursions with explicit parameters are:

$$\begin{aligned}
&M(2n) \leq 3M(n) + 7n - 3 \quad (\textit{Refined-Karatsuba}) \\
&M(3n) \leq 6M(n) + 18n - 6 \\
&M(4n) \leq 9M(n) + 34n - 12 \\
&M(5n) \leq 13M(n) + 54n - 19 \\
&M(6n) \leq 17M(n) + 85n - 29 \\
&M(7n) \leq 22M(n) + 107n - 34
\end{aligned} \tag{2.10}$$

There is the possibility to develop better algorithms by considering $8n$-bit polynomials or even an higher level of split, but the complexity of the E circuit is beyond computational resources.

Example 2.1. We show how the *Karatsuba-like* algorithm acts when $k = 2$. Given two $2n$-bit polynomials

$$\begin{aligned}
F &= f_0 + f_1 t + \ldots + f_{2n-1} t^{2n-1} \\
\text{and} \quad G &= g_0 + g_1 t + \ldots + g_{2n-1} t^{2n-1}
\end{aligned}$$

let

$$\begin{aligned}
F_0 &= f_0 + f_1 t + \ldots + f_{n-1} t^{n-1}, \\
F_1 &= f_n + f_{n+1} t + \ldots + f_{2n-1} t^{n-1},
\end{aligned}$$

doing the same for G, we have

$$H = F \cdot G = (F_0 + F_1 t^n) \cdot (G_0 + G_1 t^n).$$

In order to show how the algorithm works, we are going to follow exactly the steps of the explanation of the algorithm.

- First of all we write the product

$$F \cdot G = \sum_{i=0}^{2} U_i t^{in}$$

where $U_i = \sum_{a+b=i} F_a \cdot G_b$.

- Let $\mathbf{F} = (F_0, F_1)$ and $\mathbf{G} = (G_0, G_1)$ and $\mathbf{U} = (U_0, U_1, U_2)$. By using the heuristics from [12], we get

$$T = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}$$

and, by using the heuristic from [14], we get

$$R = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}.$$

At this point we have

$$\mathbf{U} = R \cdot [(T \cdot \mathbf{F}) \circ (T \cdot \mathbf{G})],$$

in particular: $U_0 = P_0$, $U_1 = P_0 + P_1 + P_2$ and $U_2 = P_2$.

- Let

$$\mathbf{P} = (T \cdot \mathbf{F}) \circ (T \cdot \mathbf{G}) = (P_0, P_1, P_2),$$

and split the P_i as

$$\begin{aligned} P_0 &= (L(P_0), M(P_0), H(P_0)), \\ P_1 &= (L(P_1), M(P_1), H(P_1)), \\ P_2 &= (L(P_2), M(P_2), H(P_2)). \end{aligned}$$

Now, according to the products (2.8) and (2.9), compute:

$$\begin{pmatrix} H[n-1] \\ H[2n-1] \\ H[3n-1] \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} M(P_0) \\ M(P_1) \\ M(P_3) \end{pmatrix} \tag{2.11}$$

and

$$\begin{pmatrix} H[0 \,..\, n-2] \\ H[0 \,..\, 2n-2] \\ H[3n \,..\, 4n-2] \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} L(P_0) \\ L(P_1) \\ L(P_2) \\ H(P_0) \\ H(P_1) \\ H(P_2) \end{pmatrix} \tag{2.12}$$

- By sorting the vector H according to the indices, we obtain the product of F and G.

We perform also the estimate of the cost likewise its explanation.

1. The first cost is the complexity of the circuit represented by the matrix T. It is trivial to see that just the second line, which represent the addition between F_0 and F_1, needs to be computed. Therefore the cost is $2n$ since F_0 and F_1 have n bits and the same applies for G_0 and G_1.

2. The second cost is the Hadamard product. Since T has 3 rows and each of the F_i and G_i have n-bits, it is straightforward that the cost is $3M(n)$.

3. The third cost are the complexities of (2.11) and (2.12). Regarding the former, it is trivial to see that the only row that needs to be computed is the second one and we cannot use less than two operations, therefore the cost is 2 since every $M(P_i)$ has one bit. Regarding the latter, we can use the heuristic in [12] and retrive that we need $5 \cdot (n-1)$ operations since every $L(P_i)$ and $H(P_i)$ has $(n-1)$ bits.

Summing all the costs, we obtain

$$M(2n) \leq 2n + 3M(n) + 2 + 5(n-1)$$

which is

$$M(2n) \leq 3M(n) + 7n - 3,$$

and coincides with the *Refined-Karatsuba* (2.6) proposed by Bernstein.

Giving this example, we have finished the review of the state of art and we are going to see how to improve it.

2.2. Improvements of practical interest

The first improvement we are going to see is about algorithms of practical interests. In fact, we are going to recall the *Two-level Seven-Way* and expand it. We are going to see in the section 2.3 that the benefits of this improvement are not only for the effective number of operations but also in estimates of upper bounds.

We now formally present the so called *Three-level recursion*, first appeared in [29]. Given two $(7n+k)$-bit polynomials

$$\begin{aligned} F &= f_0 + f_1 t + \ldots + f_{7n+k-1} t^{7n+k-1} \\ \text{and} \quad G &= g_0 + g_1 t + \ldots + g_{7n+k-1} t^{7n+k-1} \end{aligned}$$

having $n/2 \leq k \leq n$, let

$$\begin{aligned} F_0 &= f_0 + f_1 t + \ldots + f_{n-1} t^{n-1}, \\ F_1 &= f_n + f_{n+1} t + \ldots + f_{2n-1} t^{n-1}, \\ &\vdots \\ F_6 &= f_{6n} + f_{6n+1} t + \ldots + f_{7n-1} t^{n-1}, \\ \text{and} \quad F_7 &= f_{7n} + f_{7n+1} t + \ldots + f_{7n+k-1} t^{k-1} \end{aligned}$$

doing the same for G, and rewrite F and G as

$$F = F_0 + F_1 t^n + \ldots + F_7 t^{7n}$$
$$\text{and} \quad G = G_0 + G_1 t^n + \ldots + G_7 t^{7n}.$$

The *Three-level* can be written as follows.

$$\begin{aligned} F \cdot G &= \left(\sum_{i=0}^{7} F_i t^{in}\right)\left(\sum_{i=0}^{7} G_i t^{in}\right) \\ &= \left(\sum_{i=0}^{3} t^{in} F_i + t^{4n} \sum_{i=0}^{3} t^{in} F_{i+4}\right)\left(\sum_{i=0}^{3} t^{in} G_i + t^{4n} \sum_{i=0}^{3} t^{in} G_{i+4}\right) \end{aligned}$$

As we can see, in this first step, we simply re-organize the eight parts of F in two sums miming the first step of (2.4) and do the same for G. At this point we apply the *Refined-Karatsuba* algorithm with the following equalities.

$$\begin{aligned} F_{00} &= \sum_{i=0}^{3} t^{in} F_i \qquad F_{10} = \sum_{i=0}^{3} t^{in} F_{i+4} \\ G_{00} &= \sum_{i=0}^{3} t^{in} G_i \qquad G_{10} = \sum_{i=0}^{3} t^{in} G_{i+4} \end{aligned} \tag{2.13}$$

Note that the double subscript in (2.13) has the following meaning: the first subscript refers to the *Karatsuba* algorithm, the second subscript is the level of the recursion. We obtain

$$\begin{aligned} F \cdot G &= (F_{00} + t^{4n} F_{10})(G_{00} + t^{4n} G_{10}) \\ &= (1 + t^{4n}) \underbrace{(F_{00} G_{00} + t^{4n} F_{10} G_{10})}_{T_{00}} + t^{4n} \underbrace{(F_{00} + F_{10})(G_{00} + G_{10})}_{T_{10}}. \end{aligned} \tag{2.14}$$

Regarding term T_{10} in (2.14), we simply do some algebraic manipulation after recalling the original terms.

$$\begin{aligned} T_{10} &= \left(\sum_{i=0}^{3} t^{in} F_i + \sum_{i=0}^{3} t^{in} F_{i+4}\right)\left(\sum_{i=0}^{3} t^{in} G_i + \sum_{i=0}^{3} t^{in} G_{i+4}\right) \\ &= \left(\sum_{i=0}^{3} t^{in} (F_i + F_{i+4})\right)\left(\sum_{i=0}^{3} t^{in} (G_i + G_{i+4})\right) \end{aligned}$$

From now on, we are going to take care of T_{00}, since the remaining parts in (2.14) will not change. Recall what is T_{00}.

$$T_{00} = \underbrace{\left(\sum_{i=0}^{3} t^{in} F_i\right)\left(\sum_{i=0}^{3} t^{in} G_i\right)}_{T_{01}} + t^{4n} \underbrace{\left(\sum_{i=0}^{3} t^{in} F_{i+4}\right)\left(\sum_{i=0}^{3} t^{in} G_{i+4}\right)}_{T_{11}}$$

We explicitly write what are T_{01} and T_{11}, highlighting the power $\mathbf{t^{4n}}$ in order to better understand the transformation.

$$\begin{aligned} T_{01} &= ((F_0 + t^n F_1) + (F_2 + t^n F_3) t^{2n})((G_0 + t^n G_1) + (G_2 + t^n G_3) t^{2n}) \\ T_{11} &= \mathbf{t^{4n}}((F_4 + t^n F_5) + (F_6 + t^n F_7) t^{2n})((G_4 + t^n G_5) + (G_6 + t^n G_7) t^{2n}) \end{aligned}$$

Note that we can apply the *Refined-Karatsuba* once again on T_{01} and T_{11}, getting the following formulae.

$$\begin{aligned}
T_{01} &= (1+t^{2n})\Big((F_0+t^nF_1)(G_0+t^nG_1)+t^{2n}(F_2+t^nF_3)(G_2+t^nG_3)\Big)\\
&\quad + t^{2n}(F_0+F_2+(F_1+F_3)t^n)(G_0+G_2+(G_1+G_3)t^n)\\
T_{11} &= \mathbf{t^{4n}}\Big[(1+t^{2n})\Big((F_4+t^nF_5)(G_4+t^nG_5)+t^{2n}(F_6+t^nF_7)(G_6+t^nG_7)\Big)\\
&\quad + t^{2n}(F_4+F_6+(F_5+F_7)t^n)(G_4+G_6+(G_5+G_7)t^n)\Big]\\
&= (1+t^{2n})\Big(\mathbf{t^{4n}}(F_4+t^nF_5)(G_4+t^nG_5)+\mathbf{t^{6n}}(F_6+t^nF_7)(G_6+t^nG_7)\Big)\\
&\quad + \mathbf{t^{6n}}(F_4+F_6+(F_5+F_7)t^n)(G_4+G_6+(G_5+G_7)t^n)
\end{aligned}$$

We now re-write T_{00} factoring out $(1+t^{2n})$.

$$\begin{aligned}
T_{00} = (1+t^{2n})\Big(&\underbrace{(F_0+t^nF_1)(G_0+t^nG_1)}_{T_{02}}+t^{2n}\underbrace{(F_2+t^nF_3)(G_2+t^nG_3)}_{T_{12}}\\
&+\mathbf{t^{4n}}\underbrace{(F_4+t^nF_5)(G_4+t^nG_5)}_{T_{22}}+\mathbf{t^{6n}}\underbrace{(F_6+t^nF_7)(G_6+t^nG_7)}_{T_{32}}\Big)\\
&+t^{2n}(F_0+F_2+(F_1+F_3)t^n)(G_0+G_2+(G_1+G_3)t^n)\\
&+\mathbf{t^{6n}}(F_4+F_6+(F_5+F_7)t^n)(G_4+G_6+(G_5+G_7)t^n)
\end{aligned}$$

Finally, as could be easily noted, every T_{i2} can be computed using again *Refined-Karatsuba*.

$$\begin{aligned}
T_{02} &= (1+t^n)(F_0G_0+t^nF_1G_1)+t^n(F_0+F_1)(G_0+G_1)\\
T_{12} &= \mathbf{t^{2n}}\Big((1+t^n)(F_2G_2+t^nF_3G_3)+t^n(F_2+F_3)(G_2+G_3)\Big)\\
T_{22} &= \mathbf{t^{4n}}\Big((1+t^n)(F_4G_4+t^nF_5G_5)+t^n(F_4+F_5)(G_4+G_5)\Big)\\
T_{32} &= \mathbf{t^{6n}}\Big((1+t^n)(F_6G_6+t^nF_7G_7)+t^n(F_6+F_7)(G_6+G_7)\Big)
\end{aligned}$$

Doing again a simple algebraic manipulation, we can factor out $(1+t^n)$ from every equation. This will be very useful to write the final formula.

$$\begin{aligned}
T_{02} &= (1+t^n)(F_0G_0+t^nF_1G_1)+t^n(F_0+F_1)(G_0+G_1)\\
T_{12} &= (1+t^n)(\mathbf{t^{2n}}F_2G_2+\mathbf{t^{3n}}F_3G_3)+\mathbf{t^{3n}}(F_2+F_3)(G_2+G_3)\\
T_{22} &= (1+t^n)(\mathbf{t^{4n}}F_4G_4+\mathbf{t^{5n}}F_5G_5)+\mathbf{t^{5n}}(F_4+F_5)(G_4+G_5)\\
T_{32} &= (1+t^n)(\mathbf{t^{6n}}F_2G_2+\mathbf{t^{7n}}F_3G_3)+\mathbf{t^{7n}}(F_2+F_3)(G_2+G_3)
\end{aligned}$$

We can now write the final formula for the *Three-level* agorithm.

$$
\begin{aligned}
F \cdot G = (1+t^{4n})\Bigg\{ & (1+t^{2n})\Bigg[(1+t^{n})\left(\sum_{i=0}^{7} t^{in} F_i G_i\right) \\
& \qquad + \sum_{j=0}^{3} t^{(2j+1)n}(F_{2j}+F_{2j+1})(G_{2j}+G_{2j+1})\Bigg] \\
& + t^{2n}(F_0+F_2+(F_1+F_3)t^n)(G_0+G_2+(G_1+G_3)t^n) \\
& + t^{6n}(F_4+F_6+(F_5+F_7)t^n)(G_4+G_6+(G_5+G_7)t^n)\Bigg\} \\
& + t^{4n}\left(\sum_{i=0}^{3} t^{in}(F_i+F_{i+4})\right)\left(\sum_{i=0}^{3} t^{in}(G_i+G_{i+4})\right)
\end{aligned}
\tag{2.15}
$$

If one follows carefully the second subscript of the T terms during all over the explanation of the algorithm, it can be checked that there are three levels in which *Refined-Karatsuba* is applied, this clarifies the name of the algorithm. The cost evaluation is as follows.

1. $7M(n)$: multiplication F_iG_i, for $i = 0, \ldots, 6$
2. $M(k)$: multiplication F_7 by G_7
3. $7(n-1)$: addition $A_1 = \sum_{i=0}^{7} t^{in} F_i G_i$
4. $6n + 2k - 1$: addition $A_2 = (1+t^n)A_1$
5. $3(2n + M(n))$: multiplication $(F_{2j} + F_{2j+1})(G_{2j} + G_{2j+1})$, for $j = 0, 1, 2$
6. $2k + M(n)$: multiplication $(F_6 + F_7)(G_6 + G_7)$
7. $4(2n-1)$: addition $A_3 = A_2 + \sum_{j=0}^{3} t^{(2j+1)n}(F_{2j} + F_{2j+1})(G_{2j} + G_{2j+1})$
8. $6n + 2k - 1$: addition $A_4 = (1+t^{2n})A_3$
9. $4n + M(2n)$: multiplication
   $A_5 = (F_0 + F_2 + (F_1 + F_3)t^n)(G_0 + G_2 + (G_1 + G_3)t^n)$
10. $2n + 2k + M(2n)$: multiplication
    $A_6 = (F_4 + F_6 + (F_5 + F_7)t^n)(G_4 + G_6 + (G_5 + G_7)t^n)$
11. $2(4n-1)$: addition $A_7 = A_4 + t^{2n}A_5 + t^{6n}A_6$
12. $6n + 2k - 1$: addition $A_8 = (1+t^{4n})A_7$
13. $6n + 2k + M(4n)$: multiplication
    $A_9 = \left(\sum_{i=0}^{3} t^{in}(F_i + F_{i+4})\right)\left(\sum_{i=0}^{3} t^{in}(G_i + G_{i+4})\right)$
14. $8n - 1$: addition $A_8 + t^{4n}A_9$

Summing all the costs above, we get the recursion for the *Three-level* algorithm:

$$M(7n+k) \leq M(4n) + 2M(2n) + 11M(n) + M(k) + \\ + 67n + 12k - 17, \quad \frac{n}{2} \leq k \leq n. \quad (2.16)$$

Based on the technique explained to design the *Three-level* algorithm, one can devise the *Four-level* algorithm for two $(15n+k)$-bit polynomials, getting

$$M(15n+k) \leq M(8n) + 2M(4n) + 4M(2n) + 23M(n) + M(k) + \\ + 191n + 16k - 34, \quad \frac{n}{2} \leq k \leq n, \quad (2.17)$$

and the *Five-level* algorithm for two $(31n+k)$-bit polynomials, getting

$$M(31n+k) \leq M(16n) + 2M(8n) + 4M(4n) + 8M(2n) + 47M(n) + M(k) + \\ + 491n + 20k - 67, \quad \frac{n}{2} \leq k \leq n. \quad (2.18)$$

The first thing to notice is that the recursions for *{Three, Four, Five}-level* algorithms take the advantage of many more algorithms than the well known *Refined-Karatsuba* or even the recent *Karatsuba-like* algorithms. The state of art algorithms rely their optimization on one or two multiplication algorithms, instead, these new *{Three, Four, Five}-level* algorithms discard the inner multiplications on many more algorithms. It means that, for a generic algorithm, one needs to optimize one specific algorithm. On the contrary, take for instance the *Five-level* recursion (2.18). It uses six algorithms, therefore there are many possibilities for further optimization.

The second feature of the recursions for *{Three, Four, Five}-level* algorithms is about the nature of the algorithms on which they rely. In fact, it can be noticed that every *x-level* algorithm rely on the previous levels. This makes a sort of chain of improvements and this is the strength of this new technique.

2.2.1. Results

When designing an algorithm to improve the polynomial multiplication, one needs also to take care of the depth. We can define the depth as the higher number of operations which cannot be executed in parallel. A little example will clarify what depth is.

Example 2.2. Recall the example 1.2. We can see that the algorithm is executed in 13 operations. Looking carefully, we note that many operations could be performed contemporary to other ones because they do not need to wait for the input. Precisely, at most 9 out of those 13 operations could be done in parallel, resulting in depth equal to 3. We depict in figure 2.1 the parallel execution of the algorithm. The depth can be checked from left to right.

We can now show the results of the improvements due to *{Three, Four, Five}-level* algorithms. They can be examined in table 2.1. Note that we were not able to replicate an optimized XOR circuit for the highest level of the *Karatsuba-like* algorithm, thus, the new results from *{Three, Four, Five}-level* algorithms could be further improved.

Figure 2.1: parallel execution of polynomial multiplication.

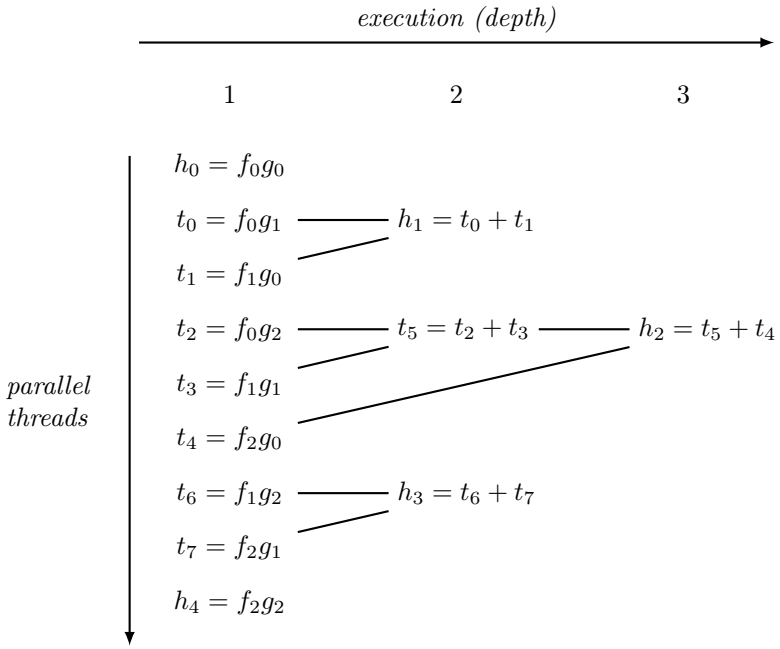


Table 2.1: improvements for n-bit polynomial multiplication.

n	n of operation before	n of operation after	depth before	depth after	algorithm
24	702 [19]	697	10	9	*Three-level*
32	1156 [19]	1148	11	10	*Three-level*
40	1703 [35]	1700	14	13	*Three-level*
47	2228 [35]	2214	13	11	*Four-level*
48	2259 [35]	2238	13	11	*Four-level*
63	3626 [35]	3612	14	12	*Four-level*
64	3673 [35]	3640	13	12	*Four-level*
72	4510 [35]	4510	25	15	*Three-level*
79	5329 [35]	5313	16	15	*Four-level*
80	5366 [35]	5345	16	15	*Four-level*
95	7073 [35]	6978	15	13	*Five-level*
96	7110 [35]	7006	16	13	*Five-level*
120	10438 [8]	10294	130	17	*Three-level*
127	11447 [8]	11277	17	14	*Five-level*
128	11466 [19]	11309	16	14	*Five-level*

A particular result is the one about 120-bit polynomials. It is not an error. The high depth is due to the fact that the algorithm has been devised through an interpolation technique that will be discuted in subsection 2.3.3. This technique is particular efficient for polynomials having more than 128 bit only in number of operations (gates), but it is not for the depth of the algorithm.

2.3. Upper bounds

Recall the inequality (2.3). It can be seen not only as an upper bound for the *School-book* algorithm itself but also as an upper bound at all. We are going to see that improvements in section 2.2 could lead to better upper bounds. In section 2.4, we will also develop further the technique presented in [19]. By doing this we will see some improvements in upper bounds but not for practical interests and this suggests to not investigate anymore this technique.

2.3.1. How to compute upper bounds

It's not difficult to compute upper bounds from costs estimates shown in previous sections. We just need a little lemma.

Lemma 2.1 (from the Master Theorem)**.** *Let a, b and i be positive integers and assume that $a \neq b$. Let $n = b^i$ and $a \neq 1$. The solution to the inductive relation*

$$\begin{cases} r_1 = e \\ r_n = ar_{n/b} + cn + d \end{cases}$$

is

$$r_n = \left(e + \frac{bc}{a-b} + \frac{d}{a-1}\right) n^{\log_b a} - \frac{bc}{a-b}n - \frac{d}{a-1}.$$

Proof. The proof is trivial. Substituting in the inductive relation the expression for r_n and $r_{n/b}$, we find an identity. □

We are going to present how to compute the *Refined-Karatsuba* upper bound using the above lemma.

Example 2.3. We start by recalling which is the *Refined-Karatsuba* recursion.

$$M(2n) \leq 3M(n) + 7n - 3$$

It does not perfectly fit the lemma's hypothesis but we simply manipulate the variable.

$$M(n) \leq 3M\left(\frac{n}{2}\right) + \frac{7}{2}n - 3$$

The application of lemma is straightforward and leads to

$$M(n) \leq 6.5n^{\log_2 3} - 7n + 1.5.$$

Unfortunately, the *{Three, Four, Five}-level* recursions have not an expression which is suitable for the lemma. Once again, a little trick could help us. We are going to present how to compute the *Two-level Seven-way* upper bound as an example to compute also *{Three, Four, Five}-level* upper bounds.

Example 2.4. Recall that, for $k = n$, the *Two-level Seven-way* recursion is

$$M(4n) \leq M(2n) + 6M(n) + 27n - 8.$$

At this stage we cannot apply lemma 2.1, but if we substitute the recursion (2.6) from the *Refined-Karatsuba* for $k = n$, we get

$$M(4n) \leq 9M(n) + 34n - 11.$$

Note also that this is not the same recursion from (2.10), the *Karatsuba-like* algorithm. At this point we can apply lemma 2.1, obtaining

$$M(n) \leq 6.43n^{log_2 3} - 6.8n + 1.38.$$

To enable an easy comparison of different algorithms, in table 2.2 we present the upper bounds of the algorithms that we have presented so far. Note that there is no an algorithm better than another one, since they apply to polynomials with different number of terms.

We are going to present a new technique to devise better upper bounds. It takes the advantage of finite fields larger than $\mathbb{F}_2$. We also need the Projective Lagrange Interpolation.

Table 2.2: upper bounds comparison.

algorithm	upper bound	number of bits
Karatsuba [43]	$M(n) \leq 7.00n^{\log_2 3} - 8.00n + 2.00$	multiple of 2
Refined-Karatsuba [8]	$M(n) \leq 6.50n^{\log_2 3} - 7.00n + 1.50$	multiple of 2
Two-level Seven-way [8]	$M(n) \leq 6.43n^{\log_2 3} - 6.80n + 1.38$	multiple of 4
Karatsuba-like [35]	$M(n) \leq 6.30n^{\log_2 3} - 6.80n + 1.50$	multiple of 4
Karatsuba-like [35]	$M(n) \leq 6.17n^{\log_2 3} - 6.75n + 1.58$	multiple of 5
Karatsuba-like [35]	$M(n) \leq 6.91n^{\log_2 3} - 7.73n + 1.81$	multiple of 6
Karatsuba-like [35]	$M(n) \leq 6.51n^{\log_2 3} - 7.13n + 1.62$	multiple of 7
Three-level [29]	$M(n) \leq 6.34n^{\log_2 3} - 6.68n + 1.35$	multiple of 8
Four-level [29]	$M(n) \leq 6.30n^{\log_2 3} - 6.62n + 1.31$	multiple of 16
Five-level [29]	$M(n) \leq 6.28n^{\log_2 3} - 6.57n + 1.30$	multiple of 32

2.3.2. Projective Lagrange Interpolation

Let $\mathbb{K}$ a field and $H \in \mathbb{K}[x]$ a polynomial,

$$H(x) = h_0 + h_1 x + h_2 x^2 + \ldots + h_n x^n. \tag{2.19}$$

We need its value in $n+1$ generic points to uniquely determine it. Thus, take $n+1$ values in the field, say $\{k_0, \ldots, k_n\} \subseteq \mathbb{K}$. We define the *Lagrange polynomials* as follows:

$$l_i(x) = \prod_{j \neq i} \frac{x - k_j}{k_i - k_j} \qquad i = 0, \ldots, n.$$

Note that the definition is designed that way to have $l_i(k_i) = 1$ and $l_i(k_j) = 0$, $\forall j \neq i$. This feature allows us to exactly reconstruct any polynomial $H \in \mathbb{K}[x]$ as

$$H(x) = \sum_{i=0}^{n} H(k_i) \cdot l_i(x).$$

For our purposes, H will be a polynomial whose degree is the cardinality of the field containing the values for the interpolation, we need one more value. Since we have not, we extend the definition of the interpolation.

Given again the polynomial H as in equation (2.19) and only n points, define the following $n-1$ degree polynomial:

$$\overline{H}(x) = \sum_{i=0}^{n-1} H(k_i) \cdot l_i(x).$$

We still have $\overline{H}(k_i) = H(k_i)$, for $i = 0, \ldots, n-1$. Let

$$l_\infty(x) = \prod_{j=0}^{n-1} (x - k_j)$$

and $H(\infty) = h_n$. Since $H(\infty) \cdot l_\infty$ vanishes at every k_i and has degree n, we can reconstruct H with the so-called *Projective Lagrange Interpolation* formula,

$$H(x) = \sum_{i=0}^{n-1} H(k_i) \cdot l_i(x) + H(\infty) \cdot l_\infty(x). \tag{2.20}$$

2.3.3. Interlude: Five-way recursion

We will see that Projective Lagrange Interpolation will be very useful to get better upper bounds but it can still be used also to devise better polynomial multiplication algorithms of practical interest such as *Five-way recursion* [8].

Given two $(2n+k)$-bit polynomials

$$\begin{aligned} F &= f_0 + f_1 t + \ldots + f_{2n+k-1} t^{2n+k-1} \\ \text{and} \quad G &= g_0 + g_1 t + \ldots + g_{2n+k-1} t^{2n+k-1} \end{aligned}$$

let

$$\begin{aligned} F_0 &= f_0 + f_1 t + \ldots + f_{n-1} t^{n-1} \\ F_1 &= f_n + f_{n+1} t + \ldots + f_{2n-1} t^{n-1} \\ \text{and} \quad F_2 &= f_{2n} + f_{2n+1} t + \ldots + f_{2n+k-1} t^{k-1} \end{aligned}$$

doing the same for G, and rewrite F and G as

$$\begin{aligned} F &= F_0 + F_1 t^n + F_2 t^{2n} \\ \text{and} \quad G &= G_0 + G_1 t^n + G_2 t^{2n}. \end{aligned}$$

Instead of going directly to the algorithm, we do an abstraction. We define $t^n = x$ and rewrite F and G as

$$\begin{aligned} F &= F_0 + F_1 x + F_2 x^2 \\ \text{and} \quad G &= G_0 + G_1 x + G_2 x^2. \end{aligned}$$

Thefore, the result from the multiplication of F and G will be as follows:

$$\begin{aligned} H &= F \cdot G \\ &= (F_0 + F_1 x + F_2 x^2) \cdot (G_0 + G_1 x + G_2 x^2) \\ &= H_0 + H_1 x + H_2 x^2 + H_3 x^3 + H_4 x^4. \end{aligned}$$

It seems that we can now apply the *Projective Lagrange Interpolation*, but a problem arises: we just have 0 and 1 to use as values for x. We need two more points. The trick is to consider t as a value. In fact, the polynomial H can be reconstructed from the values $H(0)$, $H(1)$, $H(t)$, $H(t+1)$, $H(\infty)$ by the *Projective Lagrange Interpolation* formula

$$\begin{aligned} H &= H(0)\frac{(x+1)(x+t)(x+t+1)}{t(t+1)} + H(1)\frac{x(x+t)(x+t+1)}{(1+t)t} \\ &= H(t)\frac{x(x+1)(x+t+1)}{t(t+1)} + H(t+1)\frac{x(x+1)(x+t)}{(t+1)t} \\ &= H(\infty)x(x+1)(x+t)(x+t+1). \end{aligned}$$

Note that the denominator is the same in the four fractions. This and a manual simplification allow to write the algorithm as follows

$$H = U + H(\infty)(x^4+x) + \frac{(U+V+H(\infty)(t^4+t))(x^2+x)}{t^2+t}$$

where $U = H(0) + (H(0)+H(1))x$ and $V = H(t) + (H(t)+H(t+1))(x+t)$. The following cost evaluation tell us also how to perform the algorithm.

1. $M(n)$: multiplication $H(0) = F_0G_0$
2. $M(k)$: multiplication $H(\infty) = F_2G_2$
3. $2(n+2)$, assuming $k \leq n$: additions $F_0+F_1+F_2$ and $G_0+G_1+G_2$
4. $M(n)$: multiplication $H(1) = (F_0+F_1+F_2)(G_0+G_1+G_2)$
5. $2(n-1)$, or $2k$ if $k \leq n-1$: additions $F_1t+F_2t^2$ and $G_1t+G_2t^2$
6. $2(n-1)$: additions $F_0+(F_1t+F_2t^2)$ and $G_0+(G_1t+G_2t^2)$
7. $M(n+2)$, or $M(n+1)$ if $k \leq n-1$: multiplication $H(t)$, i.e. the product of $F_0+(F_1t+F_2t^2)$ and $G_0+(G_1t+G_2t^2)$
8. $2(n-1)$: additions $(F_0+F_1+F_2)+(F_1t+F_2t^2)$ and $(G_0+G_1+G_2)+(G_1t+G_2t^2)$
9. $M(n+2)$, or $M(n+1)$ if $k \leq n-1$: multiplication $H(t+1)$, i.e. the product of $(F_0+F_1+F_2)+(F_1t+F_2t^2)$ and $(G_0+G_1+G_2)+(G_1t+G_2t^2)$
10. $2n+1$: addition $H(t)+H(t+1)$
    The coefficients of t^{2n+2} and t^{2n+1} in $H(t+1)$ are the same as the coefficients of t^{2n+2} and t^{2n+1} in $H(t)$, so this sum has degree at most $2n$
11. $3n+4$, or $3n+2$ if $k \leq n-1$: additions $V = H(t)+(H(t)+H(t+1))(t^n+t)$
    Note that $\deg(V) \leq 3n$.
12. $3n-2$: additions $U = H(0)+(H(0)+H(1))t^n$
    Note that $\deg(U) \leq 3n-2$.
13. $4k+3n-3$, assuming $n \geq 2$: additions $W = U+V+H(\infty)(t^4+t)$
    Note that $\deg(W) \leq 3n$.

14. $3n-2$: division $W/(t^2+t)$
    This division is exact and, for the same reason, some work could have been skipped in the computation of W.)

15. $5n+2k-4$: additions $H(\infty)(t^{4n}+t^n)+(W/(t^2+t))(t^{2n}+t^n)+U$

Thus, the total cost of the algorithm is

$$\begin{cases} M(2n+k) \leq 2M(n)+M(k)+2M(n+1)+25n+10k-12 & 1 \leq k \leq n-1 \\ M(3n) \leq 3M(n)+2M(n+2)+35n-12 & n \geq 2 \end{cases}$$

We list now some remarks about the *Five-way recursion.*

- Recall table 2.1. We have that using the *Five-way recursion* algorithm, the product between two 120-bit polynomials can be done using 10438 operations with a depth equal to 130. This is precisely the drawback of the interpolation technique. Even if the number of operations can be considerably improved, the depth prevents an high level of parallelization and thus the speed of execution.

- During the analysis of the algorithms, it has been taken into account the possibility of considering new set of points such as $\{0,1,t,t+1,t^2,t^2+1,t^2+t+1\}$. No one of the set considered leaded to an improvement.

- In [19, Section 3.1], some improvements are proposed. Some experiments show that the *Five-way recursion* algorithm and its improvements are the best ones for products involving polynomials with more than 128 bits.

2.3.4. Cenk – Negre – Hasan

In [21], Cenk, Negre and Hasan suggest a new algorithm that takes the advantage of the *Projective Lagrange Interpolation.* It does not lead to any improvement for degrees of practical interest, instead, it is very useful to devise better upper bounds.

Since an interpolation technique requires a field bigger than $\mathbb{F}_2$, authors use $\mathbb{F}_4$. Since we have seen in chapter 1 that the representation could be non unique, we explicitly say that we build $\mathbb{F}_4$ as the quotient $\mathbb{F}_4[z]/\langle f(z)\rangle$ with $f(z)=z^2+z+1$. Therefore, from now on, α will be a root of $f(z)$, thus, respecting the relation $\alpha^2+\alpha+1=0$. Moreover we want now to distinguish multiplications between polynomials in $\mathbb{F}_2[t]$ and multiplications between polynomials in $\mathbb{F}_4[t]$. We will denote the first ones as $M_2(n)$ and the second ones as $M_4(n)$.

The algorithm by Cenk, Negre and Hasan is called *3-way split* and uses the same split and the same substitution of the *Five-way recursion.* Given two $3n$-bit polynomials

$$\begin{aligned} F &= f_0+f_1t+\ldots+f_{3n-1}t^{3n-1} \\ \text{and}\quad G &= g_0+g_1t+\ldots+g_{3n-1}t^{3n-1} \end{aligned}$$

let

$$\begin{aligned} F_0 &= f_0+f_1t+\ldots+f_{n-1}t^{n-1} \\ F_1 &= f_n+f_{n+1}t+\ldots+f_{2n-1}t^{n-1} \\ \text{and}\quad F_2 &= f_{2n}+f_{2n+1}t+\ldots+f_{3n-1}t^{n-1} \end{aligned}$$

doing the same for G, and rewrite F and G as

$$\begin{aligned} F &= F_0 + F_1 t^n + F_2 t^{2n} \\ \text{and} \quad G &= G_0 + G_1 t^n + G_2 t^{2n}. \end{aligned}$$

Define $t^n = x$ and rewrite F and G as

$$\begin{aligned} F &= F_0 + F_1 x + F_2 x^2 \\ \text{and} \quad G &= G_0 + G_1 x + G_2 x^2. \end{aligned}$$

Thefore, the result from the multiplication of F and G will be as follows:

$$\begin{aligned} H &= F \cdot G \\ &= (F_0 + F_1 x + F_2 x^2) \cdot (G_0 + G_1 x + G_2 x^2) \\ &= H_0 + H_1 x + H_2 x^2 + H_3 x^3 + H_4 x^4. \end{aligned}$$

In order to apply the *Projective Lagrange Interpolation*, we use $\mathbb{F}_4$ as the set of values for the evaluation of interpolating points.

$$\begin{aligned} H(0) &= F_0 G_0 \\ H(1) &= (F_0 + F_1 + F_2)(G_0 + G_1 + G_2) \\ H(\alpha) &= (F_0 + F_2 + \alpha(F_1 + F_2))(G_0 + G_2 + \alpha(G_1 + G_2)) \\ H(\alpha+1) &= (F_0 + F_1 + \alpha(F_1 + F_2))(G_0 + G_1 + \alpha(G_1 + G_2)) \\ H(\infty) &= F_2 G_2 \end{aligned}$$

Note that the formulae for $H(\alpha)$ and $H(\alpha+1)$ are correct when n is odd. When n is even, we just exchange the formulae. At this point we just need to write the *Projective Lagrange Interpolation* formula:

$$\begin{aligned} H &= H(0)\frac{(x+1)(x+\alpha)(x+\alpha+1)}{\alpha(\alpha+1)} + H(1)\frac{x(x+\alpha)(x+\alpha+1)}{(1+\alpha)\alpha} \\ &= H(\alpha)\frac{x(x+1)(x+\alpha+1)}{\alpha(\alpha+1)} + H(\alpha+1)\frac{x(x+1)(x+\alpha)}{(\alpha+1)\alpha} \\ &= H(\infty)x(x+1)(x+\alpha)(x+\alpha+1). \end{aligned}$$

Similar to the *Five-way recursion*, we get the same denominator for all the terms but now we have even more. In fact, α was the root of $f(z)$, i.e. $\alpha^2 + \alpha + 1 = 0$ and this means that $\alpha^2 + \alpha = 1$. With an easy simplification, we get

$$\begin{aligned} F \cdot G = &\Big(H(0) + xH(\infty)\Big)(1 + x^3) \\ &+ \Big(H(1) + (1+\alpha)(H(\alpha) + H(\alpha+1))\Big)(x + x^2 + x^3) \\ &+ \alpha\Big(H(\alpha) + H(\alpha+1)\Big)x^3 + H(\alpha)x^2 + H(\alpha+1)x. \end{aligned} \tag{2.21}$$

The cost evaluation is

$$M_2(3n) \leq 2M_4(n) + 3M_2(n) + 29n - 12.$$

In [19] is described an improvement. Using the following relations

$$\begin{aligned} H(\alpha) &= \Big(F_0 + F_2 + \alpha(F_1 + F_2)\Big)\Big(G_0 + G_2 + \alpha(G_1 + G_2)\Big) \\ &= C_0 + \alpha C_1 \\ H(\alpha+1) &= \Big(F_0 + F_1 + \alpha(F_1 + F_2)\Big)\Big(G_0 + G_1 + \alpha(G_1 + G_2)\Big) \\ &= (C_0 + C_1) + \alpha C_1 \end{aligned}$$

it is possible to redefine (2.21).

$$\begin{aligned} F \cdot G = H(\infty)x^4 &+ H(0) \\ &+ \Big(H(0) + H(1) + C_1\Big)x^3 \\ &+ \Big(C_0 + H(1) + C_1\Big)x^2 \\ &+ \Big(H(\infty) + H(1) + C_0\Big)x. \end{aligned} \tag{2.22}$$

Therefore, the cost evaluation is refined to

$$\begin{cases} M_2(3n) \leq 3M_2(n) + M_4(n) + 20n - 5 \\ M_4(3n) \leq 5M_4(n) + 56n - 19 \end{cases} \tag{2.23}$$

Note that we can solve the second recursion with lemma 2.1 which gives

$$M_4(n) \leq 30.25n^{1.46} - 28n + 4.75.$$

By substituting this inequality in the first one of (2.23), we obtain

$$M_2(3n) \leq 3M_2(n) + 30.25n^{1.46} - 8n - 0.25. \tag{2.24}$$

We need one more lemma to solve it and get the final upper bound for the *3-way split* algorithm.

Lemma 2.2 (from the Master Theorem)**.** *Let a, b and i be positive integers. Let $n = b^i$, $a = b$, $a \neq 1$ and $\delta \neq 1$. The solution to the inductive relation*

$$\begin{cases} r_1 = e \\ r_n = ar_{n/b} + cn + fn^\delta + d \end{cases}$$

is

$$r_n = \left(e + \frac{fb^\delta}{a - b^\delta} + \frac{d}{a-1}\right) n - n^\delta \left(\frac{fb^\delta}{a - b^\delta}\right) + cn \log_b n - \frac{d}{a-1}.$$

Proof. It is just a calculation as in the proof of lemma 2.1. □

Finally, we simply apply lemma 2.2 to the inequality (2.24), obtaining

$$M_2(n) \leq 15.125n^{1.46} - 14.25n - 2.67n \log_3 n + 0.125.$$

2.4. Improvements of upper bounds

We have seen in subsection 2.3.1, in particular in table 2.2, that the upper bound for an algorithm of practical interest is $O(n^{1.58})$. Moreover, we have just seen that the algorithm (2.22) improves the estimate to $O(n^{1.46})$. We are going to see that a refinement of the *3-way split* algorithm can lead to better upper bounds.

2.4.1. A new algorithm

We take d a non negative integer and the factors F and G of the form

$$F(t) = \sum_{i=0}^{2^{d-1}} F_i(t)t^{in} \qquad \text{with } F_i \in \mathbb{F}_2[t], \quad \deg F_i \leq n-1$$

In order to simplify notation, given a factor $F(t)$ of the above form, we define

$$\widetilde{F}(x) = \sum_{i=0}^{2^{d-1}} F_i(t)x^i$$

We are now ready to suggest a new efficient algorithm.

Let's start with an observation. There is an interesting connection between $x^{2^d} + x$ and Lagrange polynomials, indeed, we can prove the following three equalities:

1. $l_0(x) = \dfrac{x^{2^d} + x}{x} = x^{2^d-1} + 1$
2. $l_{\alpha^i}(x) = \dfrac{x^{2^d} + x}{x + \alpha^i} \qquad i = 0, 1, \ldots, 2^d - 2$
3. $l_\infty = x^{2^d} + x = x(x^{2^d-1} + 1) = x \cdot l_0(x)$

We now manipulate the interpolation law as follows:

$$\widetilde{H}(x) = \widetilde{H}(0) \cdot l_0(x) + \sum_{i=0}^{2^d-2} \widetilde{H}(\alpha^i) \cdot l_{\alpha^i}(x) + \widetilde{H}(\infty) \cdot l_\infty(x)$$

$$\widetilde{H}(x) = \widetilde{H}(0) \cdot l_0(x) + \sum_{i=0}^{2^d-2} \widetilde{H}(\alpha^i) \cdot l_{\alpha^i}(x) + x\widetilde{H}(\infty) \cdot l_0(x)$$

$$\widetilde{H}(x) = \widetilde{H}(0) \cdot (1 + x^{2^d-1}) + \sum_{i=0}^{2^d-2} \widetilde{H}(\alpha^i)\frac{x^{2^d} + x}{x + \alpha^i} + x\widetilde{H}(\infty) \cdot (1 + x^{2^d-1})$$

reaching the following state

$$\widetilde{H}(x) = (1 + x^{2^d-1})(\widetilde{H}(0) + x\widetilde{H}(\infty)) + \sum_{i=0}^{2^d-2} \widetilde{H}(\alpha^i)\frac{x^{2^d} + x}{x + \alpha^i} \tag{2.25}$$

Notice that fractions in of Equation (2.25) are Lagrange polynomials of $\mathbb{F}_{2^d}^{\times}$. Using the naive division algorithm, we obtain

$$l_{\alpha^i}(x) = \frac{x^{2^d} + x}{x + \alpha^i} = \sum_{j=1}^{2^d-1} (\alpha^i)^{(j-1)} x^{2^d-j} \tag{2.26}$$

and replacing equation (2.26) in (2.25), we get

$$\widetilde{H}(x) = (1 + x^{2^d-1})(\widetilde{H}(0) + x\widetilde{H}(\infty)) + \sum_{i=0}^{2^d-2} \widetilde{H}(\alpha^i) \sum_{j=1}^{2^d-1} \alpha^{i(j-1)} x^{2^d-j}$$

$$\widetilde{H}(x) = \underbrace{(1 + x^{2^d-1})(\widetilde{H}(0) + x\widetilde{H}(\infty))}_{S_A} + \underbrace{\sum_{j=1}^{2^d-1} \left(\sum_{i=0}^{2^d-2} \alpha^{i(j-1)} \widetilde{H}(\alpha^i) \right) x^{2^d-j}}_{S_B} \tag{2.27}$$

We will now discuss the costs of this algorithm.

Consider S_A: it will always be the same in every field $\mathbb{F}_{2^d}$. The cost of the operations in S_A is:

- $M_2(n)$: multiplication $\widetilde{H}(0) = F_0 G_0$
- $M_2(n)$: multiplication $\widetilde{H}(\infty) = F_{2^{d-1}} G_{2^{d-1}}$
- $n - 1$: sum $\widetilde{H}(0) + x\widetilde{H}(\infty)$
- 0: sum $(1 + x^{2^d-1})(\widetilde{H}(0) + x\widetilde{H}(\infty))$

The last estimate holds only for $d \neq 1$, otherwise polynomials $\widetilde{H}(0) + x\widetilde{H}(\infty)$ and $x(\widetilde{H}(0) + x\widetilde{H}(\infty))$ overlap on some bits and it becomes $2n - 1$.

Consider now the sum $S_A + S_B$. The degree of S_A is $(2^d + 2)n - 2$, but its structure lacks many powers. Indeed, S_A is a polynomial that has two parts, the first with powers whose degrees are running from 0 to $3n - 2$, the second from $(2^d-1)n$ to $(2^d+2)n-2$. This is very useful because S_B has powers with degrees from n to $(2^d + 1)n - 2$, so, S_A and S_B overlaps only in two parts. The first in $(3n-2)-n+1 = 2n-1$ bits and the second in $(2^d+1)n-2-(2^d-1)n+1 = 2n-1$. Since the cost of $S_A + S_B$ does not depend on the field, it is

- $4n - 2$: sum $H(t) = S_A + S_B$

Finally, consider the sums in S_B. Supposing that the internal summation has been computed, the external one is conducted over $2^d - 1$ polynomials. These polynomials have powers from cn to $cn + 2n - 2$, with $c = 1, \ldots, 2^d - 1$ and each one overlaps the following on $n - 1$ bit. Therefore, the cost of the external sum in S_B is

- $(2^d - 2)(n - 1)$: sum $S_1 x + S_2 x^2 + \cdots + S_{2^d-1} x^{2^d-1}$

We are left to compute the internal sums in S_B. We will show that we do not need to compute all $\widetilde{H}(\alpha^i)$.

Firstly, we start with showing that if $i = 2^q i'$ for some q, then there will be a connection between the coefficients of $\widetilde{H}(\alpha^i)$ and $\widetilde{H}(\alpha^{i'})$.

Theorem 2.1. *If we take integers i and i' such that $i' = 2^q i$ for some q, then we can express the coefficients of $\widetilde{H}(\alpha^{i'})$ as a linear combination of the coefficients of $\widetilde{H}(\alpha^i)$.*

Proof. We have

$$\widetilde{H}(\alpha^i) = \widetilde{F}(\alpha^i) \cdot \widetilde{G}(\alpha^i) = \sum_{j=0}^{2^{d-1}} F_j \alpha^{ij} \sum_{k=0}^{2^{d-1}} G_k \alpha^{ik} = \sum_{l=0}^{2^d} \left(\sum_{\substack{j+k=l \\ 0 \le j,k \le 2^{d-1}}} F_j G_k \right) (\alpha^i)^l.$$

We define

$$H_l = \sum_{\substack{j+k=l \\ 0 \le j,k \le 2^{d-1}}} F_j G_k$$

thus

$$\widetilde{H}(\alpha^i) = \sum_{l=0}^{2^d} H_l \alpha^{il} \tag{2.28}$$

Remember that the field $\mathbb{F}_{2^d}$ can be viewed as vector space over $\mathbb{F}_2$. So, we can write every power of α as a linear combination of the elements of the basis $\{1, \alpha, \alpha^2, \ldots, \alpha^{d-1}\}$

$$\alpha^{il} = \sum_{b=0}^{d-1} c_{b,il} \alpha^b \tag{2.29}$$

and substitute (2.29) in (2.28), getting

$$\widetilde{H}(\alpha^i) = \sum_{l=0}^{2^d} H_l \sum_{b=0}^{d-1} c_{b,il} \alpha^b = \sum_{b=0}^{d-1} \left(\sum_{l=0}^{2^d} H_l c_{b,il} \right) \alpha^b$$

Take now $\widetilde{H}(\alpha^{iw})$ with $w > 1$, from (2.29) we have

$$\alpha^{ilw} = (\alpha^{il})^w = \left(\sum_{b=0}^{d-1} c_{b,il} \alpha^b \right)^w.$$

In order to write coefficients of $\widetilde{H}(\alpha^{iw})$ as linear combinations of the coefficients of $\widetilde{H}(\alpha^i)$, we need the following equality:

$$\left(\sum_{b=0}^{d-1} c_{b,il} \alpha^b \right)^w = \sum_{b=0}^{d-1} c_{b,il} \alpha^{bw} \tag{2.30}$$

Suppose it holds, then

$$\widetilde{H}(\alpha^{iw}) = \sum_{l=0}^{2^d} H_l \left(\sum_{b=0}^{d-1} c_{b,il} \alpha^b \right)^w = \sum_{l=0}^{2^d} H_l \sum_{b=0}^{d-1} c_{b,il} \alpha^{bw} = \sum_{b=0}^{d-1} \left(\sum_{l=0}^{2^d} H_l c_{b,il} \right) \alpha^{bw}.$$

Finally, using (2.29), we obtain

$$\widetilde{H}(\alpha^{iw}) = \sum_{b=0}^{d-1} \left(\sum_{l=0}^{2^d} H_l c_{b,il} \right) \alpha^{bw} = \sum_{b=0}^{d-1} \left(\sum_{l=0}^{2^d} H_l c_{b,il} \right) \sum_{t=0}^{d-1} c_{t,bw} \alpha^t =$$

$$= \sum_{t=0}^{d-1} \left(\sum_{b=0}^{d-1} c_{t,bw} \left(\sum_{l=0}^{2^d} H_l c_{b,il} \right) \right) \alpha^t$$

Let's go back to (2.30): since we are in characteristic two, the equality holds when $w = 2^q$, for some q. □

Secondly, we have to remember that $\alpha^{2^d} = \alpha$. So, for every $\widetilde{H}(\alpha^i)$, with $i \not\equiv 0 \pmod{2^d - 1}$, there are at most d different evaluations of $\widetilde{H}$ that can be computed with $\widetilde{H}(\alpha^i)$. They are the following set:

$$P_i = \{\widetilde{H}(\alpha^i), \widetilde{H}(\alpha^{2i}), \widetilde{H}(\alpha^{2^2 i}), \ldots, \widetilde{H}(\alpha^{2^{d-1} i})\}$$

We can count the number of P_i for every algebraic extension of $\mathbb{F}_2$, because it depends only on the degree d.

Theorem 2.2. *The number of different P_i is*

$$P = -1 + \frac{1}{d} \sum_{k=0}^{d-1} \gcd(2^k - 1, 2^d - 1)$$

In particular, if $2^d - 1$ is prime, $P = (2^d - 2)/d$.

We define an action of the (additive) group $\mathbb{Z}$ on $\mathbb{Z}/(2^d - 1)\mathbb{Z}$ as $k \cdot i = 2^k i$. Since d acts trivially, this action induces an action of $\mathbb{Z}/d\mathbb{Z}$ on $\mathbb{Z}/(2^d - 1)\mathbb{Z}$: if $O(i)$ is the orbit of $i \in \mathbb{Z}/(2^d - 1)\mathbb{Z}$, then $P_i = \{\widetilde{H}(\alpha^j) : j \in O(i)\}$. We have a trivial orbit $O(0) = \{0\}$ which would correspond to the set $P_0 = \{\widetilde{H}(1)\}$ which we will not count. In order to prove the Theorem 2.2, we need a couple of additional lemmata.

Lemma 2.3 (Burnside's Lemma)**.** *If the finite group G acts on the finite set X, then the number of orbits is*

$$\frac{1}{\#G} \sum_{g \in G} \#\,\mathrm{Fix}(g)$$

where $\mathrm{Fix}(g) = \{x \in X : g \cdot x = x\}$.

Proof. See [68, Chapter 3]. □

Lemma 2.4. *Fix an integer N and let $x \in \mathbb{Z}/N\mathbb{Z}$. Then*

$$\#\{y \in \mathbb{Z}/N\mathbb{Z} : xy = 0\} = \gcd(x, N)$$

Proof. Let $\mathcal{Z} = \{y \in \mathbb{Z}/N\mathbb{Z} : xy = 0\}$: it is not empty since it includes 0 and it is straightforward to verify that $\mathcal{Z}$ is an ideal in $\mathbb{Z}/N\mathbb{Z}$, thus $\mathcal{Z} = \langle d \rangle$ where d is a divisor of N and $\mathcal{Z}$ has N/d elements. Let $D = \gcd(x, N)$, $\nu = N/D$ and define $\tilde{x}$ as the smallest positive integer such that $\tilde{x} \equiv x \bmod N$. Since

$$\nu x = \frac{N}{D} x \equiv N \frac{\tilde{x}}{D} \equiv 0 \bmod N$$

we have that $\nu \in \mathcal{Z}$. Viceversa, if $y \in \mathcal{Z}$ and $\tilde{y}$ is the smallest positive integer such that $\tilde{y} \equiv y \bmod N$, we have that $\tilde{y}\tilde{x} = kN$ for some integer $k \geq 0$. Thus

$$\tilde{y}\frac{\tilde{x}}{D} = k\frac{N}{D} = k\nu; \quad \text{i.e.,} \quad \tilde{y}\frac{\tilde{x}}{D} \equiv 0 \bmod \nu$$

Since $\tilde{x}/D$ and $\nu = N/D$ are relatively prime, this implies $\tilde{y} \equiv 0 \bmod \nu$, i.e., ν divides $\tilde{y}$, thus $y \in \langle\nu\rangle$. This shows that $\mathcal{Z} = \langle\nu\rangle$, hence that $\#\mathcal{Z} = N/\nu = \gcd(x, N)$. □

Theorem 2.2. Fix $k \in \mathbb{Z}/d\mathbb{Z}$: we want to compute $\mathrm{Fix}(k) = \{x \in \mathbb{Z}/(2^d-1)\mathbb{Z} : k \cdot x = x\}$. If $x \in \mathrm{Fix}(k)$ then $2^k x = x$, that is $(2^k-1)x = 0$; and, viceversa, if $(2^k-1)x = 0$ then $k \cdot x = x$. Hence, $\mathrm{Fix}(k) = \{x \in \mathbb{Z}/(2^d-1)\mathbb{Z} : (2^k-1)x = 0\}$ has, by the previous lemma, $\gcd(2^k-1, 2^d-1)$ elements.

The thesis now follows from Burnside's Lemma. □

Let's sum up the costs of Equation (2.27).

- $M_2(n)$: multiplication $\widetilde{H}(0) = F_0 G_0$
- $M_2(n)$: multiplication $\widetilde{H}(\infty) = F_{2^{d-1}} G_{2^{d-1}}$
- $n-1$: sum $\widetilde{H}(0) + x\widetilde{H}(\infty)$
- 0: sum $(1 + x^{2^d-1})(\widetilde{H}(0) + x\widetilde{H}(\infty))$
- $4n-2$: sum $H(t) = S_A + S_B$
- $(2^d-2)(n-1)$: sums $S_1 x + S_2 x^2 + \cdots + S_{2^d-1} x^{2^d-1}$
- Δ_1: evaluation $\widetilde{F}(\alpha^i)$, $\widetilde{G}(\alpha^i)$
- $M_2(n)$: multiplication $\widetilde{H}(1)$
- $PM_{2^d}(n)$: multiplications $\widetilde{H}(\alpha^i)$
- Δ_2: sums S_i, $i = 1, \ldots, 2^d-1$

Some of the previous costs are left blank, in particular Δ_1 and Δ_2, since the evaluation of F, G and the sums S_i depends on the polynomial used to generate the field $\mathbb{F}_{2^d}$. Roughly speaking, we can say that $\Delta_1 = An$ and $\Delta_2 = B(2n-1)$, obtaining the following estimation:

$$M((2^{d-1}+1)n) \leq 3M_2(n) + PM_{2^d}(n) + \underbrace{(2^d + 3 + A + 2B)}_{Q_1} n + \underbrace{(-1 - 2^d - B)}_{Q_2}$$

$$M((2^{d-1}+1)n) \leq 3M_2(n) + PM_{2^d}(n) + Q_1 n + Q_2 \tag{2.31}$$

Now, we want to apply the following result.

Theorem 2.3 (Master Theorem)**.** *Let a and b be positive real numbers with $a \geq 1$ and $b \geq 2$. Let $T(n)$ be defined by*

$$T(n) = \begin{cases} aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n) & n > 1 \\ d & n = 1 \end{cases}$$

Then

1. *if $f(n) = \Theta(n^c)$ where $\log_b a < c$, then $T(n) = \Theta(n^c) = \Theta(f(n))$,*
2. *if $f(n) = \Theta(n^c)$ where $\log_b a = c$, then $T(n) = \Theta(n^{\log_b a} \log_b n)$,*
3. *if $f(n) = \Theta(n^c)$ where $\log_b a > c$, then $T(n) = \Theta(n^{\log_b a})$.*

The same results apply with ceilings replaced by floors.

Proof. See [64, Section 5.2]. □

We cannot apply theorem 2.3 to (2.31) since both M_2 and M_{2^d} appear: we will have to move everything down to $\mathbb{F}_2$-operations.

2.4.2. Bit operations and asymptotic estimation

As seen in subsection 2.4.1, we need to evaluate an $\mathbb{F}_{2^d}$-polynomial $\widetilde{F}$ of degree $2^d - 1$. Recall that the field $\mathbb{F}_{2^d}$ can be seen as an $\mathbb{F}_2$-vector space of dimension d. Thus, for all i, we can evaluate $\widetilde{F}(\alpha^i)$ as follows:

$$\widetilde{F}(\alpha^i) = \sum_{j=0}^{d-1} F_j \alpha^j \qquad F_j \in \mathbb{F}_2[t]$$

To compute $\widetilde{H}(\alpha^i)$ we need to multiply the two evaluations of $\widetilde{F}$ and $\widetilde{G}$.

$$\widetilde{H}(\alpha^i) = \widetilde{F}(\alpha^i)\widetilde{G}(\alpha^i) = \sum_{j=0}^{d-1} F_j \alpha^j \sum_{k=0}^{d-1} G_k \alpha^k = \sum_{l=0}^{2d-2} \underbrace{\left(\sum_{\substack{j+k=l \\ 0 \leq j,k \leq d-1}} F_j G_k \right)}_{H_l} \alpha^l$$

We want now to compute H_l. We take care only of multiplications. If we look at H_l, we note that it is formed by the sum of the products between F_j and G_k such that $j + k = l$. We separate the two cases: $j = k$ and $j \neq k$. If $j = k$, we need the multiplication $F_j G_j$. If $j \neq k$, we need two multiplications, which are $F_j G_k$ and $F_k G_j$. For the latter, we exchange one multiplication with four sums, since $\operatorname{char} \mathbb{F}_{2^d} = 2$ and we have already computed $F_j G_j$.

$$F_j G_k + F_k G_j = (F_j + F_k)(G_j + G_k) + F_j G_j + F_k G_k$$

The required multiplications are

$$d + \binom{d}{2} = d + \frac{d(d-1)}{2} = \frac{d^2 + d}{2}.$$

Now, we can write the estimation for bit calculations over $\mathbb{F}_{2^d}$, assuming a generic estimate for the number of bit additions

$$M_{2^d}(n) \leq \frac{d^2+d}{2} M_2(n) + Cn + D \tag{2.32}$$

Substituting (2.32) in the estimation (2.31), we obtain a formula which we can apply theorem 2.3 to:

$$M_2((2^{d-1}+1)n) \leq 3M_2(n) + P\left(\frac{d^2+d}{2} M_2(n) + Cn + D\right) + Q_1 n + Q_2$$

$$M_2((2^{d-1}+1)n) \leq \left(3 + \frac{P(d^2+d)}{2}\right) M_2(n) + (Q_1 + CP)n + (Q_2 + DP)$$

Applying the third case of theorem 2.3, we get:

$$M_2(n) = \Theta\left(n^E\right), \quad \text{where} \quad E = \frac{\log\left(3 + \frac{P(d^2+d)}{2}\right)}{\log(2^d+1)}$$

If we compute the exponent E for $1 \leq d \leq 20$, it is not difficult to see that E decreases from 1.58 to 1.17.

Example 2.5 (Case *d=2.*)**.** Using Equation (2.27), we are able to find a better best case bound than that presented in [19] (see CNH 3-way split algorithm (24)). Indeed,

- $M_2(n)$: multiplication $\widetilde{H}(0) = F_0 G_0$
- $M_2(k)$: multiplication $\widetilde{H}(\infty) = F_2 G_2$
- $2k$: sums $S_1 = F_0 + F_2$, $S_2 = G_0 + G_2$
- $2k$: sums $S_3 = F_1 + F_2$, $S_4 = G_1 + G_2$
- $2n$: sums $S_5 = S_1 + F_1$, $S_6 = S_2 + G_1$
- 0: multiplications $P_1 = \alpha S_3$, $P_2 = \alpha S_3$
- 0: sums $S_7 = S_1 + P_1$, $S_8 = S_2 + P_2$
- $M_2(n)$: multiplication $\widetilde{H}(1) = S_5 S_6$
- $M_4(n)$: multiplication $\widetilde{H}(\alpha) = S_7 S_8 (= C_0 + C_1 \alpha)$
- $2n-1$: sum $S_9 = \widetilde{H}(1) + C_1$
- $2n-1$: sum $S_{10} = S_9 + C_0$
- $2n-1$: sum $S_{11} = S_{10} + C_1$
- $2(n-1)$: sums $S_{12} = S_9 x^3 + S_{10} x^2 + S_{11} x$
- $n-1$: sum $S_{13} = \widetilde{H}(0) + x\widetilde{H}(\infty)$
- 0: sum $S_{14} = (1 + x^3) S_{13}$

- $4n-2$: sum $H = S_{14} + S_{12}$

Summing all the costs, we obtain

$$\begin{cases} M(2n+k) \leq 2M_2(n) + M_2(k) + M_4(n) + 15n + 4k - 8 & n/2 \leq k \leq n \\ M(3n) \leq 3M_2(n) + M_4(n) + 19n - 8 & k = n \end{cases} \tag{2.33}$$

But this is not enough. In order to get the best case bound, we have to compute the costs for the same algorithm that uses polynomials over $\mathbb{F}_4$. In this case, we cannot deduce the expression for $\widetilde{H}(\alpha+1)$ from $\widetilde{H}(\alpha)$. In addition, from equation

$$\alpha(a_0 + a_1\alpha) = a_1 + (a_0 + a_1)\alpha$$

we have that the cost of the multiplication by α is 1, and from

$$(a_0 + a_1\alpha) + (b_0 + b_1\alpha) = (a_0 + b_0) + (a_1 + b_1)\alpha$$

we have that the cost of the sum between two polynomials is doubled. Thus,

- $M_4(n)$: multiplication $\widetilde{H}(0) = F_0G_0$
- $M_4(n)$: multiplication $\widetilde{H}(\infty) = F_2G_2$
- $4n$: sums $S_1 = F_0 + F_1$, $S_2 = G_0 + G_1$
- $4n$: sums $S_3 = F_1 + F_2$, $S_4 = G_1 + G_2$
- $2n$: multiplications $P_1 = \alpha S_3$, $P_2 = \alpha S_4$
- $4n$: sums $S_5 = S_1 + P_1$, $S_6 = S_2 + P_2$
- $4n$: sums $S_7 = S_5 + S_3$, $S_8 = S_6 + S_4$
- $4n$: sums $S_9 = S_1 + F_2$, $S_{10} = S_2 + G_2$
- $M_4(n)$: multiplication $\widetilde{H}(1) = S_9S_{10}$
- $M_4(n)$: multiplication $\widetilde{H}(\alpha) = S_7S_8$
- $M_4(n)$: multiplication $\widetilde{H}(\alpha+1) = S_5S_6$
- $8n-4$: sum $S_{13} = \widetilde{H}(1) + \widetilde{H}(\alpha) + \widetilde{H}(\alpha+1)$
- $10n-5$: sum $S_{14} = \widetilde{H}(1) + \widetilde{H}(\alpha+1) + \alpha(\widetilde{H}(\alpha) + \widetilde{H}(\alpha+1))$
- $4n-2$: sum $S_{15} = \widetilde{H}(1) + \widetilde{H}(\alpha) + \alpha(\widetilde{H}(\alpha) + \widetilde{H}(\alpha+1))$
- $4(n-1)$: sums $S_{16} = S_{13}x^3 + S_{14}x^2 + S_{15}x$
- $2(n-1)$: sum $S_{17} = \widetilde{H}(0) + x\widetilde{H}(\infty)$
- 0: sum $S_{18} = (1+x^3)S_{17}$
- $8n-4$: sum $H = S_{18} + S_{16}$

The sum of the costs in $\mathbb{F}_4$ is

$$M_4(3n) \leq 5M_4(n) + 58n - 21$$

We observe that this is not good as

$$M_4(3n) \leq 5M_4(n) + 56n - 19 \tag{2.34}$$

which can be found in [19]. Applying lemma 2.1 to (2.34), we get

$$M_4(n) \leq 30.25n^{1.46} - 28n + 4.75$$

Then, we substitute the preceding inequality to the second of (2.33) obtaining

$$M_2(3n) \leq 3M_2(n) + 30.25n^{1.46} - 9n - 3.25$$

Finally, to get the best case bound, we apply lemma 2.2:

$$M_2(n) \leq 15.125n^{1.46} - 3n\log_3 n - 15.75n + 1.625.$$

3. Real case implementation

Quantum computers are no longer ideas but concrete devices. They could represent a threat for public key cryptography, therefore in 2017 NIST initiated a process to evaluate and standardize a number of quantum-resistant cryptographic algorithms. Some of these algorithms handle large size keys that may cause a reduction of performances in specific contexts. In this chapter, we are going to investigate the possibility to speed up the key-pair generation phase of McEliece cryptosystem. Taking the advantage of the improved polynomial multiplication in field of characteristic 2 developed in chapter 2, remodelling matrices and handling cache in a clever way, it is possible to speed up the generation phase of private and public keys, paying a negligible amount of memory [22].

3.1. Post-Quantum Cryptography

Quantum computing theory and even more quantum computers represent a threat for public key cryptography. Nowadays, they have been built but they do not still have enough qubits to actually break a cipher like RSA implemented with the recommended bit security level. In addition, key establishment schemes and digital signature algorithms are not secure [55] anymore. Therefore, in the last years, researchers have suggested new cryptographic algorithms which are resistant to quantum and classic computers, and we will refer to them as Post-Quantum Cryptography (PQC).

The National Institute of Standards and Technology (NIST) published a call for proposals in order to find new good standards for Post-Quantum Cryptography. Currently, we reached the third round, with seven finalists: four Public-key Encryption and Key-establishment Algorithms, namely Classic McEliece (code-based), CRYSTALS-KYBER (Learning With Errors), NTRU (lattice-based), SABER (Module Learning With Rounding) and three Digital Signature Algorithms, namely CRYSTALS-DILITHIUM (lattice-based), FALCON (lattice-based) and Rainbow (multivariate). Moreover, there are eight Alternate Candidates, divided as follows: five Public-key Encryption and Key-establishment Algorithms called BIKE (code-based), FrodoKEM (lattice-based), HQC (code-based), NTRU Prime (lattice-based), SIKE (supersingular curves isogeny), and three Digital Signature Algorithms, i.e. GeMSS (multivariate), Picnic (zero-knowledge), SPHINCS+ (hash-based). Despite they were well known, the post-quantum cryptographic algorithms had to address a number of issue [66]. For example, the size of the keys were not negligible and post-quantum algorithms were not optimized for real-time applications and resource-constrained devices.

We will focus on *Classic McEliece*, a promising algorithm that reached the

third round of the NIST Post-Quantum Cryptography Standardization Process. This algorithm is based upon a public key cryptosystem by Robert J. McEliece [57] but has to address the problem of its large key sizes. Notice that the key generation process takes a non-negligible time if compared with the encryption/decryption process. One may think that there is no need to speed up the key generation but, for instance, think about generating millions of keys for the citizens of some State. In such a use case, a simple optimization may positively affect the whole key generation process. When generating public and private keys of Classic McEliece cipher, the most important operation is polynomial multiplication. The aim in the following sections is to exploit such speed ups, but not only, to improve the performances of the key-pair generation phase.

3.2. Classic McEliece

In this section, we briefly recall the code-based Classic McEliece cryptosystem, that we will optimize in what follows.

Let n be the length of the code involved in our cryptosystem and $t \geq 2$ its error-correction capability. We consider a finite field $\mathbb{F}_q \equiv \mathbb{F}_2[z]/\langle f(z)\rangle$, where $q = 2^m$ and $m \in \mathbb{N}$, such that $n \leq q$, $mt < n$ and $f(z)$ is a monic irreducible polynomial in $\mathbb{F}_2[z]$ of degree m, defining a representation of the finite field. This defines the code dimension $k = n - mt$.

- *Key generation phase*:
  1. select $g(x) \in \mathbb{F}_q[x]$, a random monic irreducible polynomial;
  2. select $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, n random distinct elements of $\mathbb{F}_q$;
  3. compute the $t \times n$ matrix $\hat{H}$, where $\hat{H}_{ij} = \dfrac{\alpha_j^{i-1}}{g(\alpha_j)}$, for $i = 1, \ldots, t$ and $j = 1, \ldots, n$;
  4. compute the $mt \times n$ matrix H, replacing $\hat{H}_{ij} = a_0 + a_1 x + a_2 x^2 + \ldots$ with its binary representation $a_0, a_1, a_2, \ldots$;
  5. reduce H in the standard form $H = [\mathbb{I}_{n-k} \mid T]$, through a binary Gaussian elimination; if it is not possible return to the first step;
  6. the $(n-k) \times k$ matrix T is the public key, while $(g(x), \alpha_1, \alpha_2, \ldots, \alpha_n)$ is the private key.
- *Encryption phase*:
  1. select a binary message e, which is a column vector of length n and weight t;
  2. construct $H = [\mathbb{I}_{n-k} \mid T]$ using the public key;
  3. compute the ciphertext $c = He$.
- *Decryption phase*:
  1. define $v = (c, 0)$, appending k zeroes to c;
  2. compute the syndrome $s = Hv$ and transform it in a polynomial of $\mathbb{F}_q[x]$;

3. find the value e, using s as input in the Berlekamp-Massey algorithm; if $w(e) = t$ and $c = He$, e is the right message, otherwise the algorithm failed.

Notice that the algorithm could fail. Probabilistic cryptosystems are common in Post-Quantum cryptography. Another famous example is NTRU.

3.3. Software Optimization

In what follows, we present new software optimizations that make use of the following technologies: `CLMUL` set instruction extension [37, 41], advanced polynomial multiplication and a parallel implementation for the public key generation.

Classic McEliece has been proposed as a candidate to the NIST Post-Quantum Cryptography with four different software implementations; due to its better performances, the AVX implementation will be used as basis for the optimizations we will develop in what follows.

Experimental results show that the big bottleneck of this cipher is the *key-pair generation*. This is due to the high dimension of the public key which, in this parameters' configuration, reaches almost 1.4MB: compared to a 2048 bit RSA key, it is clear why this cipher is not recommended in a lightweight environment. On the other hand, the encapsulation and decapsulation phases appear to be very efficient, providing execution times that are *in average* lower than a millisecond. Due to the previous insight, we concentrate our improvements only on the key-pair generation. We identified in this phase *five different operations* that could be strongly optimized, in order to obtain a global speedup: polynomial multiplication (private key), Gaussian reduction (private key), Gaussian reduction (public key), computation of a linear map (public key) and application of the linear map (public key).

3.4. The private key

In McEliece cryptosystem the private key is a Goppa code; in *mceliece8192128* this code is represented by an irreducible Goppa polynomial of degree $t = 128$, obtained by a Gaussian reduction of a randomly defined 129×128 matrix, initialized by the multiplication of two elements in $\mathbb{F}_{(2^{13})^{128}}$, expressed in polynomial form.

We start by optimizing the polynomial multiplication in $\mathbb{F}_{(2^{13})^{128}}$, which relies on the multiplication of their coefficients in $\mathbb{F}_{2^{13}}$, represented again as polynomials over $\mathbb{F}_2$. This means that we have to optimize two different multiplications, namely, `gf_mul_13` for $\mathbb{F}_{2^{13}}$ and `gf_mul_13_128` for $\mathbb{F}_{(2^{13})^{128}}$. The `gf_mul_13` multiplication takes as an input two polynomials that can be represented in software in a very efficient way as 16-bit integer values. The speedup has been obtained via the `CLMUL` and the `PCLMULQQD` instructions [37, 41]. We can perform the operation in 5 instructions instead of the almost 50 required by the original algorithm.

The `gf_mul_13_128` multiplication requires a much more complex optimization, involving the usage of advanced multiplication algorithms as the one described in [29]. This algorithm, applied to the *mceliece8192128* parameters,

generates a total of 11309 operations, 3888 of which are multiplications and 7420 summations.

For a software implementation a problem needs to be addressed: the intermediate results, produced during the computation, need to be stored in memory and retrieved to calculate the subsequent intermediate results. Since the multiplications is repeated 127 times, we get some advantages in environment with cache memory.

The needed modular reduction on the 25-bit polynomials we get as result for `gf_mul_13` cannot be improved, but since we store all 11309 intermediate results of the products, we can avoid to reduce at all: the result of a multiplication on $\mathbb{F}_{2^{13}}$ will never be the input of another multiplication but only of summations, never growing over 25 bits. If we double the memory requirements for memorizing the intermediate results, the modular reduction of `gf_mul_13_128` can be almost completely removed. It remains necessary only for the final 255 polynomials that represent the coefficients of the `gf_mul_13_128`'s result. After initializing the matrix with the polynomial multiplication, Classic McEliece reduces it by means of the Gaussian elimination. Our optimization is based on two aspects, namely that the reduction's algorithm can be substituted with a better one like PLU, PLUQ or CUP decomposition [42] and that the reduction's basic operations can be optimized (therefore, we are again using our multiplication optimization). For the first aspect, the M4RIE library is the perfect tool. It offers three different Gaussian reductions: naive, ple, newton-john and, among them, the one performing better for small matrices is *naive*. The algorithm used for the multiplication in M4RIE is very inefficient compared to `gf_mul_13`, so we substituted it with `gf_mul_13`. The combination of the two techniques generates a huge speedup.

3.5. The public key

The data structures used in the public key's creation are two matrices. For *mceliece8192128* the main one is a 1664×128 matrix (`mat`), and the second one is a 1664×26 (`ops`), both of which hold 64-bit integer values. The algorithms we want to optimize share a basic logic in common: every operation performed on the data structures is a mask application, obtained with a logical `AND` operation (&) between one element of the matrix and one temporary element (the mask). Another common feature is the low dependency between operations and data; this means that we can produce an efficient parallel implementation of the same algorithms. These common features allow a simple and similar optimization on all of the targeted operations and, with a good memory handling, these improvements can grant a huge speedup in the computation of the public key.

Changing the implementation with a parallel one, may change the memory access pattern of the algorithm; rearranging rows and columns can drive to very different performances. In order to obtain a good speedup, we need to modify the matrices' memory layout to allow efficient access in their parallel implementation. Gaussian reduction and linear map computation use only the first 26 columns of `mat` and the entire `ops`. The row access on these matrices is efficient, the one in columns is not. The application of the linear map uses the last 102 columns of `mat` and the entire `ops`. Both row and column accesses to `mat` are efficient because now the number of columns is not as small as before.

The Gaussian elimination algorithm performs an iterative reduction of a sub-matrix in order to bring the initial matrix into echelon form. The sub-matrix reduction is a complete parallel task (100% capable, without dependencies among operations and data), which can be implemented efficiently with multiple threads computation. The synchronization step slows down the parallel computation, but this is unavoidable.

All the masks' applications depend entirely on one single column i. If this column is computed before the other 25 ones and the intermediate masks are saved in temporary memory, we can apply the masks to the other 25 columns completely in parallel without any synchronization among the columns. This opens up to a good parallel implementation that heavily differs from the classical parallel Gaussian implementation. Every thread accesses a portion of the 25 columns and independently applies the masks to the matrix. To have an efficient access to the columns we apply a transposition to the matrices. For the implementation, we choose the OpenMP since it offers the thread-pool concept and allows a fast and easy usage. We opt for a parallel section that encloses the entire algorithm, every thread works only on a subset of columns (the dimension is determined at runtime) and the threads synchronize on each, outermost, iteration by a `#pragma omp barrier`. The column, i, that generates dependency among the other columns, needs to be computed in a non parallel section and this can be realized with the `#pragma omp single` directive. The masks' applications work in parallel where every thread applies the masks to its columns subset. In the linear map's creation the mask is based on a read-only data and does not depend on previous values, therefore the application can be executed on all 26 columns without any synchronization construct. The implementation follows that for the Gaussian reduction; the entire algorithm is enclosed with a parallel region, every thread processes a subset of columns, but this time no synchronization is required, allowing the parallel computation to run at full speed. The last step of the public key creation is the linear map's application to the matrix `mat`.

The map application's complexity is higher than any other operation in the cipher and, thanks to the 100% parallel implementation, we expect a great performance boost. The parallelization is more or less the same as before, a unique parallel region that encloses the entire algorithm, with no synchronization because the mask depends on read-only data. The main problem is the temporary array that we need to store (`one_row`). In a sequential implementation we have only one array but, in parallel, we need one array for each thread. The memory cost of defining a new array for every thread is too high and so it is not affordable. A smarter approach is to use the first 26 columns of `mat` as a buffer to hold the intermediate values of `one_row`.

The memory poses another problem. At this point `ops` is not in the original form, but in its transposed one (`ops_t`). However, the original algorithm performs access on `ops`'s rows, which are inefficient on `ops_t`. To cope with this issue we have to transpose again the rows to their original form. In order to do that, we transpose `ops_t`'columns onto `mat_t`'rows. Thanks to this unusual transformation all memory accesses become efficient and the parallel implementation can run at full speed.

3.6. Testing and results

In this section, we test our improvements to see to which extent we can speedup the original implementation. The test machine is a Dell XPS 15 9560, with the following specifications:

- CPU: Intel i7-7700HQ, 2.8GHz to 3.8GHz, 4 Core;
- RAM: 8GB 2400MHz;
- GPU: Nvidia 1050 (not used by the cipher);
- OS: Ubuntu 19.10 virtualized with VMWare on a Windows 10 host system.

All tests have been conducted with the better performance mode offered by the machine, using the electrical grid instead of the battery, as the power source. The test of a specific optimization has been repeated many times, to compensate variations on the machine status.

As illustrated by the specifications of the XPS machine, the CPU has 4 physical cores and respectively 8 virtual threads. As stated before, the parallelism can be a great tool, but it requires a very careful setup. In theory, we could run the parallel optimizations with 8 different logical threads (as the CPU allows), but virtual threads share the same physical cores. This means that the threads will be in conflict for the same hardware resources, thing that, in cryptography, may become quite a big problem. In this scenario, the waiting imposed to the threads would dramatically slow down the computation, instead of speeding it up. With this in mind, the better configuration is to use the same amount of logical threads as the number of physical cores the CPU contains.

One can improve even further with the binding of every logical thread to a specific core: during the entire computation the thread will never switch to another core, reducing the management overhead. Thanks to these considerations, we have conducted all the following tests with 4 logical threads instead of 8. This also means that the obtained results can be further improved with a more powerful CPU. The optimization's performances are highly bounded to the hardware and configuration uses for the computation, a good mix of the two can deliver a great speedup as reported in the following test.

3.6.1. Private Key

Polynomial multiplication. This operation has been optimized in its two component `gf_mul_13` and `gf_mul_13_128`. Thanks to `PCLMULQQD`, we can remove almost 40 machine instructions from `gf_mul_13`; this translates to a very fast multiplication, which allows a good speed up on the private key computation since `gf_mul_13` is used also in the Gaussian reduction. To evaluate this optimization, we perform 10 millions multiplications and we report the results in table 3.1. The result of this test gives an idea on how much we gain from this dedicated, machine instruction, that directly executes a complex task. We have a reduction in the computation times of almost an order of magnitude.

The `gf_mul_13_128`'s optimization gives a speedup of approximately 60%. The following test concerns the timing of the entire matrix initialization phase. This means that a single test repeats the `gf_mul_13_128` operation 127 times. Table 3.2 reports the performance of 4000 consecutive test.

Table 3.1: Computation times of `gf_mul_13` over 10.000.000 tests

`gf_mul_13`	Original	Optimized
Total time (msec)	84,356	3,634

Table 3.2: Performance comparison of the entire polynomial multiplication phase over 4000 tests.

Percentile	Original (msec)	Optimized (msec)	**Speedup (%)**
25%	3,149	1,271	59,63
50%	3,211	1,281	60
75%	3,415	1,299	61,96

Gaussian reduction. Thanks to the very fast polynomial multiplication, the advanced reduction algorithm can achieve its maximum performance, with a speedup of almost 75%. This represents a huge increment of performance for the private key generation since the Gaussian reduction constitutes almost 67% of its computation time. Table 3.3 reports the results of 4000 tests, where the same matrix has been reduced with the two different implementations.

Table 3.3: Private key Gaussian reduction comparison over 4000 tests.

Percentile	Original (msec)	Optimized (msec)	**Speedup (%)**
25%	8,201	2,303	71,91
50%	8,320	2,313	72,20
75%	9,005	2,334	74

3.6.2. Public Key

The public key's test can be negatively affected by the machine status and configuration. Therefore, we conducted the following tests in the most stable and quiet environment possible, with no other applications running (other than the related OS).

Gaussian reduction. This parallel optimization is particularly difficult because of all the synchronization required to correctly execute the elimination algorithm; also, for the same reason, the performance speedup is not as relevant as the one obtained on the optimizations of the map creation and application. To allow a fast execution of this optimization, a change in the memory layout may be necessary. In table 3.4, the timings of the two versions are summarized; it is easy to see that this operation is very complex to optimize and even with a parallel implementation, the speedup barely reaches the 28%. The problem related to the machine status can be seen in the maximum computation time

registered, with 90,698 msec for the original implementation and 115,134 msec for the optimized one.

Table 3.4: Public key Gaussian reduction comparison, over 4000 tests.

Percentile	Original (msec)	Optimized (msec)	**Speedup (%)**
25%	37,329	28,151	24,58
50%	39,898	28,586	28,35
75%	43,073	35,103	18,50

Linear map creation. Thanks to the memory layout modifications, this operation can be executed 100% in parallel without any synchronization, thus producing very good performance. With these properties, the speedup is directly related with the number of cores/threads at disposal and having zero synchronization means to have a close to zero management overhead, so the threads can run at full speed producing good overall performance. Table 3.5 reports the timings of 4000 tests, the results indicate that this is the best optimization in terms of percentage increment, with a peak of almost 80% speedup.

Table 3.5: Generation of the linear map, over 4000 tests.

Percentile	Original (msec)	Optimized (msec)	**Speedup (%)**
25%	10,877	2,380	78
50%	11,117	2,398	78,42
75%	12,092	2,468	79,60

Application of the linear map. The same considerations made for the previous operation can be also applied to this one: full parallel implementation with no synchronization is required. As reported in table 3.6, the test indicates a lower percentage increment, but, since this operation is by far more complex and costly than all the others, the overall decrease of computation time is definitely the best one. Thanks to this optimization, the entire key-pair generation highly reduces its impact on the global execution time.

Table 3.6: Application of the linear map, over 4000 tests.

Percentile	Original (msec)	Optimized (msec)	**Speedup (%)**
25%	62,112	17,474	71,86
50%	63,216	17,624	72,12
75%	70,300	18,574	73,57

3.6.3. Key-pair

We give a global review of the two implementations' timings, indicating the overall optimization's performance and speedups. In table 3.7 we reported the timings of 2000 sequential tests. The results indicate a global speedup of 55% on the fourth quartile and this represents quite a good improvement: the execution times are almost halved.

Table 3.7: Timings compare, over 2000 tests.

Private key generation			
Percentile	Original (msec)	Optimized (msec)	**Speedup (%)**
25%	12,201	4,599	62,30
50%	12,356	4,628	62,50
75%	13,421	4,675	65,16
Public key generation			
Percentile	Original (msec)	Optimized (msec)	**Speedup (%)**
25%	111,852	49,388	55,84
50%	115,142	51,385	55,37
75%	127,750	67,702	47
Key-pair generation			
Percentile	Original (msec)	Optimized (msec)	**Speedup (%)**
25%	185,268	98,963	45,70
50%	191,184	107,776	43,63
75%	212,542	118,116	55,57

3.6.4. Memory consumption

In order to have a good optimization, the additional memory requirements need to be very low, to prevent the explosion of an already problematic parameter. In this section, we give a brief overview of the memory consumption of every defined optimization.

- **Polynomial multiplication**: the `gf_mul_13`'s optimization uses only the `PCLMULQDQ` instruction, so no additional memory is required; `gf_mul_13_128` requires the memorization of the entire intermediate results produced by the multiplication technique. In this configuration the intermediate results are 11309 25-bit polynomials (remember the removal of the modulo reduction); every polynomial is represented with a 32-bit variable, meaning a total memory consumption of about 44KB;
- **Private-key Gaussian reduction**: this optimization is based on the `naive reduction` function offered by the `M4RIE` library. Since the memory requirement for this optimization depends entirely on a third party library,

we do not include this quantity into the total count, because it could change basing on the library's version or future modification;

- **Public-key Gaussian reduction**: this optimization requires the memorization of two different columns (holding the masks which generate the dependencies) from the `mat` matrix. The matrix has 1664 row and every element is a 64-bit variable, therefore the additional memory required is 25KB.
- **Computation of the linear map**: no additional memory is required.
- **Application of the linear map**: thanks to the smart reuse of `ops_t` no further memory is required.

The analysis shows that the total additional memory used by the optimizations is, approximately, 70KB, while the total amount of memory required by *mceliece8192128* is 3.3MB (1.3MB for the public key memorization + 2MB for the matrix used to compute the public key). The additional memory required by the optimizations (excluding the `M4RIE`) is the 2% of the global memory required by Classic McEliece. At the end, the optimization's evaluation is very good: using almost 2% additional memory, we can halve the execution time with a consumer CPU.

Part II

Pre-image attack on SHA-1

4. SAT

In this chapter, we are going to describe the satisfiability of a generic boolean formula. The notion of satisfiability (SAT) was first introduced by Alfred Tarski in the 1930s [11], but only in the 1960s a precise algorithm capable of solving a satisfiability problem in a reasonable amount of time [28] appeared. The proof that SAT is a NP-complete problem is due to Cook, in 1971 [24].

Although we are interested in using SAT for cryptanalysis, it can also be used to model and solve problems in other fields such as the following ones.

- Hardware model checking: a model that checks automatically whether a hardware design satisfies a given specification. SAT solvers have improved the scalability of symbolic model checking, a technique that represents a set of states and transition relations of circuits as a formula. Indeed, this avoids the expensive enumeration of explicit states.

- Software systems: model to check dependencies among packages that constitute a program. More precisely, SAT solvers are used to determine whether a software configuration is safe or not. The dependencies among packages are modeled using propositional logic. For example, package A depends on package B and package C can be expressed by the clauses ($\neg A \vee B$ and $\neg A \vee C$) [47]. SAT can also be used for installation optimization. For example, it can be used to propose an installation with the minimum number of packages or to reduce the amount of space on the disk [47].

After giving a formal definition of the SAT problem, we describe Davis, Putnam, Logemann, Loveland (DPLL) and Conflict-Driven Clause Learning (CDCL), two of the many algorithms used to solve the SAT problem. Finally, we will describe DIMACS, the widespread standard when implementing the solving algorithms.

4.1. The problem

In order to give a formal definition of the boolean satisfiability problem, we start by recalling the well known convention: 0 is the value for false and 1 for true. In the following, there will be no other values.

Definition 4.1. A *variable* is an unknown truth value, so, its domain is $\{0, 1\}$.

This means that taking a generic variable x, it could assume only the value 0 or the value 1.

As well as in algebra, we want now to define some operations in order to combine variables. We are going to define just three of them because it is enough in our discussion.

Definition 4.2. Let x_1 and x_2 two variables. We define the following three operations.

- `AND`: we will use the symbol $\wedge$ and we define $x_0 \wedge x_1 = 1$ if $x_0 = 1$ and $x_1 = 1$, 0 otherwise.
- `OR`: we will use the symbol $\vee$ and we define $x_0 \vee x_1 = 0$ if $x_0 = 0$ and $x_1 = 0$, 1 otherwise.
- `NOT`: we will use the symbol $\neg$ and we define $\neg x_0 = 1$ if $x_0 = 0$ and viceversa.

It table 4.1, we resume the complete evaluation of the three operations.

Table 4.1: evaluation of `AND` ($\wedge$), `OR` ($\vee$) and `NOT` ($\neg$).

x_0	x_1	$x_0 \wedge x_1$
0	0	0
0	1	0
1	0	0
1	1	1

x_0	x_1	$x_0 \vee x_1$
0	0	0
0	1	1
1	0	1
1	1	1

x_0	$\neg x_0$
0	1
1	0

Sometimes each of the operations $\wedge$, $\vee$, $\neg$ are called conjuction, disjunction and negation respectively. In general, we can call each of them a *connective*.

At this time, we have variables and ways to combine them, so, we can now give the definition [70] of what a formula is.

Definition 4.3. Every variable is a *formula*. If F_1 and F_2 are formulas then $F_1 \wedge F_2$, $F_1 \vee F_2$ and $\neg F_1$ are formulas.

Moreover, the rules in definition 4.3 constitutes a minimal set in order to build every logic formula.

It is trivial that for every formula using n variables, we have 2^n possible evaluation, corresponding to the possibility of assigning 0 or 1 to every variable. We can easily see this through an example.

Example 4.1 (Formula)**.** We take the formula $(x_0 \vee x_1) \wedge (x_0 \vee x_2)$. We see that there are $2^3 = 8$ possible combinations and three of them evaluates the formula to false: $\{x_0 = 0, x_1 = 0, x_2 = 0\}$, $\{x_0 = 0, x_1 = 0, x_2 = 1\}$, $\{x_0 = 0, x_1 = 1, x_2 = 0\}$. It can be checked using rules in table 4.1.

We turn the above discussion in a formal way.

Definition 4.4. An *assignment* is a mapping from the variables to the values 0 or 1. This is denoted as $\{x_0 = v_0, x_1 = v_1, x_2 = v_2, \ldots\}$, where x_i are variables and $v_i \in \{0, 1\}$.

We can now define the problem.

Problem 4.1 (Boolean SATisfiability)**.** Given a formula, finding an assignment such that the formula evaluates to 1, in this case we will say that the formula is satisfiable.

The problem could be solved simply going through every possible solution, but the computational time grows exponentially with respect to the cardinality of the variables set.

Before describing the main approaches to solve the SAT problem, we introduce a standard representation [70] of it.

Definition 4.5 (CNF formula, clauses, literals, pure literal)**.** A boolean formula F is in Conjunctive Normal Form (CNF) when it is written as

$$F = C_0 \wedge C_1 \wedge \dots C_{m-1}.$$

C_i are again formulas called clauses and must have the following form

$$C_i = l_{i,0} \vee l_{i,1} \vee \dots \vee l_{i,k-1},$$

where the $l_{i,j}$ are literals, that is variables or variables negation:

$$l_{i,j} \in \{x_0, x_1, \dots, x_{n-1}\} \cup \{\neg x_0, \neg x_1, \dots, \neg x_{n-1}\}.$$

Moreover, if a variable always occur in either affirmative or negative form in a given CNF formula, we say that it is a pure literal.

It is widely accepted and a standard de facto that every formula has to be transformed in CNF before trying to guess an assignment. Moreover, we remark that standard input file for implementations has one clause per row with no $\vee$ connective and uses numbers instead of labels for variables [40]. We will go into details in section 4.3.

4.2. Main solving algorithms

In literature, there is a wide range of solving techniques for SAT problem, but many of them are refinements of two main approaches.

4.2.1. DPLL

The first we examine is Davis, Putnam, Logemann, Loveland (DPLL) algorithm [28, 27]. It is based on backtracking, so, it goes basically through the whole tree of solutions by iteratively rejecting inconsistent solutions.

DPLL is based on the usage of the following two procedures. Suppose that a CNF form F is given.

1. *Unit propagation.* If there is in F a clause containing only one literal, we simply choose the right assignment for the variable and propagate it, i.e. we substitute the value in all other clauses of F.

2. *Pure literal elimination.* If there is a pure literal in F, we remove clauses containing it. This is quite obvious, in fact, we can assign to the corresponding variable the value that evaluates the pure literal to 1.

Algorithm 1: Davis-Putnam-Logemann-Loveland

Input: F, a CNF formula
Output: SAT and A, a satisfying assignment otherwise UNSAT

```
A = ∅
Procedure DPLL(F, A):
    if F = 1 then
        return SAT and A
    if F = 0 then
        return UNSAT
    if F contains a one literal clause then
        add to A the assignment making literal true and propagate
    if F contains a pure literal then
        add to A the assignment making literal true and remove clauses
          containing it
    choose a variable a
    assign a value to this variable
    substitute such a value over F
    return DPLL(F, A ∪ {a = 0}) ∨ DPLL(F, A ∪ {a = 1})
```

If we find an assignment of all variables such that F evaluates to true, we can return it and say that the CNF is satisfiable, otherwise, we say that it is unsatisfiable. The unsatisfiability is returned when we find a contradiction such as $x \wedge \neg x$. We say that a conflict is detected and it is impossible to find an assignment for F. A pseudo-code for DPLL can be found in algorithm 1.

We are going to show a little example of resolution using DPLL in order to remark the importance of the two mentioned rules.

Example 4.2 (Satisfy CNF using DPLL)**.** Find a satisfying assignment for the following formula F.

$$\begin{aligned} F = &(x_0 \vee x_1 \vee x_2 \vee x_4) \\ &\wedge (x_2 \vee x_3) \\ &\wedge (x_2 \vee \neg x_3) \\ &\wedge (x_1 \vee \neg x_2 \vee x_3) \\ &\wedge (\neg x_1 \vee \neg x_2 \vee x_3) \\ &\wedge (\neg x_0 \vee x_1 \vee \neg x_2) \\ &\wedge (\neg x_0 \vee \neg x_1 \vee x_2) \\ &\wedge x_4 \end{aligned}$$

Note that F is already in CNF. In order to clarify the formula, each clause is in a separate row.

- First of all, we see that the last clause is a unit clause, so, by applying unit propagation rule, we set $x_4 = 1$ and propagate its value in the whole F.
- Then, $\neg x_0$ becomes a pure literal, so, we set $x_0 = 0$ in force of pure elimination literal.

- We have now to make a decision. We set $x_1 = 0$.
- Another decision is unavoidable, so, we choose $x_2 = 0$, but doing this, a conflict between second and third clause is thrown. We backtrack to $x_1 = 0$ and we set $x_2 = 1$.
- The only variable left is x_3. It is now trivial that $x_3 = 1$.

We remark the importance of unit propagation and pure literal elimination, in fact, the former clears the CNF at the beginning of the example, as well as, in general, during the algorithm, see e.g. the last step of the example, and the latter keeps high the number of free variables, for example setting $x_1 = 0$.

4.2.2. CDCL

The second main strategy to solve problem 4.1 is called Conflict-Driven Clause Learning (CDCL) [54].

The basic idea is the same as DPLL, but with a slight change. CDCL does not backtrack but *backjump*, meaning that it does not methodically go through every node of the tree of the solution. Instead, it jumps to the appropriate decision level. To achieve this, CDCL keeps track of the assignments in a more accurate way than DPLL. In fact, it constructs the so called implication graph. We need a preliminary definition.

Definition 4.6. Given a clause C containing a variable x, the *antecedent assignments* of x, denoted as $A_C(x)$, are defined as the set of assignments to variables other than x with literals in C.

Intuitively, $A_C(x_1)$ designates those variable assignments that are directly responsible for implying the assignment of x_0 in C. For example, let $C = x_0 \vee x_1 \vee \neg x_2$. We can say that $A_C(x_0) = \{\neg x_1, x_2\}$, by writing this we mean $A_C(x_0) = \{x_1 = 0, x_2 = 1\}$.

We define I, the *implication graph*, as follows:

- I is a directed graph;
- every node in I is labeled with an assignment;
- the predecessors of vertex with an assignment for x in I are the antecedent assignments $A(x)$;
- vertices corresponding to decision assignments have no predecessors.[1]

Besides this, CDCL keeps track also about any arbitrary assignment in a binary tree. Every arbitrary assignment creates a new level in this tree and every implied assignment is put at the same level it depends on.

When a conflict is found, we need to find the cut in graph for the conflict. Suppose that we find a conflict (CD part) in variable x_4, i.e. we have both x_4 and $\neg x_4$, as unit clauses. Then, we construct a new clause (CL part) in order to avoid wrong assignment in this way: we take both $A(x_4)$ and $A(\neg x_4)$ and we construct the negation of the conjuction of literals in $A(x_4) \cup A(\neg x_4)$.

[1] An assignment can derive either from unit propagation, so, it has predecessors, or from an arbitrary decision of SAT solver, in this case it cannot have predecessors (this is the meaning of adjective *decision*).

Algorithm 2: Clause-Driven Clause Learning

Input: F, a CNF formula and an assignment $\{x_i = v_i\}$
Output: SAT and A, a satisfying assignment otherwise UNSAT

```
1  A = {x_i = v_i}
2  Procedure CDCL(F,A):
3      if unit propagation of {x_i = v_i} == conflict then
4          return UNSAT
5      decision level = 0
6      while not all variables are assigned do
7          choose new assignment: {x_j = v_j}
8          decision level += 1
9          A = A ∪ {x_j = v_j}
10         if unit propagation of {x_j = v_j} == conflict then
11             if decision level of the conflict cannot be found then
12                 return UNSAT
13             else
14                 backjump to decision level of the conflict
15     return SAT and A
```

Example 4.3 (Clause Learning)**.** Suppose we have a CNF containing, among others, two clauses:

$$\ldots \wedge (\neg x_0 \vee \neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge \ldots$$

Suppose also that a conflict is raised due to $A(x_3) = \{x_0, x_1\}$ and $A(\neg x_3) = \{x_1, x_2\}$. We write the negation of the conjunction of literals in $A(x_3) \cup A(\neg x_3)$

$$\neg(x_0 \wedge x_1 \wedge x_1 \wedge \neg x_2),$$

in other words

$$\neg x_0 \vee \neg x_1 \vee x_2.$$

We add it to the CNF. Then, we search for the lowest level among those of the variables and backjump to it, reversing the assignment.

We can summarize the technique in the algorithm 2.

4.3. Implementations

DIMACS is the *de facto* standard for input and output CNF formula when implementing a SAT algorithm. There is no reference paper in which we can find all the rules for the format, but they are few and widespread all over the papers about the topic of SAT solvers as well as in their manuals. We summarize the main rules of the format. Regarding the input, we have the following:

- the extension should be `.cnf`;
- comment lines start with `c` char, e.g.

```
c This is a comment
```

- there must be a single-line mandatory header of the form

  ```
  p cnf <number_of_variables> <number_of_clauses>
  ```

  where `<number_of_variables>` and `<number_of_clauses>` are positive integers whose meaning is quite clear;

- clauses appear immediately after the header and are sequence of integers separated by a single space and terminated by `0`, e.g.

  $$\begin{array}{lcl} x_2 \vee \neg x_3 \vee \neg x_4 & & \texttt{2 -3 -4 0} \\ x_4 & \longrightarrow & \texttt{4 0} \\ x_1 \vee \neg x_3 & & \texttt{1 -3 0} \end{array}$$

The output has this format:

- comment lines start with `c` char, e.g.

  ```
  c This is a comment
  ```

  this is the same convention for the input file;

- timing information should be in lines starting with `t` char, e.g.

  ```
  t Time spent:  711s
  ```

  but they can be considered like comments;

- the line of the satisfiability starts with `s` char and has the form `s value` where value can be `SATISFIABLE` or `UNSATISFIABLE` (or `INDETERMINATE` when an exception occurs, e.g., manual interrupt), e.g.

  ```
  s SATISFIABLE
  ```

- lines of the assignment of variables starts with `v` char, e.g. suppose that variables `1`, `2` and `3` are found `false` and `4` is found `true`, then the line for the assignment of variables is

  ```
  v -1 -2 -3 4 0
  ```

Even if the experimental results we will show in chapter 6 are conducted with the CryptoMiniSat software, we briefly describe the most famous implementations.

The first SAT solver to cite is undoubtedly MiniSat [32]. It defines itself fast and lightweight. It is written only in `C` and `C++` and this contributes with a great speedup. Unfortunately, the last update dates back to more than ten years ago. Despite this, it can be easily found in many Linux distribution repositories, such as Ubuntu and Fedora.

CryptoMiniSat [73], derived from MiniSat, is a SAT solver dedicated to cryptography. This comes from its capability to handle pure xor expression

without transforming them in CNF. It has an high level of granularity when handling all SAT and hardware features, in fact, we can choose between various restart policies [69], such as geometry, glucose and luby and control sizes of matrices in the Gauss simplification of xor clauses. The amount of threads and memory used are customizable.

Glucose [5] is a SAT solver heavily based on MiniSat. It makes a clever use the CDCL: it not only learns clauses, but also removes bad ones, keeping the CNF formula clean and optimized. Moreover, in SAT 2011 competition, it was placed fourth in the rank of parallel solvers despite the fact it was sequential.

Yices-sat [30] is a minimal SAT solver. It is part of a bigger project that focuses on SMT. It is not customizable but its performances are comparable to the Minisat and CryptoMiniSat ones. As for Minisat, it can be easily installed from any Linux distribution.

MathSAT [23] is another SMT solver that works very well also for SAT problems and has high performances. It does not have as many feature as CryptoMiniSat, but it is still highly configurable. We can decide, for example, restart policies, branching options and preprocessing usage. Unfortunately, it is closed source.

We remark that all of the SAT solvers can be used in all three major platforms: Linux, Mac OS and Windows. Some other details on SAT solvers can be found in table 4.2.

Table 4.2: summary of the main SAT solvers.

Name	*License*	*Features*	*Interfaces*
CryptoMiniSat	MIT	handling pure xor storing learnt clauses multithreading cryptography oriented highly customizable	command line C++ Python
Gluecose	MIT	efficient clause learning	command line
MathSAT	Proprietary	highly customizable (preprocessor in particular)	command line C
MiniSat	MIT	fast lightweight	command line C++
Yices-sat	GNU GPL v3	not customizable good performances	command line

5. SHA-1

In this chapter, we are going to briefly review the mechanism of the cryptographic hash function called SHA-1. A hash function is a function that, given an arbitrary length sequence of bits as input, which is called *message*, returns a fixed length sequence of bits as output, which is called *digest*. Actually, there is a bound in the message length, but it is so high that it is very unlikely that it could be reached.

A *cryptographic* hash function h is a hash function that satisfies three properties.

1. *First pre-image resistance*: given y, it is computationally infeasible to find an x such that $h(x) = y$.
2. *Second pre-image resistance*: given x, it is computationally infeasible to find an x' such that $h(x) = h(x')$.
3. *Collision resistance*: it is computationally infeasible to find x and x' such that $h(x) = h(x')$.

Even if collisions for full SHA-1 has been found [76], the SHA-1 hash function is still widely used in algorihms such as HMAC useful for instance in Wi-Fi Protocol Access (WPA) [77] or in disk encryption like LUKS [18].

5.1. Notation

In order to explain the mechanism of SHA-1, we are going to follow the orginal Request For Comments (RFC) 3174 [31].

Basic Notation. We now describe the notation for numbers.

- An *hex digit* is an element of the set $\{\texttt{0}, \texttt{1}, \ldots, \texttt{9}, \texttt{A}, \texttt{B}, \ldots, \texttt{F}\}$. Each of them is the representation of a 4-bit string. For instance $\texttt{7} = \texttt{0111}$, $\texttt{A} = \texttt{1010}$.
- A *word* is a 32-bit string, therefore it can be represented as an 8-hex digit string. For example

  $$\texttt{00000011110011001010110100010001} = \texttt{0x03CCAD11}.$$

  Note that we will prepend `0x` chars to every hex digit string. We will use capitalized letters for words except F that will be used for a specific function and M that will be used for blocks.
- A *block* will be a 512-bit string. It can be viewed as a sequence of 16 words.

Operations. Moreover we define the set of operations between words that we will use in the following.

- The `AND`, `OR`, `XOR` and `NOT` will be simply applied bit per bit between words.
- We will use $X \boxplus Y$ to denote the mod 32 addition between the numbers represented by the words.
- The circular left shift operation $X \lll n$, where X is a word and n is an integer such that $0 \leq n < 32$, is defined by

  $$X \lll n = (X \ll n) \texttt{ OR } (X \gg 32 - n).$$

  In the above, $X \ll n$ is obtained as follows: discard the left-most n bits of X and then pad the result with n zeroes on the right (the result will still be 32 bits). $X \gg n$ is obtained by discarding the right-most n bits of X and then padding the result with n zeroes on the left. Thus $X \lll n$ is equivalent to a circular shift of X by n positions to the left.

Functions. SHA-1 uses 4 functions depending on a parameter and acting on three words.

- $F_t(B, C, D) = (B \texttt{ AND } C) \texttt{ OR } ((\texttt{NOT } B) \texttt{ AND } C)$ $\quad 0 \leq t \leq 19$
- $F_t(B, C, D) = B \texttt{ XOR } C \texttt{ XOR } D$ $\quad 20 \leq t \leq 39$
- $F_t(B, C, D) = (B \texttt{ AND } C) \texttt{ OR } (B \texttt{ AND } D) \texttt{ OR } (C \texttt{ AND } D)$ $\quad 40 \leq t \leq 59$
- $F_t(B, C, D) = B \texttt{ XOR } C \texttt{ XOR } D$ $\quad 60 \leq t \leq 79$

Constants. SHA-1 makes use of 4 constants. Similarly to the functions, the choice of the constant depends on a parameter.

- $K_t = \texttt{0x5A827999}$ $\quad 0 \leq t \leq 19$
- $K_t = \texttt{0x6ED9EBA1}$ $\quad 20 \leq t \leq 39$
- $K_t = \texttt{0x8F1BBCDC}$ $\quad 40 \leq t \leq 59$
- $K_t = \texttt{0xCA62C1D6}$ $\quad 60 \leq t \leq 79$

5.2. Computation

SHA-1 has been designed following the so called Merkle-Damgård construction [59]. This is a method for building collision-resistant cryptographic hash functions from collision-resistant one-way compression functions. Basically, we can summarize the Merkle-Damgård construction in the five steps:

1. divide the message M in n blocks $M_1, \ldots, M_n$ having the same length; if it is not possible, choose a suitable padding;
2. choose an initial value IV_0;
3. compute IV_i applying the compression function to IV_{i-1} for all $i = 1, 2, \ldots, n$;

4. the output is IV_n.

Merkle [58] and Damgård [25] have independently proven that if the compression function is collision resistant then the above procedure defines another collision resistant function.

SHA-1 algorithm is based on the Merkle-Damgård construction but in order to use it, the message needs to be padded, thus, we can be described the SHA-1 algorithm in two stages: pre-processing and digest computation.

Pre-processing. Since a Merkle-Damgård construction acts on inputs whose number of bits is multiple of a fixed amount, we need to pad the input in some way. Therefore, suppose a message has l bits. We perform the following steps:

- append `1` getting a sequence of bit whose length is $l + 1$;
- append p zeros such that

$$l + 1 + p \equiv 448 \pmod{512};$$

- append the 2-word representation of l, the number of bits in the original message.

Note that after the third step we eventually reach a padded message whose number of bits is multiple of 512, in fact we have

$$l + 1 + p + 64 \equiv 448 + 64 = 512 \equiv 0 \pmod{512}.$$

Thus, the padded message will contain n blocks for some $n > 0$ and we will denote them as $M_1, M_2, \ldots, M_n$.

Digest computation. Before processing any blocks, the H's are initialized as follows:

- $H_0 =$ `0x67452301`
- $H_1 =$ `0xEFCDAB89`
- $H_2 =$ `0x98BADCFE`
- $H_3 =$ `0x10325476`
- $H_4 =$ `0xC3D2E1F0`.

Now $M_1, M_2, \ldots, M_n$ are processed. To process M_i, we proceed as follows:

a. Divide M_i into 16 words $W_0, W_1, \ldots, W_{15}$, where W_0 is the left-most word.

b. For $t = 16$ to 79 let

$$W_t = (W_{t-3} \boxplus W_{t-8} \boxplus W_{t-14} \boxplus W_{t-16}) \lll 1.$$

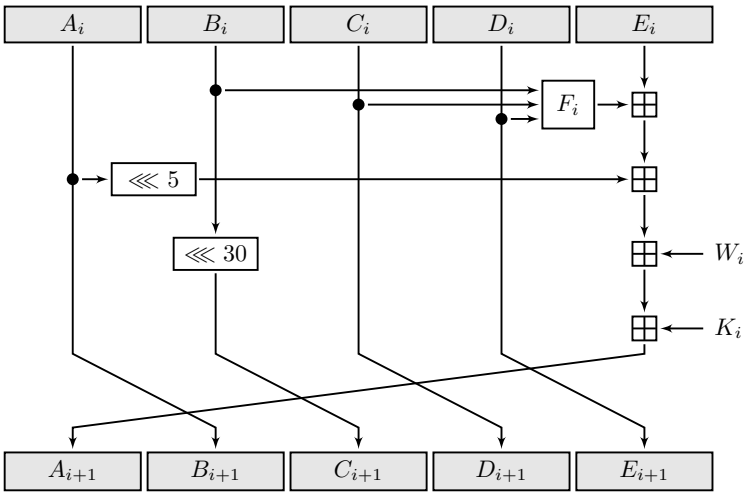
c. Let $A = H_0$, $B = H_1$, $C = H_2$, $D = H_3$, $E = H_4$.

d. For $t = 0$ to 79 do

$T = (A \lll 5) \boxplus F_t(B, C, D) \boxplus E \boxplus W_t \boxplus K_t;$
$E = D;$
$D = C;$
$C = B \lll 30;$
$B = A;$
$A = T;$

Refer to the figure 5.1 for a graphical representation of this step

Figure 5.1: SHA-1: round function.



e. Let $H_0 = H_0 \boxplus A$, $H_1 = H_1 \boxplus B$, $H_2 = H_2 \boxplus C$, $H_3 = H_3 \boxplus D$, $H_4 = H_4 \boxplus E$.

After processing M_n, the message digest is the 160-bit string represented by the 5 words: H_0, H_1, H_2, H_3, H_4.

6. Pre-image attack

In this chapter, we are going to analyze how SHA-1 can be modeled as a satisfiability problem. Furthermore, we explore a new way of computing the first pre-image, measuring how many rounds can be reversed [7].

6.1. Related works

Since 2004, a vast literature studied the collision resistance of SHA-1 and of its predecessor SHA-0, see e.g. [76, 50] and their references, that resulted in the 2017 discovery of the first collision of the full hash by Stevens et al. [76] in a computation that required $2^{63.1}$ SHA-1 applications.

SHA-1 is also subject to pre-image attacks. To date, none of these attacks to the full SHA-1 algorithm have been published in the literature. Anyway, many applications of hash functions rely on the pre-image resistance and breaking this kind of resistance also allows to find collisions.

Theoretical pre-image attack history on SHA-1 starts in 2008 with the work of De Cannière and Rechberger [17] (attack complexity of 2^{157} for 45-round reduced SHA-1). In [3], the authors exploited a meet-in-the-middle attack to present a pre-image attack on 48 out of 80 rounds with complexity of 2^{159}. Their work was improved by Knellwolf and Khovratovich, thanks to a differential view on the MiTM approach ($2^{158.7}$ for 57-round reduced SHA-1), and finally by Espitan et al. in [33] thanks to higher order differential (62-round two-block pre-image with padding with complexity $2^{159.3}$).

In all the above mentioned approaches, the purpose of the authors is to show that the analyzed hash function is not ideal, but the computational cost of the attacks is really high and practically out of reach.

One less explored approach in studying the security of a hash function is to determine which is the maximum number of rounds for which a pre-image of the hash can be found "in practice". Of course, this expression mostly depends on the computational power the attacker has at his disposal. This second approach is very common in public key cryptography, where, for example, challenges are proposed to break the largest possible parameters of a hard mathematical problem such as, among many others, factorization [46] or decoding [4]. The most recent work we could find in this direction is from Nejati et al. in 2017 [61]. Here, the problem of finding a pre-image of a reduced version of SHA-1 is modeled into a SAT instance, and then different SAT solvers are used to solve the instance. The authors claim that, with this method, pre-images for more than 23 steps cannot be constructed in a reasonable amount of time even with the latest techniques and hardware [63, 48, 49]. Moreover, they also assert 27

round is their best result for a partial pre-image attack when leaving only 40 free bits of the input message.

6.2. Modelling SHA-1 as satisfiability problem

We have seen in chapter 5 that SHA-1 compresses arbitrary length inputs. Even if the number of blocks can be arbitrary, we consider in our discussion a one block message M. Therefore, SHA-1 can be represented as a function $h : \{0,1\}^{512} \to \{0,1\}^{160}$ with $h(M) = d$, where M is a message block and d is the digest corresponding to M. Theoretically, it is possible to explicitly express each digest bit as a boolean expression depending on M, and to gather 160 such expressions in a system representing the full hash function. Although 160 equations are sufficient to do that, the number of literals of each expression is exponential in the number of bits in M and IV — it is exponential even if we applied heuristics [80] — making this approach not feasible in practice. A better solution is to model each component of the cipher, by expressing the output bit of each component as an expression of its input bits.

In what follows, we detail how to construct the SAT model that we use to find pre-images of SHA-1.

Proposition 6.1. *Given two 32-bits words x and y, the system of boolean formulas that represents the modular addition modulo 2^{32} between x and y and returning the word z is:*

$$\begin{cases} c_0 = x_0 \wedge y_0 & \\ c_i = (x_i \wedge y_i) \vee (x_i \wedge c_{i-1}) \vee (y_i \wedge c_{i-1}) & i = 1, \ldots, 30 \\ z_0 = x_0 \oplus y_0 & \\ z_i = x_i \oplus y_i \oplus c_{i-1} & i = 1, \ldots, 31 \end{cases}$$

The index i denotes the bit number where the MSB is indexed by 31.

a Input message M is divided in sixteen 32-bit words $W_0, \ldots, W_{15}$. Every bit of M is assigned to a variable, resulting in **512** variables for a one-block message.

b Other sixty-four words $W_{16}, \ldots, W_{79}$ are computed through the message expansion. For every 32-bit word we need 32 equations. The total number of equations required is $64 \cdot 32 =$ **2048**. **2048** variables are introduced.

c 160 variables are added to represent the first state, that is A_0, B_0, C_0, D_0, E_0. The IVs H_0, H_1, H_2, H_3, H_4 are assigned to this state. To do so, we need $5 \cdot 32 =$ **160** equations and $5 \cdot 32 \cdot 2 =$ **320** new variables.

d The SHA-1 step function is obtained as follows. Notice that the most important operation is the addition modulo 2^{32}. We split the computation of T, a 32-bit word, in four additions modulo 2^{32}.

$$\begin{cases} t_{t-1} = K_{(t-1)} \boxplus E_{t-1} \\ u_{t-1} = t_{t-1} \boxplus F_{t-1}(B_{t-1}, C_{t-1}, D_{t-1}) \\ v_{t-1} = u_{t-1} \boxplus A_{t-1} \\ T_{t-1} = v_{t-1} \boxplus W_{t-1} \\ E_t = D_{t-1} \\ D_t = C_{t-1} \\ C_t = B_{t-1} \lll 30 \\ B_t = A_{t-1} \\ A_t = T_{t-1} \end{cases} \tag{6.1}$$

Using 6.1, four additions modulo 2^{32} are represented by $4 \cdot 63 = 252$ equations. Moreover, we have five variables reassignments, namely $5 \cdot 32 = 160$ equations, hence $252 + 160 = 412$ equations per round, that means a total of $412 \cdot 80 =$ **32960** equations. For each addition modulo 2^{32} we need an extra variable for the carry bit, therefore $(16 + 4) \cdot 32 = 640$ variables per round are required, that means a total of $640 \cdot 80 = 51200$ variables. Notice that such number includes the 160 variables for each bit of A_0, B_0, C_0, D_0, and E_0, hence it can be reduced to $51200 - 160 =$ **51040**.

e We rewrite the final step: $Z_0 = H_0 \boxplus A$, $Z_1 = H_1 \boxplus B$, $Z_2 = H_2 \boxplus C$, $Z_3 = H_3 \boxplus D$, $Z_4 = H_4 \boxplus E$. Again, using 6.1, these five additions modulo 2^{32} require $5 \cdot 63 =$ **315** equations. The variables used in these modular additions come from the initial state and from the last round of step **d** respectively. Therefore, we only need Z for each word and each bit, namely $5 \cdot 32 =$ **160** variables.

To sum up, our model is represented by a system of **35483** equations and **54080** variables that can be simplified.

In particular, as described in [75, 39] we can substitute constants, input values and remove equations with different names but same meaning. Notice that if we have $x = 0$ and $x + y + z = 0$, after substituting the value of x in the second equation, we get $y = z$ and we can substitute the occurrences of y. Therefore, this strategy is repeated iteratively until no more substitution is possible.

In order to speed up the execution of our model, we tried to understand the relationship between variables that appear in our CNF. We suppose that the more a variable is connected to other variables, the more important it will be because its truth value will affect many others. In our specific case, the number of connections of a variable i is defined by the number of variables that appear in the same clause of i.

Because SAT solvers usually work with a top-down approach, we rearrange the CNF by placing the clauses that contains the most connected variables at the top. Doing so, the SAT solver will immediately assign a value to these variables, providing an answer to our satisfiability problem in a shorter time. Looking closely at table 6.1, it is possible to measure this speed-up.

Table 6.1: SHA1 pre-image attack before and after the optimizations, 440 bits free

Number of Rounds	After Simplification	After Reorganization Clauses
19	0.24s	0.20s
20	0.681s	0.61s
21	403.334s	335.93s

6.3. Experimental results

We implemented a Python tool called *Shanatomy* [72], to help us analyze the boolean representation of SHA-1.

The tool can be used to generate the boolean representation of SHA-1 variants operating on words with any number of bits. Moreover, it translates the boolean system into DIMACS standard, commonly used by SAT solvers. Handling the extension of the SAT solver, the tool can also preserve oplus operations (which usually is not part of the DIMACS standard). The advantages of keeping opluss are obvious: the solver has much less equations since a oplus with n inputs needs 2^{n-1} equations to be represented.

In order to test the efficiency of a pre-image attack to reduced versions of SHA-1, we choose CryptoMiniSat5 [73]. In the plethora of SAT solvers, this is not a random choice. Indeed, we experimentally observed that MiniSat [32], MathSat [23] and Yices [30] seems to perform better than CryptoMiniSat up to round 20, but they take more time than CryptoMiniSat to compute round 21 and above.

We run our code on a server equipped with (CPU) 2 x Intel ® Xeon ® Gold 6258R @2.70GHz, (Memory) 768GB 2933MHz, (OS) Ubuntu 18.04.5 LTS and on a cluster equipped with (CPU) 128 x IBM POWER9 AC922 @3.1GHz, (Memory) 256GB 2666 MHz, (OS) Red Hat Enterprise Linux Server 7.6. The SAT Solver adopted is Cryptominisat v5.8.0.

In our testing activities we run our code using all combinations between four polarity modes (auto, false, random, true) and three restart policies (geom, glue, luby), enabling the oplus extension of CryptoMiniSat. In order to run many SAT instances and compare their running time, we set two upper thresholds. Such thresholds are set on the basis of the average execution times of solved instances measured on our server. More precisely, we set 10 seconds as maximum execution time for round 20 and 180 seconds for round 21. For each polarity mode (four), each restart policy (three) and each hardware configuration (four), we executed 100 runs for a total of $4 \times 3 \times 4 \times 100 = 4800$ execution times collected. Average values are reported in table 6.2 and table 6.3. Notice that we highlighted with grey color the number of pre-images found (out of 100), and we used the word "`n.o.t.`" to indicate the number of threads used. Due to space constraint we omit data collected on our cluster where we have not found significant new results but only better timings.

Experimental results (see tables 6.2 and 6.3) show that (a) although there

Table 6.2: Server: Average times (secs) to compute a pre-image, 440 bits free.

polar	restart	n.o.t.	round 20	round 21
auto	geom	1	96/2.640	16/167.619
		2	99/1.660	29/157.426
		4	100/1.138	36/149.510
		8	100/0.926	43/146.887
	glue	1	97/2.662	23/163.576
		2	99/2.074	19/166.040
		4	100/1.421	16/167.765
		8	100/1.224	19/167.341
	luby	1	94/3.389	26/160.880
		2	100/1.632	23/162.603
		4	100/1.437	23/163.843
		8	100/1.036	35/151.600
false	geom	1	95/2.558	17/171.267
		2	99/1.837	11/173.127
		4	100/1.357	20/165.835
		8	100/0.940	29/156.152
	glue	1	91/4.062	12/173.967
		2	100/1.801	18/167.625
		4	100/1.544	9/176.643
		8	100/1.210	13/172.139
	luby	1	97/3.304	8/174.264
		2	100/1.686	14/168.858
		4	100/1.237	18/167.466
		8	100/1.044	25/163.203

isn't a combination of "polarity mode–restart policy" which always performs better than others, in our testing activities, on average, we achieved better results with "auto", "random" (polarity mode) and "geom" (restart policy). Notice that we received evidence of these results measuring the execution time; (b) sometimes we had indirect evidence of the performance improvement by measuring the number of pre-images found. In this case, also the combination "true–luby", round 21, is noteworthy; (c) a greater number of threads allows to decrease the execution time, but this does not happen in a linear way. This behaviour can be observed analyzing the computational time spent to execute, for example, round 21, [auto, geom, 1-8] and [rnd, geom, 1-8].

Interestingly, the difficulty of finding a pre-image on round 22, 440 input bits free, is sensibly higher. Indeed, our server was able to find only 3 pre-images (out of 100) in less than one hour using 32 threads. Therefore, to find pre-images for a larger number of rounds, we reduce the number of free input bits as near as possible to 160 — recall that the complexity of a brute-force pre-image attack is 2^{160}. Firstly, we run our code leaving two consecutive 32-bit words W_i free, i.e. 64 bits free. Collected data are shown in 6.4 and they represent an average value of 30 measurements. In particular, this table shows that leaving 64 bits free at the beginning of our input data, for example W_6, W_7, does not have the same effect of leaving them at the end, namely W_{14}, W_{15}. The measured gap is

Table 6.3: Server: Average times (secs) to compute a pre-image, 440 bits free.

polar	restart	n.o.t.	round 20	round 21
rnd	geom	1	96/2.488	4/175.615
		2	100/1.541	7/174.211
		4	100/0.921	23/158.744
		8	100/0.744	31/148.996
	glue	1	69/5.914	2/178.810
		2	100/1.840	10/171.939
		4	100/1.425	11/168.737
		8	100/1.181	5/176.104
	luby	1	90/3.523	15/175.700
		2	100/1.799	11/171.910
		4	100/1.126	12/172.044
		8	100/0.777	27/153.750
true	geom	1	96/2.460	27/163.243
		2	100/1.430	21/167.822
		4	100/1.267	29/161.265
		8	100/0.974	35/151.634
	glue	1	87/4.566	21/170.374
		2	100/1.956	21/168.125
		4	100/1.613	22/168.964
		8	100/1.274	24/168.082
	luby	1	91/3.698	25/165.653
		2	100/1.741	19/169.599
		4	100/1.374	35/153.619
		8	100/0.961	43/152.214

noteworthy. In one case we talk about hours while in the other of hundredths of a second.

Table 6.4: SAT based attack with two consecutive W_i unknown words.

Unknown words W_i	Rounds						
	21	22	23	24	25	26	27
6&7	4.16s						
7&8	0.45s	1h 51m 19.23s					
8&9	1m 17.31s	1m 40.51s					
9&10	1.17s	2m 30.03s	1m 57.31s				
10&11	0.02s	2.40s	2m 31.21s	2m 22.43s			
11&12	0.02s	0.02s	2.10s	13.83s	2m 29.93s		
12&13	0.01s	0.08s	3.75s	50.42s	56.13s	2h 12m 39.69s	
13&14	0.01s	0.02s	0.19s	14.41s	17.33s	6m 43.31s	
14&15	×	0.01s	0.02s	0.33s	13.46s	13.83s	7m 7.29s

Secondly, we adopt a slightly different approach. Due to its structure — we are performing oplus operations on a ring of integers modulo 2, where $W_i = (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \lll 1$, for $i = 16, \ldots, 79$ — it is possible unfold the entire message expansion of SHA-1 and rewrite such expansion as a set of any sixteen consecutive words, which do not have to be necessarily the first

Table 6.5: Computational time (secs) spent to execute a partial SHA-1 pre-image attack.

	equivalent free bits				
rounds	64	80	96	112	128
20	**<0.01**	**<0.01**	**<0.01**	**0.01**	**0.03**
21	**<0.01**	**<0.01**	**0.01**	**0.02**	**5.94**
22	**<0.01**	**<0.01**	**0.22**	**6.84**	**2679.73**
23	**<0.01**	**0.01**	**0.17**	**5.32**	6266.63
24	**0.01**	**0.02**	**13.92**	**550.90**	1766488.38
25	**0.03**	**2.21**	**1727.79**	**106508.62**	
26	**1.37**	**89.17**	172552.81	528294.05	
27	**4.98**	**137.46**			
28	**131.02**	4717.71			
29	**4588.24**	593366.71			
30	9924.96				
31	16255.33				

sixteen. When we try to break round r, 64 bits free, fixing words from W_{r-14} to W_{r-1} is particularly effective.

In fact, guessing the value of the two words W_{r-15} and W_{r-16} we are able to get the 16 words of the input message. Notice that, exploiting the equation of the message schedule we can obtain relations between some input words, e.g. $W_0 \oplus W_2$, also without guessing W_{r-15} and W_{r-16}. Doing so, the computational time spent by CryptoMiniSat to execute a pre-image attack gives better results than those obtained fixing the input message words in other ways (e.g. first 14 words). In addition to 64 bits free, we also executed an attack leaving 80, 96, 112 and 128 bits free. Gathered data are reported in table 6.5 using the bold notation to represent average values on 30 runs. Notice that, due time consuming, the 25th round 112 bits free represents an average value on 16 runs.

Part III

Conclusion

7. Conclusion

In this final chapter, we are going to clarify the common theme of the two previous parts that is the base of this work. As well, we are going to highlight the results obtained. Lastly, we will describe possible future works based on this one.

7.1. Work highlights

In the first part, we have seen how to improve the polynomial multiplication algorithms. Note that it was done both from theorical and practical point of view. In fact, we have shown how to decrease the complexity of the polynomial multiplication algorithms and how to obtain new algorithms which are improved both in terms of gates and in terms of depth respect to the ones currently in the state of art. Remembering what an SLP is (refer to the end of chapter 1), we can say that we have found new ways of representing the same algorithm. In particular, techniques developed in this work are more efficient than the existing ones [8, 19]. Moreover, we have proved through a real case application that one of the new polynomial multiplication algorithms can be sucessfully used to further improve the efficiency of the implementation of the McEliece cryptosystem submitted to the NIST for the standardization of Post-Quantum Cryptography.

Secondly, we have seen that a good and optimized CNF, can decrease the computational effort needed to find a solution to a satisfiability problem. This method, beside other assumptions, was used to recover a round reduced SHA-1 first pre-image outperforming in some cases the current state of art [3, 17, 33].

Summing up, we can say that, in this work, we achieved better representations for cryptographic primitives which is the common theme of the two parts. In particular, this is crucial when analyzing the strenght of cryptographic systems, in fact, we can speed up algorithms such as the McEliece cryptosystem with optimized polynomial multiplication or even measure the real strength of an adversary when trying to break hash functions.

7.2. Future works

Regarding the first part, i.e. the polynomial multiplication algorithm, we need to explore new algorithms. Even if many records were obtained and it is straight to check that the minimum is obtained for low degrees, it is still not clear how to reach the minimum number of operations nor the minimum depth for higher

degrees. Although there are new promising techniques such as the *Karatsuba-like* that could be used in general, it has been shown [35] that the old *School-book* algorithm is still efficient in some cases and more research needs to be conducted. Moreover, the new Post-Quantum Cryptography offers many real case implementation in which this improvements can be employed to obtain the highest speed possible.

Regarding the second part, i.e. the first pre-image attack to SHA-1, new representations have to be explored. Satisfiability can be seen as an efficient constraint solver, but its power needs to be exploited with good representation of the problem. Therefore, possible future works include exploring new way of representing hash functions and the usage of different solvers.

Bibliography

[1] Adámek Jiří. "Foundations of coding. A Wiley-Interscience Publication". In: *New York* (1991).

[2] Albrecht Martin R. "The M4RIE library for dense linear algebra over small fields with even characteristic". In: *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*. 2012, pp. 28–34.

[3] Aoki Kazumaro and Sasaki Yu. "Meet-in-the-middle preimage attacks against reduced SHA-0 and SHA-1". In: *Annual International Cryptology Conference*. Springer. 2009, pp. 70–89.

[4] Aragon Nicolas, Lavauzelle Julien, and Lequesne Matthieu. *decodingchallenge.org*. http://decodingchallenge.org, last accessed March 2022.

[5] Audemard Gilles and Simon Laurent. "Predicting learnt clauses quality in modern SAT solvers". In: *Twenty-first international joint conference on artificial intelligence*. Citeseer. 2009.

[6] Bellare Mihir, Canetti Ran, and Krawczyk Hugo. "Keying hash functions for message authentication". In: *Annual international cryptology conference*. Springer. 1996, pp. 1–15.

[7] Bellini Emanuele et al. "New Records of Pre-image Search of Reduced SHA-1 Using SAT Solvers". In: *Proceedings of the Seventh International Conference on Mathematics and Computing*. Springer. 2022, pp. 141–151.

[8] Bernstein Daniel J. "Batch binary edwards". In: *Annual International Cryptology Conference*. Springer. 2009, pp. 317–336.

[9] Bernstein Daniel J. "Fast multiplication and its applications". In: *Algorithmic number theory* 44 (2008), pp. 325–384.

[10] Bernstein Daniel J et al. "Classic McEliece: conservative code-based cryptography". In: *NIST submissions* (2017).

[11] Biere Armin, Heule Marijn, and Maaren Hans van. *Handbook of satisfiability*. Vol. 185. IOS press, 2009.

[12] Boyar Joan, Matthews Philip, and Peralta René. "Logic minimization techniques with applications to cryptology". In: *Journal of Cryptology* 26.2 (2013), pp. 280–312.

[13] Boyar Joan and Peralta René. "A new combinational logic minimization technique with applications to cryptology". In: *International Symposium on Experimental Algorithms*. Springer. 2010, pp. 178–189.

[14] Boyar Joan, Peralta René, et al. "Small low-depth circuits for cryptographic applications". In: *Cryptography and Communications* 11.1 (2019), pp. 109–127.

[15] Brier Eric, Clavier Christophe, and Olivier Francis. "Correlation power analysis with a leakage model". In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2004, pp. 16–29.

[16] Buchberger Bruno. "Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal". In: *PhD thesis, Universitat Insbruck* (1965).

[17] Cannière Christophe De and Rechberger Christian. "Preimages for Reduced SHA-0 and SHA-1". In: *Annual International Cryptology Conference*. Springer. 2008, pp. 179–202.

[18] Cano-Quiveu German et al. "Embedded LUKS (E-LUKS): A Hardware Solution to IoT Security". In: *Electronics* 10.23 (2021), p. 3036.

[19] Cenk Murat and Hasan M Anwar. "Some new results on binary polynomial multiplication". In: *Journal of Cryptographic Engineering* 5.4 (2015), pp. 289–303.

[20] Cenk Murat, Hasan Mohammed Anwar, and Negre Christophe. "Efficient subquadratic space complexity binary polynomial multipliers based on block recombination". In: *IEEE Transactions on Computers* 63.9 (2013), pp. 2273–2287.

[21] Cenk Murat, Negre Christophe, and Hasan M Anwar. "Improved three-way split formulas for binary polynomial multiplication". In: *International Workshop on Selected Areas in Cryptography*. Springer. 2011, pp. 384–398.

[22] Ceria Michela et al. "Optimizing the Key-Pair Generation Phase of McEliece Cryptosystem". In: *4th International Conference on Wireless, Intelligent and Distributed Environment for Communication*. Springer. 2022, pp. 111–122.

[23] Cimatti Alessandro et al. "The mathsat5 smt solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2013, pp. 93–107.

[24] Cook Stephen A. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing*. 1971, pp. 151–158.

[25] Damgård Ivan Bjerre. "A design principle for hash functions". In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 416–427.

[26] Dang Quynh, Blank Rebecca M, Barker Comments From Elaine, et al. "NIST Special Publication 800-107 Revision 1 Recommendation for Applications Using Approved Hash Algorithms". In: (2012).

[27] Davis Martin, Logemann George, and Loveland Donald. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[28] Davis Martin and Putnam Hilary. "A computing procedure for quantification theory". In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215.

[29] De Piccoli Alessandro, Visconti Andrea, and Rizzo Ottavio Giulio. "Polynomial multiplication over binary finite fields: new upper bounds". In: *Journal of Cryptographic Engineering* 10.3 (2020), pp. 197–210.

[30] Dutertre Bruno. "Yices 2.2". In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 737–744.

[31] Eastlake 3rd D and Jones Paul. *Rfc3174: Us secure hash algorithm 1 (sha1)*. 2001.

[32] Eén Niklas and Sörensson Niklas. "An extensible SAT-solver". In: *International conference on theory and applications of satisfiability testing*. Springer. 2003, pp. 502–518.

[33] Espitau Thomas, Fouque Pierre-Alain, and Karpman Pierre. "Higher-order differential meet-in-the-middle preimage attacks on SHA-1 and BLAKE". In: *Annual Cryptology Conference*. Springer. 2015, pp. 683–701.

[34] Faugere Jean-Charles. "A new efficient algorithm for computing Gröbner bases (F4)". In: *Journal of pure and applied algebra* 139.1-3 (1999), pp. 61–88.

[35] Find Magnus Gaudal and Peralta René. "Better circuits for binary polynomial multiplication". In: *IEEE Transactions on Computers* 68.4 (2018), pp. 624–630.

[36] Gomes Carla P, Selman Bart, Kautz Henry, et al. "Boosting combinatorial search through randomization". In: *AAAI/IAAI* 98 (1998), pp. 431–437.

[37] Gueron Shay and Kounavis Michael E. "Intel® carry-less multiplication instruction and its usage for computing the GCM mode". In: *White Paper* (2010), p. 10.

[38] Hoffstein Jeffrey, Pipher Jill, and Silverman Joseph H. "NTRU: A ring-based public key cryptosystem". In: *International algorithmic number theory symposium*. Springer. 1998, pp. 267–288.

[39] Iuorio Andrea Francesco and Visconti Andrea. "Understanding optimizations and measuring performances of PBKDF2". In: *International Conference on Wireless Intelligent and Distributed Environment for Communication*. Springer. 2018, pp. 101–114.

[40] Janhunen Tomi. "Intermediate languages of ASP systems and tools". In: *de Vos and Schaub [16]* (2007), pp. 12–25.

[41] Jankowski Krzysztof, Laurent Pierre, and O'Mahony Aidan. "Intel polynomial multiplication instruction and its usage for elliptic curve cryptography". In: *White paper, April* (2012).

[42] Jeannerod Claude-Pierre, Pernet Clément, and Storjohann Arne. "Rank-profile revealing Gaussian elimination and the CUP matrix decomposition". In: *Journal of Symbolic Computation* 56 (2013), pp. 46–68.

[43] Karatsuba Anatolij A. and Ofman Yu. "Multiplication of Multidigit Numbers on Automata". In: *Soviet physics. Doklady* 7 (1963), pp. 595–596.

[44] Kelsey John and Schneier Bruce. "Second preimages on n-bit hash functions for much less than 2 n work". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2005, pp. 474–490.

[45] Kocher Paul, Jaffe Joshua, and Jun Benjamin. "Differential power analysis". In: *Advances in Cryptology — CRYPTO' 99*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397.

[46] Laboratories RSA. *RSA Factoring Challenge*. http://tiny.cc/osbbtz, last accessed March 2022.

[47] Le Berre Daniel and Parrain Anne. "On SAT technologies for dependency management and beyond". In: *First Workshop on Software Product Lines (ASPL'08)*. 2008, pp. 197–200.

[48] Legendre Florian, Dequen Gilles, and Krajecki Michaël. "Encoding hash functions as a sat problem". In: *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*. Vol. 1. IEEE. 2012, pp. 916–921.

[49] Legendre Florian, Dequen Gilles, and Krajecki Michaël. "Logical reasoning to detect weaknesses about SHA-1 and MD4/5". In: *Cryptology ePrint Archive* (2014).

[50] Leurent Gaëtan and Peyrin Thomas. "From collisions to chosen-prefix collisions application to full SHA-1". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2019, pp. 527–555.

[51] Luby Michael, Sinclair Alistair, and Zuckerman David. "Optimal speedup of Las Vegas algorithms". In: *Information Processing Letters* 47.4 (1993), pp. 173–180.

[52] Ma Lijie et al. "NIST bullet signature measurement system for RM (Reference Material) 8240 standard bullets". In: *Journal of Forensic Science* 49.4 (2004), pp. 1–11.

[53] Makarim Rusydi H and Stevens Marc. "M4GB: an efficient Gröbner-basis algorithm". In: *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*. 2017, pp. 293–300.

[54] Marques-Silva Joao P and Sakallah Karem A. "GRASP: A search algorithm for propositional satisfiability". In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521.

[55] Mavroeidis Vasileios et al. "The impact of quantum computing on present cryptography". In: *arXiv preprint arXiv:1804.00200* (2018).

[56] Maximov Alexander and Ekdahl Patrik. "New circuit minimization techniques for smaller and faster AES SBoxes". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), pp. 91–125.

[57] McEliece Robert J. "A public-key cryptosystem based on algebraic". In: *Coding Thv* 4244 (1978), pp. 114–116.

[58] Merkle Ralph C. "A certified digital signature". In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 218–238.

[59] Merkle Ralph Charles. *Secrecy, authentication, and public key systems*. Stanford university, 1979.

[60] Milne James S. *Fields and Galois theory*. 2020.

[61] Nejati Saeed et al. "Adaptive restart and CEGAR-based solver for inverting cryptographic hash functions". In: *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer. 2017, pp. 120–131.

[62] NIST. "FIPS PUB 180–4. Secure Hash Standard (SHS)". In: (2012).

[63] Nossum Vegard. "SAT-based preimage attacks on SHA-1". MA thesis. University of Oslo, 2012.

[64] Orellana Rosa. *Course notes in discrete mathematics in computer science*. https://math.dartmouth.edu/archive/m19w03/public_html/book.html, last accessed March 2022.

[65] Paar Christof and Rosner Martin. "Comparison of arithmetic architectures for Reed-Solomon decoders in reconfigurable hardware". In: *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No. 97TB100186)*. IEEE. 1997, pp. 219–225.

[66] Perlner Ray A and Cooper David A. "Quantum resistant public key cryptography: a survey". In: *Proceedings of the 8th Symposium on Identity and Trust on the Internet*. 2009, pp. 85–93.

[67] Reyhani-Masoleh Arash, Taha Mostafa, and Ashmawy Doaa. "Smashing the implementation records of AES S-box". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), pp. 298–336.

[68] Rotman Joseph J. *An introduction to the theory of groups*. Vol. 148. Springer Science & Business Media, 2012.

[69] Ryvchin Vadim and Strichman Ofer. "Local restarts in SAT". In: *Constraint Programming Letters (CPL)* 4 (2008), pp. 3–13.

[70] Schöning U. and Torán J. *The Satisfiability Problem: Algorithms and Analyses*. Mathematik für Anwendungen. Lehmanns Media, 2013. ISBN: 9783865416483.

[71] Sendrier Nicolas. "Encoding information into constant weight words". In: *Proceedings. International Symposium on Information Theory, 2005. ISIT 2005.* IEEE. 2005, pp. 435–438.

[72] *Shanatomy*. https://github.com/Crypto-TII/shanatomy, last accessed March 2022.

[73] Soos Mate, Nohl Karsten, and Castelluccia Claude. "Extending SAT solvers to cryptographic problems". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2009, pp. 244–257.

[74] Stallings William. *Cryptography and Network Security: Principles and Practice*. 5th. USA: Prentice Hall Press, 2010. ISBN: 0136097049.

[75] Steube Jens. "Optimising computation of hash-algorithms as an attacker". In: *Passwords (Las Vegas, 2013)* (2016). http://hashcat.net/events/p13/js-ocohaaaa.pdf, last accessed March 2022.

[76] Stevens Marc et al. "The first collision for full SHA-1". In: *Annual international cryptology conference*. Springer. 2017, pp. 570–596.

[77] Sudar Chandramohan, Arjun SK, and Deepthi LR. "Time-based one-time password for Wi-Fi authentication and security". In: *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE. 2017, pp. 1212–1216.

[78] Van Harmelen Frank, Lifschitz Vladimir, and Porter Bruce. *Handbook of knowledge representation*. Elsevier, 2008.

[79] Visconti Andrea and Gorla Federico. “Exploiting an HMAC-SHA-1 optimization to speed up PBKDF2”. In: *IEEE Transactions on Dependable and Secure Computing* 17.4 (2018), pp. 775–781.

[80] Visconti Andrea, Schiavo Chiara Valentina, and Peralta René. “Improved upper bounds for the expected circuit complexity of dense systems of linear equations over GF (2)”. In: *Information processing letters* 137 (2018), pp. 1–5.

[81] Walsh Toby et al. “Search in a small world”. In: *Ijcai*. Vol. 99. Citeseer. 1999, pp. 1172–1177.

[82] Wang Xiaoyun, Yin Yiqun Lisa, and Yu Hongbo. “Finding collisions in the full SHA-1”. In: *Annual international cryptology conference*. Springer. 2005, pp. 17–36.

[83] Zhou Gang and Michalik Harald. “Comments on" A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Field"”. In: *IEEE Transactions on Computers* 59.7 (2010), pp. 1007–1008.