Author:
Bentahar, Kamel

Title:
Theoretical and practical efficiency aspects in cryptography

Theoretical and Practical Efficiency Aspects in Cryptography

Kamel Bentahar

A thesis submitted to the University of Bristol in accordance with the requirements for the Degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

April 9, 2008

Abstract

The issue of efficiency is a crucial factor in the field of Cryptology. Be it a cryptographic protocol, a cryptanalytic attack or a proof of security – they all have to be efficient. This thesis is concerned with this issue in the context of Cryptography, i.e. providing security, as opposed to Cryptanalysis. Two model cases are considered in this thesis:

❶ The first of which can be referred to as *efficient practical realisation of primitives*, where we study two concrete problems:

- **The RSA modular arithmetic at high security parameters.** We deal with the recent recommendation by NIST of increasing key sizes from the currently used range 1,024–4,096 bits to around 15,360 bits. The traditional methods are too slow for these operand sizes, and hence there is a pressing need to explore and develop newer methods. A range of possibilities are suggested and analysed in detail, and the theoretical results are further tested and confirmed with an implementation.
- **LASH: Lattice based hash function.** We investigate an old idea for a secure hash function based on lattices. We show that this latter hash function proposal is *not* secure if instantiated with concrete parameters, despite it being *asymptotically* secure. We propose a practical modification which is efficient, resists the known attacks, yet is not provably secure.

❷ The second case that we will study is *efficiency in theoretical arguments*, where we concern ourselves with the establishment of a tight proof of equivalence of the Discrete Logarithm Problem (DLP) and Diffie-Hellman Problem (DHP) by employing optimised algorithms in the reduction proposed by Maurer. We consequently conclude lower bounds for the computational complexity of the DHP (Assuming the generic exponential complexity of the DLP for elliptic curves).

BLANK IN ORIGINAL

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِيم

وَصَلَّى اللَّهُ وَسَلَّمَ عَلَى نَبِيِّنَا مُحَمَّدٍ وَآلِهِ وَصَحْبِهِ وَمَنْ تَبِعَهُمْ بِإِحْسَانٍ إِلَى يَوم الدِّين

In the name of God (Allah), the most gracious, the most merciful
and may peace and blessings of God be upon the prophet who was sent unto us Muhammad, his household, his companions and all those who follow them in righteousness till the Day of Reckoning (Judgement)

إِلَى وَالِدَيَّ حَفِظَهُمَا اللَّهُ وَبَارَكَ فِيهِمَا

To my parents, may God protect and bless them

إِلَى جَدِّي الْبَشِير وَجَدَّتَيَّ الْيَاقُوت وَالْحَاجَّة وَعَمِّي قَدُّور رَحِمَهُمُ الله

إِلَى جَدِّي مُحَمَّد وَجَدَّتِي مَسْعُودَة وَأَسَاتِذَتِي وَأَقَارِبِي حَفِظَهُمُ الله

To my grandfather "al-bašīr," my grandmothers "al-yāqūt" and "al-ḥāǧǧah" and my uncle "qaddūr," may God have mercy upon them [as they passed away]
To my grandfather "muḥammad" and grandmother "masʿūdah," my teachers and relatives, may God preserve them

إِلَى الْجَزَائِرِ الْحَبِيبَة

To Algeria the beloved

BLANK IN ORIGINAL

Acknowledgements

اَلْحَمْدُ لِلّٰهِ الَّذِي عَلَّمَ بِالْقَلَمِ

عَلَّمَ الْإِنسَانَ مَا لَمْ يَعْلَمْ

Praise be to God, who taught with the pen
[He] taught man that which he did not know

I WOULD LIKE TO THANK *my supervisor Prof. Nigel P. Smart for his guidance, support and encouragement. Without his kindness, patience, humour and infallible dedication and love of the subject I would not have been able to produce this thesis. Nigel, thanks for everything!*

SECONDLY, I THANK *all of my family, starting from my dear parents, who have given me every support during my studies abroad, to my brothers and sisters whom I cherish so much, to all of my friends both back at home and in the UK.*

I WOULD ALSO LIKE TO THANK *all of my friends in the Information Security Group (in alphabetical order): Amoss, Bogdan, Dan, Elisabeth, Fangfang, Johnny, Paul, Philip, Pooya and Rob; and last but not least Fré, John, Manuel, Marcel, Martijn, Peter and Richard who left us before the end of my studies. I am especially grateful to John and Dan as they helped me greatly during my PhD. Special thanks should go to my friends in the Bristol Islamic Society who encouraged me during the writing of this thesis, especially* عبد العَظيم *for proof reading my thesis and* عبد المجيد *for lending me his laptop.*

I AM GRATEFUL *to Prof. Steven Galbraith and Dr. Henk Muller for their kindness and constructive remarks which have improved the quality of this thesis greatly.*

FINALLY, I THANK *the Algerian government for giving me this opportunity of pursuing my studies in the UK. In particular, I thank the Algerian consulate in London who have been understanding and helpful throughout.*

Paginated blank pages are scanned as found in original thesis

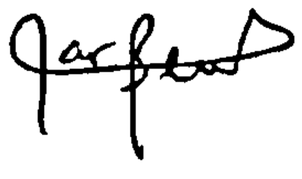
No information is missing

Declaration

I declare that the work in this dissertation was carried out in accordance with the Regulations of the University of Bristol. The work is original except where indicated by special reference in the text and no part of the dissertation has been submitted for any other degree.

Any views expressed in the dissertation are those of the author and in no way represent those of the University of Bristol.

The dissertation has not been presented to any other University for examination either in the United Kingdom or overseas.

SIGNED:  DATE: 11/04/2008
Bristol

BLANK IN ORIGINAL

Contents

CONTENTS

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

"الحيط بوذنيه *al-ḥayṭ bwadhnīh*"

— *"The wall has ears."* Algerian saying referring to eavesdroppers.

Mainly used to protect state secrets for the past few thousand years, cryptography has always been regarded as the obscure and crafty twin of communication. This view dramatically changed at the end of the last century with the advent of digital computers and large-scale open networks. This shift in technology and change in scope has allowed cryptography to gain extensive grounds and, in fact, it is now part of most people's daily life – from businessmen managing their business to the citizens using their computers, bank cards and their mobile phones or digital media players.

Historically, cryptography has been more of a black art than a science, with which one can grant friends access to private and confidential information while denying it to foes. Over the many centuries of its development, cryptography gradually became the study of mathematical techniques that may aid in achieving a number of *information security* goals which stem from actual needs and experience. These include *confidentiality* and *authentication* as the prime goals, because the former insures that data is intelligible to the desired parties *only* and thus guarantees *secrecy*. The latter goal, which includes *data integrity*, prevents anyone from pretending to be a legitimate party, and allows for the detection of any modification, insertion or deletion of messages while transported.

There is however another type of need which is less abstract but has always been a main concern in any design of cryptosystems or any reasoning about security, and that is *efficiency*. For the early civilisations, it was very important for the scribes to be able to encrypt and decrypt messages in reasonable time – either mentally or using simple tools available at the time. With the advancement of technology and science, better devices were built and more complex techniques were devised and deployed, as the older ones got discovered and broken. The persistent search for perfect cryptography has kept great minds from many civilisation busy trying to invent simple yet secure techniques, and even today we are still pursuing this dream taking speed and storage requirements as the second most important design criteria after security.

In this thesis, we focus on the issue of providing cryptographic products in a form that allows them to be used efficiently, especially with respect to execution time. Products here should not only be taken to mean commercial products but also intellectual products that have earned academic merit. We study a few sample cases that stretch from implementation to design of cryptographic primitives to theoretically arguing about the complexity of computational cryptographic problems, and hence their level of security.

This introductory chapter will review a few background topics of general interest to cryptography. We will then spend two chapters preparing specific material needed for the work presented in the subsequent chapters. The next three chapters will contain the actual contribution of this thesis, summarising the results of the work done in the course of my PhD research. This then naturally leads us to the final conclusion and open research topics chapter. A more detailed overview of the structure of this thesis is given at the end of this chapter on page 35.

We will start by giving a quick sketch of the historical development of cryptology. There are certainly missing pieces from the jigsaw as knowledge about ciphers used to be kept secret and only known in limited circles as part of making the whole system harder to crack – hence the difficulty of tracing the exact course of progress.

1.1 Brief history

A good reference and a joyful book on the history of cryptology is David Kahn's *The Code Codebreakers* [Kah67]. For modern cryptography, one may consult the *Handbook of Applied Cryptography* [MOV97], *Applied Cryptography* [Sch96] or *Mathematics for Cryptography* [Pin97].

The historical development of cryptology has been shaped by numerous factors, but the military is undoubtedly the main one. The other factors are not much less important however. Language morphology and writing systems, for example, have played a prime role in pre 20th century cryptologic techniques. Furthermore, the cryptographic techniques that can be used at any point in time depend directly on the authoring and communication technological means available, and that is why whenever there is a breakthrough in these aspects there is usually another in cryptology.

Cryptography, as a tool, is very old and dates back to as long ago as humanity can remember. Auguste Kerckhoff comments on this in his historic paper *La Cryptographie Militaire* [Ker83] by saying

> La Cryptographie ou l'Art de chiffrer est une science vieille comme le monde ; confondue à son origine avec la télégraphie militaire, elle a été cultivée, dès la plus haute antiquité, par les Chinois, les Perses, les Carthaginois ; elle a été enseignée dans les écoles tactiques de la Grèce, et tenue en haute estime par les plus illustres généraux romains.

With regards to the earliest known methods of encryption, *blinding* or *obscuring* information to counter eavesdroppers and keep it private, has been in common use since, at least, the time of the ancient Egyptians. The methods used were mainly variations of script, transposition of characters or mono-alphabetic substitution combined with clever alterations and shorthands, sometimes with the use of special ink (Steganography). These were the prevalent techniques from about 1900 B.C., and likely even before, to the eighth century; on the Nile river banks, in China, India, Mesopotamia (Persia), Carthage, Greece and Rome. The Hebrews, for example, had standard shift substitution ciphers which they called אַתְבַּשׁ *atbaš*, אָלְבָּם *albam* and אַתְבָּה *atbah*, the Spartans (~475 B.C.) used a device called a *skytale* for a

transposition cipher, *Artha-śastra* (321~300 B.C.) from India wrote *Kantalya* where he speaks about what can be regarded as a substitution cipher, *Herodotus* mentions some steganographic techniques in his *Histories* and the greek *Aeneas the Tactician* wrote the first known text in history on the topic of information security titled *On the defence of fortified places.*

Essentially, the early techniques achieved *security through obscurity*, i.e. by making it hard for the eavesdropper to know how the message was enciphered and hence, as was believed at the time, making it practically impossible to guess the used method or the original message. For example, the Romans used the *Caesar Cipher* which amounts to merely shifting the letters of the alphabet by three positions! One would have thought that these ancient civilisations must have had some interest in developing better methods but, as far as we know, none of them did. The methods known then seemed good enough because of the lack of literacy and also, as it seems, because no-one knew how to break them or no one tried. Furthermore, the fact that encryption and decryption methods were kept secret helped.

The eighth century saw the beginning of the Muslim era, with which came a fresh interest in spreading literacy and developing all kinds of disciplines. Amidst the contributions made during this era are works on the foundation of Cryptology laid out through a systematic study and classification of the different types of ciphers known then, and also through studying and devising generic techniques to cryptanalyse them, [Kah67, p. 71-93]. As a manifestation of this period's influence, the word *cipher*, for example, derives from the Arabic word for zero: صِفْر *ṣifr*, [Al-92]. One factor that encouraged the development of Cryptanalysis in this age was the intense activity of book translation, where they sometimes had to decipher encoded books from previous civilisations that tried to keep its knowledge secret by enciphering its books (e.g. Alchemy books), [MMA97]. Prominent cryptographers from this era include اَلخَلِيل بنُ أَحْمَد الفَرَاهِيدِي *al-ḫalīl bnu ʾaḥmad al-farāhīdī*, اِبنُ كَيْسَان *ibnu kaysān*, اِبنُ وَحْشِيَّة النَّبطِي *ibnu waḥšiyyah al-nnbṭī*, أَبُو حَاتِم السِّجِسْتَانِي *ʾabū ḥātim as-siǧistānī*, but the one who excelled and is awarded the title "Father of Cryptology" is يَعْقُوب الْكِنْدِي *yaʿqūb al-kindī* (c. 801–873) for his work on cryptanalysis:

رِسَالَةٌ فِي اسْتِخْرَاجِ الْمُعَمَّى *risālatun fī 'stiḫrāği 'l-muʿammā*" (Literally: A discourse on the extraction of the blinded) where he introduced, among other techniques, several statistical analysis techniques for cryptanalysis.

This interest then faded away for a while until the Mongols' attacks and the Crusades in the 13th and 14th century, when cryptographers were needed again. Figures of this second period include إِبْنُ دُنَيْنِير *ibnu dunaynīr*, إِبْنُ عَدْلَان *ibnu ʿadlān*, إِبْنُ الدُّرَيْهِم *ibnu 'd-durayhim*, [MMA97]. Development then attenuated with the decline of the Arabs but, fortunately, their effort was not wasted and was rediscovered in Europe at the end of the 15th and beginning of the 16th century during the European Renaissance.

With this rise of Europe, some new and better ciphers were developed and more cryptanalytic techniques were made popular. Cryptography then started enjoying a more mathematical treatment and gradually became demystified. The most influential cryptographer of this period was the Italian Leone Battista Alberti (1404–1472), who earned the title "Father of Cryptology in the West." He wrote *De componendis cifris* and created a poly-alphabetic cipher which is now called after him (Alberti Cipher). Other talented cryptographers of the time were: Johannes Trithemius in Germany (*Steganographia*, 1499 Pub. 1606), Giovan Batista Belaso (*La cifra del. Sig.*, 1553), Giambattista della Porta in Italy (*De Furtivis Literarum Notis*, 1563), Girolamo Cardano (Cardan grille, 1550) and Blaise de Vigenère (*Traicté de Chiffres*, 1585), to whom we mis-attribute the poly-alphabetic substitution known as the Vigenère cipher originally described by Belaso (1553) .

Cryptography had to wait until the 20th century when it played an indispensable role in the two World Wars. It has since enjoyed considerable growth and has become a powerful and profound theoretical and applied discipline. Its development still continues to our day and at a very fast pace, both theoretically and technologically to meet the practical needs and the ever increasing challenges. One new technical advancement in the period of the wars that is worthy of notice is the *rotor machine* – an electro-mechanical device which mainly consists of a set of rotating disks, called rotors. The rotors bear some electrical contacts allowing each of them to act like a substitution cipher. When operating, these rotors may

advance positions after each encryption of a new symbol. The whole set of rotors is configured such that the resulting poly-alphabetic substitution is of the largest period possible. The historical German Enigma and Japanese PURPLE machines were based on this technique; and while these machines are no longer good enough for today's security requirements, the principle behind them still lives in the design of block (and stream) ciphers.

On the foundational side of cryptography, Claude Shannon studied and wrote about the mathematical theory of secrecy in the early 40s but his work was kept secret and was not published until the end of the second world war [Sha48, Sha01]. His paper *A mathematical theory of communication* [Sha48] introduced the concept of *perfect secrecy* and gave a useful measure of the amount of information contained in a message. Shannon then showed that the amount of information that can be perfectly secured is no more than that present in the key. That is to say that the ideal encryption method is to encrypt the message with a truly random key containing at least as much entropy as the message to be encrypted. This method is nowadays commonly known as the *one-time pad*, a method due to Gilbert Vernam (1917). The encryption operation in this case is usually bitwise XOR of the message and the key.

Numerous ciphers were invented and deployed in the last century but very few survived the attacks of the many skilled cryptanalysts. Among these ciphers are the widely used successors of rotor machines: *block ciphers*, which are mainly a result of Horst Feistel's work at IBM in the early 70s. Well known examples of these are the infamous *Data Encryption Standard* (DES) and its successor the *Advanced Encryption Standard* (AES). With these advancements in encryption techniques, other advanced cryptanalytic techniques were developed too. In particular, the well celebrated discovery of *differential cryptanalysis* had a great impact on block ciphers and still plays a very important role in symmetric cryptography.

The major milestone and turning point in the history of Cryptology as a whole is the invention of Public-Key Cryptography. It all started with Diffie and Hellman's paper *New Directions in Cryptography* [DH76], published in 1976, which motivated the interplay between the theories of communication and computation, and addressed "the need for new types of cryptographic systems, which minimize the

need for secure key distribution channels and supply the equivalent of a written signature," [DH76]. This idea was a long awaited feat that changed how we study cryptography and a genuine fulfilment of Kerckhoff's principle, which states that the security of an encryption scheme should only depend on the secrecy of the key, not on the description of encryption and decryption algorithms.

> 1. Le système doit être matériellement, sinon mathématiquement, indéchiffrable; 2. Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi; 3. La clef doit pouvoir en être communiquée et retenue sans le secours de notes écrites, et être changée ou modifiée au gré des correspondants ; . . . [Ker83].

In 1978, Rivest, Shamir and Adleman introduced the first practically feasible public-key cryptosystem called RSA in their paper *A method for obtaining digital signature and public-key cryptosystems* [ARS78]. The RSA algorithm was suggested since it is based on the hardness of factoring large numbers assumption. ElGamal introduced another practical public-key cryptosystem, in 1985, based on the belief that computing discrete logarithms (sometimes also called *indices*) in finite fields is difficult. Development in the whole field of Cryptology has since been very rapid and many schemes were later suggested and either were broken, inefficient or successfully stood up to the test of time and became adopted in standards.

A relatively recent theoretical development is *Provable Security*, where one tries to argue that a probabilistic public-key scheme is immune to attacks from computationally-bounded adversaries. The proofs are almost all reductionist, in the sense that the results are of the type: If the adversary can *systematically* break the scheme, then an efficient algorithm to solve a *related hard problem* exists and can be constructed by using the adversary as an oracle. The hard problems are either computational or decisional problems that have withstood researchers' attacks, and have consequently come to be believed intractable (by polynomial time algorithms). Most of these problems come from the field of Computational Number Theory. The proof techniques themselves have undergone incredible development and an inexhaustible amount of new concepts and results have been introduced and thoroughly studied in the last decade and a half.

There were many other theoretical discoveries and developments during the last few decades, and the subject is still rapidly growing and maturing to become a more established science. A compendium of progress in the foundations of cryptography is Goldreich's two volume book *Foundations of Cryptography* [Gol04a, Gol04b], but the fact remains that most of the recent achievements are still in the research papers and has not been collected in book form yet.

On a warning note, one must note that there are so many issues to take account of after seeing the "proof of security," as it should be interpreted properly and not be given more value than it merits. Furthermore, there are many more practical issues that arise when a system is implemented in practice. For example, since current computational devices leak information about their inputs and intermediate stages then exploiting these leaks has lead to new attacks mainly based on the power consumption of cryptographic devices or their electromagnetic emission, in cases. This kind of attacks are referred to as *Side Channel Attacks* (SCA).

1.2 Mathematical preliminaries

Now we shall start reviewing some useful material to make the thesis self contained. We start here with some mathematical background then recall some notions from Complexity Theory in the next section and finish with some general concepts from cryptography.

As usual, we denote by $\mathbb{N}$ the set of natural numbers $\{1,2,3,\ldots\}$, and by $\mathbb{Z}$ the set of integers $\{\ldots,-2,-1,0,1,2,3,\ldots\}$ and by $\mathbb{R}$ the set of real numbers. The *greatest common divisor* of two integers x and y, denoted by $\gcd(x,y)$, is the largest number that divides both of them; and if this divisor is equal to 1 then we say that they are *coprime*. A natural number $n \neq 1$ is *prime* if all of its positive divisors are trivial, namely 1 and n itself. (Equivalently, the number of its distinct positive divisors is exactly 2. The first few prime numbers are $2,3,5,7,11,\ldots$). If a non-zero integer is not a prime nor a unit (± 1) then it is called *composite*, and it factors uniquely into a product of prime powers up to multiplication by units and reordering of the prime factors (The Fundamental Theorem of Arithmetic).

1.2.1 Abstract algebra

The structures known as group, rings and fields are of prime importance in modern cryptography, and thus we will review them next. The second half of Chapter 2 is dedicated to (generic) algorithms in groups that we will use in subsequent chapters.

Groups

Recall that a *monoid* is a non-empty set G that is closed under an *associative* binary operation on the elements of G and is such that there exists a *neutral element*, which when multiplied with any elements yields the same element back. If every element of G also has an *inverse*, such that when multiplied together yield the neutral element, then G is called a *group*. If, furthermore, the operation is commutative then the group is said to be a *commutative group* or *Abelian group*.

In the case where the set G is finite then we denote the number of its elements by $|G|$ or $\#G$ and refer to it as its *order*; the group is then called a *finite group*. A group is *cyclic* in the special case where all the elements of G are powers of a fixed element $g \in G$, called a *generator* of G. We then write $G = \langle g \rangle$ and say that G is generated by g. If a subset $H \subset G$ contains the neutral element of G and is closed under the same operation and inversion then it is a group too and is called a *subgroup* of G.

Several familiar group structures are met in practice among which are the additive group $\mathbb{Z}/m\mathbb{Z}$ of size m, for an integer $m \in \mathbb{Z}_{\neq 0,\pm 1}$, and the multiplicative group $(\mathbb{Z}/m\mathbb{Z})^\times$ of size $\varphi(m)$. Also, of special interest are the elliptic-curve groups over finite fields (described in 1.2.2).

In most cryptographic applications, finite cyclic Abelian subgroups with an *efficiently computable* representation and group operation are used. For security considerations, the group order is usually either prime or has a small cofactor. Such groups are used because they usually come with a computationally hard problem, such as the the so called Discrete Logarithm Problem (DLP), described in subsection 1.4.1, which can be used in designing practical provably secure cryptosystems.

Rings and (Finite) Fields

Let R be a nonempty set with two binary operations: $+$ and $\times$. We say that R is a *ring* if it is an Abelian group with respect to $+$ and a monoid with respect to $\times$. If the $\times$ operation is commutative then we call R a *commutative ring*. If every element in R has a multiplicative inverse then we call it a *field*.

In cryptography, we are mainly interested in *finite* fields. These are denoted by $\mathbb{F}_q$ (or GF(q) for Galois Field) where q is either a prime p or a prime power $q = p^e$. Furthermore, these are the only possibilities up to isomorphism. The prime p is called the field *characteristic*. If $p = 2$ we call $\mathbb{F}_q = \mathbb{F}_{2^m}$ a *binary field* of *extension degree* m, and if $p > 2$ then we call $\mathbb{F}_p$ a *prime field*. We denote by $\bar{\mathbb{F}}_q$ the algebraic completion of a finite field $\mathbb{F}_q$.

1.2.2 Elliptic curves

Elliptic curves were first proposed for use in cryptography by Neal Koblitz [Kob87] and Victor Miller [Mil86], independently. This gave birth to *Elliptic-Curve Cryptography* (ECC), which enjoyed rapid growth and huge popularity in the following decades. ECC is now accepted as the efficient alternative to RSA and finite field discrete-logarithm-problem based schemes. Good reference books on ECC are [HMV03] and [CFA+06].

In this thesis, and in cryptography in general, we are only interested in the case of elliptic curves over a finite field K. In the case when $K = \mathbb{F}_q$ is a finite field of characteristic greater than 3, an elliptic curve E over $\mathbb{F}_q$ is the set of points $(x, y) \in \bar{\mathbb{F}}_q \times \bar{\mathbb{F}}_q$ satisfying the Weierstrass equation

$$y^2 = x^3 + ax + b, \quad \text{for some } a, b \in \mathbb{F}_q \text{ satisfying } 4a^3 + 27b^2 \neq 0 \tag{1.1}$$

together with the *point at infinity* which we denote by O.

In the case of a binary field $K = \mathbb{F}_{2^m}$, the curve equation takes one of the following two forms

$$y^2 + xy = x^3 + ax^2 + b \quad \text{for some } a, b \in \mathbb{F}_{2^m} \tag{1.2}$$

or

$$y^2 + cy = x^3 + ax + b \quad \text{for some } a, b, c \in \mathbb{F}_{2^m} \text{ and } c \neq 0. \tag{1.3}$$

The first equation gives rise to a non-supersingular curve while the second yields a supersingular curve.

Group Structure of elliptic curves

An elliptic curve E has a structure of an Abelian group with the point at infinity O as its identity element. The addition operation is given by the *line and chord* rule, as illustrated in Figure 1.1. The exact formulae for addition are expressible algebraically as rational functions in the points' coordinates.

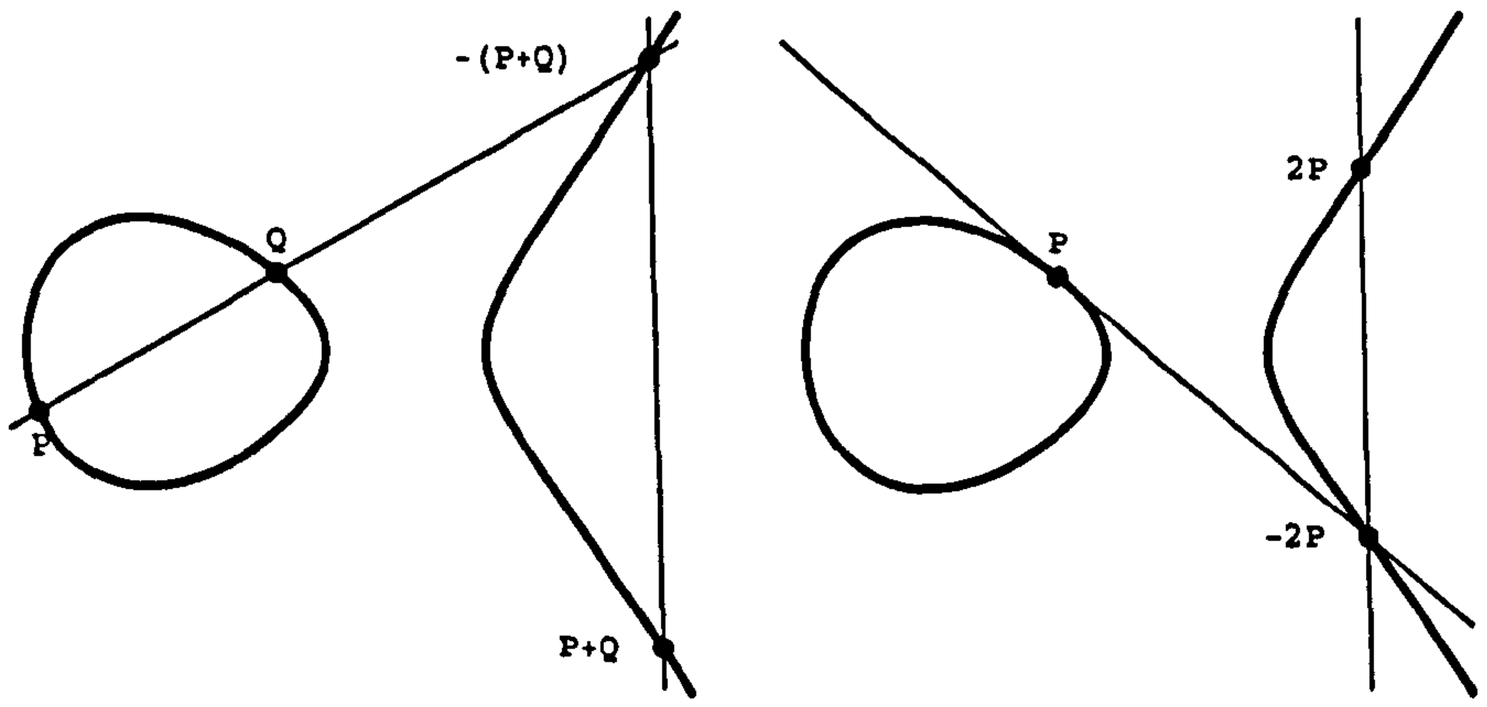


Figure 1.1: Adding and doubling points on an elliptic curve over $\mathbb{R}$.

Let K have characteristic greater than 3, and let $P_1 = (x_1, y_1), P_2 = (x_2, y_2) \in E$. Then, $-P_1 = (x, -y_1)$ and if $P_2 \neq -P_1$ (otherwise $P_1 + P_2 = O$) then $P_1 + P_2 = (x_3, y_3)$ where

$$\begin{cases} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \end{cases}, \tag{1.4}$$

where λ is given by

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P_1 \neq P_2 \\ \frac{3x_1^2 + a}{2y_1} & \text{if } P_1 = P_2. \end{cases}$$

Similar formulae can be derived for binary fields but we will not present them here because we will not need them in this thesis. The interested reader may find them in [BSS99].

Note that the above given description of elliptic curves and the addition formulae are all given using the *affine coordinate system*. Some other coordinate systems and relevant algorithms will be discussed in Chapter 2.

The set of points on E having coordinates in K itself (i.e. not in $\bar{K} - K$) together with the point O is denoted by $E(K)$ and is known as the set of *K-rational points*. The set $E(\mathbb{F}_q)$ with the previously defined addition has an Abelian group structure of rank 1 or 2, and we have $E(\mathbb{F}_q) \cong C_{n_1} \oplus C_{n_2}$ where $n_2 | n_1$, $n_2 | q-1$ and C_n is the cyclic group of order n, [Sil86].

The size of $E(\mathbb{F}_q)$ is given by the *Hasse theorem* as

$$\#E(\mathbb{F}_q) = q + 1 - t, \tag{1.5}$$

where t is the value of the *Fröbenius trace* and is bounded by $|t| \leq 2\sqrt{q}$.

Waterhouse [Wat69] showed that if q is prime then there exists at least one elliptic curve for each possible trace value $|t| \leq 2\sqrt{q}$. For the case where $q = 2^m$, this only holds for the odd values of t. Furthermore, Lenstra showed that the distribution of the orders is almost uniform for $t \leq \sqrt{q}$, [LJ87], which is in accordance with the Sato-Tate conjecture [Sil92, p. 120] (or alternatively see [CFA+06, p. 605]).

1.3 Complexity theory

Complexity Theory studies the cost of computation as a function of the length of the input. The measured cost is mainly the computation time or the storage space, but could also be some other computational resource. The lower the cost is, the more efficient the computation is regarded. The theory of complexity is very rich and vast, but we will concisely summarise the relevant notions and results here. A more in-depth introduction to subject can be found in [Sip05, Pap94].

The usual cost analyses consider either the *average case complexity* or the *worst case complexity*, where the latter produces an absolute upper-bound on the machine's running time while the former estimates its running time on a random instance selected uniformly at random from the set of instances (or according to some other

relevant distribution). In cryptography, the average case complexity[1] tends to be more important as it guarantees a certain level of difficulty for the practically deployed instances.

The set $\Sigma = \{0,1\}$ is called the set of binary alphabet. A binary *string* is a sequence of bits (symbols from Σ). The set $\Sigma^n = \{0,1\}^n$ is the set of binary strings of *length* exactly n bits, where the length or *size* of a bit string is the number of bits in it. The set of all binary strings (arbitrary but finite length) is denoted by $\Sigma^* = \{0,1\}^* = \cup_{n=0}^{\infty}\{0,1\}^n$. A *language* over Σ is a subset of Σ^*.

A *Turing machine* (TM) is an idealised abstract model of *binary* computers consisting of an *infinite memory tape* that holds binary strings, a *reading-writing head* and a *program* that governs the head's movements on the tape and its reading and writing actions. In this setup, the cost of computation for input of size n is the maximum number of head transitions (time complexity) or space used on the memory tape (space complexity). This is the natural measure used in Complexity Theory and is commonly referred to as the worst case complexity, in contrast to the average case complexity where the cost is averaged over all instances.

We shall now explore some of the complexity classes of interest in Cryptology. We reiterate that all definitions in Complexity Theory assume the worst case cost for decision problems and that the cost is measured as a function of the input length, unless otherwise specified. The classification is with respect to whether a random decision problem can be correctly decided to be in a given language or not.

The *polynomial time* complexity class $\mathcal{P}$ is the set of decision problems for which there exists a *deterministic* TM that decides whether an instance is in the language or not in a polynomial number of steps.

The *bounded-error probabilistic polynomial time* complexity class $\mathcal{BPP}$ is the set of decision problems for which there exists a *probabilistic* TM that also has access to a string of random bits and correctly accepts or rejects with probability $1/2 + \epsilon$ for any $\epsilon > 0$. A machine with this property is usually referred to as a *two-sided Monte*

[1] Average complexity should cover almost all practical instances. The best case instances (with the cheapest cost to break) should be rare and hard to find or of no practical interest

Carlo Turing machine. Note that confidence can be increased by running the TM an odd number of times then taking the majority vote.

If, in the definition of $\mathcal{BPP}$, the TM correctly accepts with probability at least 1/2 but always correctly rejects, then the TM is called a *one-sided Monte Carlo Turing machine* and the corresponding class is called *random polynomial time* $\mathcal{RP}$. If the TM correctly *rejects* with probability at least 1/2 but always correctly accepts, then we call the corresponding class co-$\mathcal{RP}$.

The *zero-error probabilistic polynomial time* $\mathcal{ZPP}$ class is defined to be $\mathcal{RP} \cap$ co-$\mathcal{RP}$, and its corresponding TM is called a *Las Vegas Turing machine*. Note that this class has a TM that always answers correctly but only has an estimated polynomial running time which does not necessarily mean a polynomial upper-bound on the running time. We hence have $\mathcal{P} \subset \mathcal{ZPP}$.

For the next complexity class, we will need to introduce *non-deterministic Turing machines* (NDTMs). These are the same as the traditional TMs except that the head transition function is one-to-many, meaning that the NDTM makes all the possible next transitions in parallel. This can be thought of as making enough copies of the current TM then running each new TM with a different next transition, and so on.

The *non-deterministic polynomial time* $\mathcal{NP}$ class consists of the decisional problems that admit solution with a NDTM in polynomial time and which can be verified with a traditional TM in polynomial time too given an auxiliary string of length polynomial in the length of the problem (TM accepts on input of a *decisional problem* and an *auxiliary string*). It is conjectured by Cook that $\mathcal{P} \neq \mathcal{NP}$ but this still stands unproven to this date, [Coo06].

A problem X is *reducible* to a problem Y if there exists a deterministic TM for X that can decide membership in X given oracle access to a TM for Y. Informally speaking, this reduction means that X is no harder than Y, and we write $X \leq Y$. If, furthermore, Y is reducible to X then they are said to be *polynomial-time equivalent*.

For a complexity class C, a decision problem is dubbed *C-hard* if every problem in C is reducible to it. If in addition this decision problem is itself in C then it is referred to as being *C-complete*. The intuition here is that C-complete problems are the hardest problems in C. (C is usually $\mathcal{NP}$).

In this thesis, Turing Machines will be too abstract for our work, so we will use *algorithms* and study their complexity in a similar fashion to what was described here, as is the common practice. The randomised complexity classes will mainly appear in Chapter 6 while the other classes are general and affect all chapters.

Asymptotic notation

Due to the difficulty of estimating the exact cost of a given Turing machine, we usually content ourselves with a good approximation or an asymptotic formula for it. There is a well established set of tools and notation for this purpose, and stated here are some of the notation that we will be using.

The most widely used notation is the *big-Oh* notation, where we write $f = O(g)$ and mean that f grows no faster than g, asymptotically, to within an absolute constant multiple. More rigorously, it means that $\exists c, n_0 \in \mathbb{N}: 0 \leq f(n) \leq cg(n) \ \ \forall n \geq n_0$. The lesser used notation $f = o(g)$ is used to indicate that f is not only asymptotically bounded by a multiple of g but that it is also asymptotically negligible compared to g, or more formally said $\forall c > 0, \exists n_0 > 0: 0 \leq f(n) \leq cg(n) \ \ \forall n \geq n_0$. This *small-Oh* notation appears mainly as $o(1)$ denoting a quantity which tends to zero as $n \to \infty$.

When giving lower bounds we write $f = \Omega(g)$ if $\exists c, n_0 \in \mathbb{N}: f(n) \geq cg(n) \geq 0 \ \ \forall n \geq n_0$. This means that f grows asymptotically at least as fast as g, to within an absolute constant multiple. To give the exact order of a function f we write $f = \Theta(g)$ when $\exists c_1, c_2, n_0 \in \mathbb{N}: c_1 g(n) \leq f(n) \leq c_2 g(n) \ \ \forall n \geq n_0$. We summarise these below

$$
\begin{aligned}
f &= O(g) &&\iff \exists c, n_0 \in \mathbb{N}: 0 \leq f(n) \leq cg(n) && \forall n \geq n_0.\\
f &= o(g) &&\iff \forall c > 0, \exists n_0 > 0: 0 \leq f(n) \leq cg(n) && \forall n \geq n_0.\\
f &= \Omega(g) &&\iff \exists c, n_0 \in \mathbb{N}: f(n) \geq cg(n) \geq 0 && \forall n \geq n_0.\\
f &= \Theta(g) &&\iff \exists c_1, c_2, n_0 \in \mathbb{N}: c_1 g(n) \leq f(n) \leq c_2 g(n) && \forall n \geq n_0.
\end{aligned}
$$

One useful function that frequently appears in practical complexity analysis of sub-exponential algorithms is the so-called *L function*: For constants $c > 0$ and $\alpha \in [0, 1]$, define ($e = 2.71828\ldots$ is the base of the natural logarithm)

$$L_n(\alpha, c) := e^{(c+o(1))\cdot(\log n)^{\alpha}(\log\log n)^{1-\alpha}}.$$

Note that when $\alpha = 0$ we get a polynomial cost $L_n(0,c) = (\log n)^{c+o(1)}$ in the length of n, and when $\alpha = 1$ this turns into an exponential cost $L_n(1,c) = n^{c+o(1)}$. In general, the smaller α is the better.

1.3.1 Recurrence equations of the form $\mathcal{R}(n) = a\mathcal{R}(n/b) + cn + d$

This class of recurrence equations is common in the analysis of recursive algorithms and will be needed in Chapter 4. We now solve this equation in the general case.

We are interested in solving the recurrence equation $\mathcal{R}(n) = a\mathcal{R}(n/b) + cn + d$ subject to the *threshold condition*

$$\mathcal{R}(n) = f(n) \quad \text{for } n < T := b^\tau,$$

where T is a fixed threshold value and f is a given function. We distinguish two cases according to whether a and b are equal or not.

- Let us examine the case where $a \neq b$ first. Set $k = \log_b n$, then by induction we get

$$\mathcal{R}(b^k) = a^\ell \mathcal{R}(b^{k-\ell}) + \frac{(a/b)^\ell - 1}{a/b - 1} \cdot cb^k + \frac{a^\ell - 1}{a-1} d \quad \text{for any } \ell \in \mathbf{N}.$$

We want ℓ to be the least number such that $b^{k-\ell}$ is just below the threshold T i.e. $b^{k-\ell} < b^\tau \leq b^{k-(\ell-1)}$, so we set

$$\ell = \lceil k - \tau \rceil = \lceil \log_b(n/T) \rceil =: \ell_b(n/T). \tag{1.6}$$

We then get that (using $\lceil x \rceil = x + \{-x\}$, where $\{x\}$ is the fractional part of x)

$$k - \ell = k - \lceil k - \tau \rceil = \tau - \{\tau - k\}.$$

So, for $n \geq T$ and $a \neq b$, we have the following solution

$$\mathcal{R}(n) = a^{\ell_b(n/T)} f\left(\frac{T}{b^{\{\log_b(T/n)\}}}\right) + \frac{(a/b)^{\ell_b(n/T)} - 1}{a/b - 1} \cdot cn + \frac{b^{\ell_b(n/T)} - 1}{b-1} d. \tag{1.7}$$

- If $a = b$ then induction yields

$$\mathcal{R}(b^k) = b^\ell \mathcal{R}(b^{k-\ell}) + \ell c b^k + \frac{b^\ell - 1}{b-1} d \quad \text{for any } \ell \in \mathbb{N}.$$

With the same choice of ℓ as before, we get for $n \geq T$ and $a = b$

$$\mathcal{R}(n) = b^{\ell_b(n/T)} f\left(\frac{T}{b^{\lfloor \log_b(T/n) \rfloor}}\right) + cn\ell_b(n/T) + \frac{b^{\ell_b(n/T)} - 1}{b-1} d. \tag{1.8}$$

1.4 Computationally hard problems

Next, we will describe a few problems from the field of Computational Number Theory that are believed to be computationally hard to solve. The first two problems are the main subject of Chapter 6 and will be described in greater detail.

Before starting, we first introduce the concept of *generic algorithms* as we will present some complexity results which depend on this notion. An algorithm operating on group elements is called *generic* if it only uses the group as a black-box, meaning that it only multiplies elements of the group, computes inverses and can check for equality of elements. More formally, in the generic group model, the only operations that may be used are performed through the following three oracles:

1. Evaluation: On input (a, b) outputs their product ab.

2. Inversion (Negation): On input a outputs its inverse a^{-1}.

3. Comparison: Tests if two elements a, b are equal (Redundant if bit representation is unique).

1.4.1 The Discrete Logarithm Problem (DLP)

Let $G = \langle g \rangle$ be a given a multiplicative cyclic (sub-)group of order n with an efficiently computable group law.

The Discrete Logarithm Problem (DLP) is the discrete analogue of the usual analytic logarithm problem, viz. given an element $h \in G$ different from g, find the

unique integer x (modulo n) such that $h = g^x$. We write $x = \log_g h$. A more formal definition of the problem can be stated as follows.

Definition 1.1 (DLP). *The Discrete Logarithm Problem is to compute from an input of a cyclic group $G = \langle g \rangle$ of order n and an element $h \in G$, the unique integer $x \in \{0, 1, 2, \ldots, n-1\}$ such that $h = g^x$ (all represented (uniquely) as bit-strings).*

In practice, the group order n may be unknown, but we will assume that this value is given or easily computable from the group definition.

The list of practical candidate groups where the DLP is believed to be intractable (in polynomial time) includes multiplicative groups of finite fields, $(\mathbb{Z}/n\mathbb{Z})^\times$ for a composite n, non-supersingular elliptic and hyper-elliptic curve groups, algebraic tori and ideal class-groups of imaginary quadratic fields.

Two practical instances of this problem are of particular interest to us: Discrete logarithms in finite fields where the group G is the multiplicative group $\mathbb{F}_q^\times$ of the finite field $\mathbb{F}_q$, which is a cyclic group of size $q-1$. The second source of instances is elliptic curves over finite fields, where G is a subgroup of an elliptic curve group $E(\mathbb{F}_q)$. The problem in this latter case is written additively and reads as follows: Given two points P, Q on $E(\mathbb{F}_q)$ where $Q \in \langle P \rangle = G \subset E(\mathbb{F}_q)$, find $x \in \{0, 1, \ldots, \#G-1\}$ such that $Q = xP$.

Computational Complexity of the DLP. First note that if G has smaller subgroups then we can reduce the the DLP over G to a collection of DLP's over the subgroups by the *Pohlig-Hellman reduction*, which states that we only need to solve the problem in the prime power subgroups of G. The solution is obtained using the Chinese Remainder Theorem (CRT) applied to the solutions of the restricted problem over the prime power subgroups. More concretely, suppose that $n = \#G = \prod_{i=1}^{t} p_i^{e_i}$, where p_i are distinct primes and $e_i \in \mathbb{N}$. Then G is isomorphic to a product of cyclic groups of prime-power orders: $\bigotimes_{i=1}^{t} C_{p_i^{e_i}}$. The projection map from G to some C_{p^e} is given by $h \mapsto h^{n/p^e}$. Now, by solving the projected DLP modulo $p_i^{e_i}$ for $i = 1, \ldots, t$ we can then use the CRT to reconstruct the solution modulo n, i.e. over G. To solve the projected DLP, we first solve it modulo p in C_p using any sensible method, of

which brute-force is an option if p is small enough, then using the Hensel lifting we lift the solution to $C_{p^2}, \ldots, C_{p^e}$.

Nechaev and Shoup [Nec94, Sho97] showed that the DLP needs $\Omega(\sqrt{p})$ oracle calls (group operations) with *generic* algorithms having success probability bounded away from zero ($\mathcal{BPP}$), where p is the largest prime divisor of n.

It is obvious that an exhaustive search to solve the DLP over a group G of a cryptographic size n is very inefficient as it costs $O(n)$ operations, which is an exponential cost. There are many other algorithms which have a better asymptotic cost, and we will now describe a few.

The generic lower bound of $\Omega(\sqrt{n})$ for the DLP is achievable with a space/time tradeoff modification of the brute-force method known as the *Baby-Step Giant-Step* (BSGS) method. It is due to Shanks and has an asymptotic cost of $O(\sqrt{n})$ group operations and storage for $O(\sqrt{n})$ group elements. This is a very useful generic algorithm to solve the DLP and will be used in Chapter 6 together with the Pohlig-Hellman reduction to solve DLPs on elliptic curves of *smooth order*. We describe this method here for future reference.

The BSGS method. If we let $m = \lceil \sqrt{n} \rceil$ and write the solution of $h = g^x$ as $x = im + j$ with $i, j \in \{0, 1, \ldots, m-1\}$, then we see that

$$(g^{-m})^i h = g^j.$$

So, if we precompute (g^j, j) for $j \in \{0, 1, \ldots, m-1\}$ and sort them by the first entry then we can identify the correct value of i by trying all possible $i \in \{0, 1, \ldots, m-1\}$ and checking if $(g^{-m})^i h$ matches any of the precomputed values g^j for some $j \in \{0, 1, \ldots, m-1\}$. ■

Another method which also achieves the generic lower bound is the *Pollard ρ method*. It is a Las Vegas probabilistic algorithm exploiting the birthday-paradox with expected running time of $O(\sqrt{n})$ and a negligible storage requirement which makes it favourable in many cases.

The Pollard ρ method. It proceeds by defining a pseudo-random walk on a finite graph defined by a function f over G, and when a collision happens in the

walk we get a solution with a high probability. A collision is expected to occur after about $\sqrt{|G|}$ steps, as expected by the birthday paradox. Using the Floyd's cycle-finding method for collision detection, we only need minimal storage and expect to find a collision in an expected time roughly equal to the square root of the points in the cycle. The actual function f that is used in practice is given by first setting $x_0 = 1$ and then using the iterative function

$$x_+ = f(x) := \begin{cases} hx & \text{if } x \in S_1 \\ x^2 & \text{if } x \in S_2 \\ gx & \text{if } x \in S_3 \end{cases},$$

where the notation $x_+ = f(x)$ denotes the recurrence $x_{n+1} = f(x_n)$ for $n = 0, 1, 2, \ldots$ and the sets S_1, S_2, S_3 have roughly the same size and form a partition of G with the condition that $1 \notin S_2$. The ith term of the sequence induced by f has the form $x_i = g^{a_i} h^{b_i}$ for $i \geq 0$ with $a_0 = b_0 = 0$ as $x_0 = 1$. The corresponding iteration function for the exponents a_i and b_i is

$$(a,b)_+ = \begin{cases} (a, b+1) \mod n & \text{if } x \in S_1 \\ (2a, 2b) \mod n & \text{if } x \in S_2 \\ (a+1, b) \mod n & \text{if } x \in S_3 \end{cases},$$

Using the Floyd's cycle-finding method will yield a pair (x_i, x_{2i}) such that $x_i = x_{2i}$, i.e. $g^{a_i} h^{b_i} = g^{a_{2i}} h^{b_{2i}}$ which means that $g^{a_i + b_i \log_g h} = g^{a_{2i} + b_{2i} \log_g h}$ and hence we have

$$(b_i - b_{2i}) \log_g h \equiv (a_{2i} - a_i) \pmod{n}.$$

So, if $b_i \not\equiv b_{2i} \pmod{n}$ which is the case with high probability, then we can retrieve $\log_g h$ by a simple division modulo n.

Note that it turns out that the walk as defined above is not random enough as supposed in the theoretical analysis. The set G should be partitioned into about 20 subsets to have a good practical performance, [Tes98, Tes01]. The corresponding

iteration function can be taken to be

$$x_+ = f(x) = xg^{m_s}h^{n_s} \text{ if } x \in S_s \quad \text{for } s \in \{1,\ldots,r\} \text{ and } r \approx 20,$$

where $\{S_i\}_{i=1}^r$ are of roughly the same size and form a partition of G, and m_s, n_s are integers. ■

Another similar method is also due to Pollard and is called the *Pollard λ method* or *Tame and Wild Kangaroos*. It has an expected running time of $\Theta(\sqrt{n}\log n)$ and also requires little storage.

Given the square-root best cost of these generic algorithms, the corresponding key size for cryptographic primitives requiring a security level 2^n is $2n$ bits, provided that the computational group they are based on is generic. The multiplicative group of a finite field unfortunately fails to be generic and admits a subexponential attack on the DLP, which is referred to as *index calculus* – hence, the key sizes for primitives which are based on non-generic groups need to be increased accordingly.

Index calculus gives a sub-exponential time algorithm for the DLP in any group for which we can define a *factor base*, i.e. if there is measure to decide how "small" an element is, and such that a significant portion of elements can be efficiently expressed as a product of these elements. This algorithm is mainly used over finite fields, but does not work for elliptic curves.

1.4.2 The Diffie-Hellman Problem (DHP)

Let $G = \langle g \rangle$ be a given a multiplicative cyclic (sub-)group of order n with an efficiently computable group law.

Definition 1.2 (DHP). *Given $g^a, g^b \in G$ where a, b are unknown, the DH problem is to compute g^{ab}.*

A related problem is the *Decisional Diffie-Hellman Problem* (DDH): Given group elements g, g^a, g^b, g^c, decide whether $g^{ab} = g^c$ or not. This problem also seems intractable in general, but is known to be easy for supersingular elliptic curves because of the existence of efficient *pairings* (Bilinear maps).

It is easy to see that the DHP is reducible to the DLP as one can compute $a = DL(g^a)$ then on computing $(g^b)^a$ we get the desired answer. The reverse reduction is not trivial but it is known to hold for almost all cyclic groups. This will be the topic of Chapter 6.

There are a few types of DH oracles which are all polynomial-time equivalent, and we will describe two of them now. The squaring DH-oracle is an oracle that on input g^a computes g^{a^2}. The reduction of the usual oracle to the squaring oracle is straight forward using the identity $ab = \frac{1}{4}[(a+b)^2 - (a-b)^2]$, and the reverse reduction is obvious. The ϵ-DH-Oracle, for some $\epsilon > 0$, is a probabilistic oracle that solves the DHP correctly with probability at least ϵ if the input is uniformly distributed.

Fixed group generators vs. randomly chosen generators. It is worth signalling at this stage that there is a distinction between the DLP or DHP with respect to a *fixed* group generator and those with respect to a (randomly) *chosen* generator. Consider, for example, the generation of Decisional Diffie-Hellman triples (g^a, g^b, g^{ab}). Using the generic group model, Dent showed in [Den06] that if such a triple is generated by an algorithm $\mathcal{A}$ then either a or b can be extracted from the the inputs, outputs and random coins of $\mathcal{A}$; which means that one needs to know one the discrete logarithm of one of the inputs to be able to generate the DH triple. On the other hand, if the the group generator can freely be chosen then one can easily generate such triples without solving any DLP *with respect to the chosen generator*, by simply uniformly choosing a random element $h \in G$ and $y \in \mathbb{Z}$ then computing the triple

$$(h^y, h^{y^{-1} \bmod p}, h),$$

which produces uniformly distributed instances of DH triples. This trick does not work for the fixed generator case because if $h = g^x \in G = \langle g \rangle$ such that $x \not\equiv 0, \pm 1 \pmod{p}$ then

$$\mathcal{DH}_g(h^y, h^{1/y}) = \mathcal{DH}_g(g^{xy}, g^{x/y}) = g^{x^2} \neq h,$$

and the remaining non-trivial possibility $(g^y, g^{\pm 1/y}, g^{\pm 1})$ does not provide uniformly distributed DH instances.

This subtle distinction is very important to our treatment of the reduction of DLP to DHP in Chapter 6. We will have to treat the two types of DH oracles according to whether the group generator is fixed or can be chosen and given as an output to the DH oracle.

1.4.3 RSA and Rabin problems (Modular square and e^{th} roots)

In this problem, we are given a modulus $N = pq$, where p and q are large primes of roughly the same size, $e \in \mathbb{N}_{\geq 2}$ and $c \in \mathbb{Z}_N^\times$ such that

$$c = m^e \mod N$$

for some $m \in \mathbb{Z}_N^\times$, and we are asked to find m. That is to extract e-th roots modulo an RSA modulus N (See §1.5.2 for the specification of RSA).

If the factorisation of N is known then this operation is easy, as one can solve it modulo the prime factors p and q first then reconstruct the solution modulo N using the CRT or, for $e > 2$, compute $d = e^{-1} \mod \varphi(N) = (p-1)(q-1)$ and then compute $m = c^d \mod N$. However, if the factorisation of N is unknown, then for $e = 2$ it corresponds to breaking the Rabin encryption scheme which is known to be equivalent to factoring N. For $e \geq 3$, it is not known whether this problem is equivalent to factoring N or not, but there are some arguments that suggest that the RSA problem may be easier than factoring [BV98].

Factorisation of integers into primes

Given a large composite integer N, this problem asks for its factorisation into a product of prime powers. It is believed to be very hard in general as the best known general factorisation algorithm, the General Number Field Sieve (GNFS), has a heuristic sub-exponential running time of

$$L_N(1/3, \sqrt[3]{64/9}).$$

For integers of moderate sizes (around 80 digits) however, one should use the Multiple Polynomial Quadratic Sieve (MPQS) factoring method which costs $O(e^{\sqrt{\log n \log\log n}}) = L_N(1/2, 1)$ asymptotically but performs better for this range. If N has a small divisor then the Elliptic Curve Method (ECM) should be the best; otherwise, one should try Pollard's ρ method.

This problem is the basis for cryptosystems like RSA, but will unfortunately be a hindrance to us in Chapter 6 as it will prevent us from finding the auxiliary data that is needed for our analysis.

1.4.4 Lattice problems

In the process of trying to devise a practical hash function from a previous proposal [GGH96], in Chapter 5, we will have to deal with a mathematical structure called *lattices* and some of the computational problems associated with it. This section will briefly review the notion of lattices and describe some related topics. A good reference book on this topic from a cryptographic perspective is [MG02] by Micciancio and Goldwasser.

The *subset sum problem* is to decide whether a subset of a given finite set of integers $\{a_1, a_2, \ldots, a_n\}$ sum to a given integer s. This problem is known to be $\mathcal{NP}$-complete, while its computational version is $\mathcal{NP}$-hard.

Given a matrix $B \in \mathbb{R}^{m \times n}$ whose columns are linearly independent vectors $v_1, \ldots, v_n \in \mathbb{R}^m$, the corresponding *lattice* L_B is defined to be the set of all possible integer linear combinations of these vectors

$$L_B = \left\{ \sum_{i=1}^{n} c_i v_i, c_i \in \mathbb{Z} \right\} = \mathbb{Z}v_1 + \mathbb{Z}v_2 + \cdots + \mathbb{Z}v_n.$$

In other words, the lattice L_B is a discrete additive subgroup of $\mathbb{R}^m$ induced by the vectors in B, i.e.

$$L_B = \left\{ Bx : x \in \mathbb{Z}^n \right\}.$$

The matrix $B = (v_1|v_2|\ldots|v_n)$ is called *a basis* of the lattice L_B. Note that this basis is not unique and that any other matrix which is equal to BU, for some unimodular

matrix $U \in \mathbb{Z}^{n\times n}$, is another possible basis. The determinant of L_B is defined to be $\det B$.

Next we will define two important computational problems related to lattices, but before that we will first review the different definitions of a vector's length. If $w = (w_1, w_2, \ldots, w_m) \in \mathbb{R}^m$ is a vector then its length is usually defined to be its *Euclidean norm* which is given by

$$\|w\| = \sqrt{\langle w, w\rangle} = \sqrt{w_1^2 + w_2^2 + \cdots + w_m^2}$$

where $\langle .,. \rangle$ is the inner product operator. This is also known as the ℓ_2 norm. For a general $p \in \mathbb{N}$, the ℓ_p norm is given by

$$\|w\|_p = (w_1^p + w_2^p + \cdots + w_m^p)^{1/p}.$$

We denote by $\lambda(L)$ the Euclidean length of the shortest non-zero vector in a lattice L and by $\lambda(L, b)$ the distance between the vector $b \in \mathbb{R}^n$ and its closest lattice point.

The Shortest Vector Problem (SVP)

The *Shortest Vector Problem* (SVP) asks for a shortest nonzero vector in a given lattice. The answer vector is not unique, but the shortest length is unique and upper bounded by $\gamma_n \det(L_B)^{1/n}$, where $\gamma_n \sim \sqrt{n}$ is an absolute constant that depends on n only (Theorem of Minkowski). The SVP gained instant popularity when Shamir showed how to heuristically reduce the knapsack problem to SVP, [SS81].

Approximating the shortest vector problem (ApproxSVP) in any ℓ_p norm to within any constant factor less than $2^{1/p}$ is $\mathcal{NP}$-hard under randomised reductions [Mic01]. In particular, approximating the shortest vector problem is not in $\mathcal{RP}$, unless $\mathcal{NP} = \mathcal{RP}$. ApproxSVP is proved to be a proper $\mathcal{NP}$-hard problem under a reasonable number theoretic conjecture on the distribution of square-free smooth numbers.

The Closest Vector Problem (CVP)

The *Closest Vector Problem* (CVP) asks for the closest lattice point to a given point in $\mathbb{R}^m$. That is, given a vector $u \in \mathbb{R}^m$, we are asked to find a lattice point $v \in L_B$ such that the norm $\|u - v\|$ is minimal.

CVP can be reduced to SVP in the following way, which is due to Babai [Bab86]. Given a vector $u \in \mathbb{R}^m$ and a lattice L_B, construct a new lattice L_M defined by the matrix

$$M = \begin{pmatrix} B & u \\ 0 & 1 \end{pmatrix}.$$

Now, suppose that the closest vector in L_B to u is v. Then the vector $(u - v, 1)^T$ is a short vector in L_M, and hence one can *attempt* to retrieve it by solving the SVP in L_M and then recover v from the short vector.

CVP is generally regarded to be harder than SVP. In fact, CVP is known to be NP-hard to approximate to within any constant factor. Furthermore, any efficient algorithm that efficiently approximates CVP can be used to efficiently approximate SVP, which means that SVP is not harder than CVP [MG02, §3.3].

The LLL basis reduction algorithm

In 1982, A.K. Lenstra, H.W. Lenstra and L. Lovász presented the first deterministic polynomial time algorithm called LLL or L^3 which, given a basis for a lattice $L \subset \mathbb{R}^m$, can find a vector which is guaranteed to be no more than $2^{(m-1)/2}\lambda(L)$, where $\lambda(L)$ is the length of the shortest vector in L. The factor $2^{(m-1)/2}$ was later reduced to $(1+\epsilon)^m$ for arbitrarily small $\epsilon > 0$ by C.P. Schnorr at the expense of more work to be done by the algorithm.

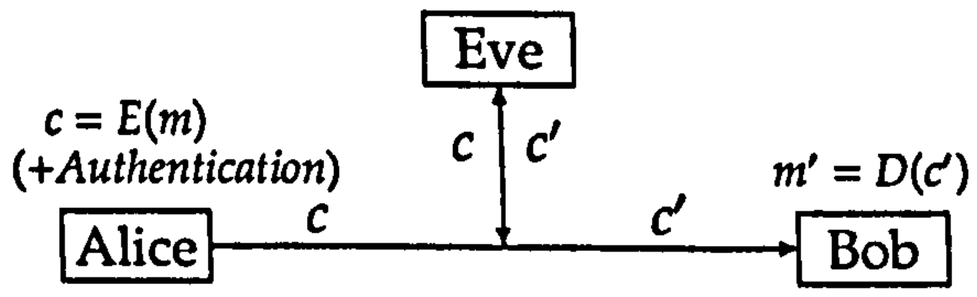
The LLL algorithm is of prime importance in cryptanalysis as almost all of the lattice based cryptographic primitives only require a good approximation to the shortest vector to be broken. Furthermore, the LLL algorithm tends to yield a vector which is much shorter than $2^{(m-1)/2}\lambda(L)$ in practice, but this depends on the quality of the basis and is not yet well understood.

1.5 General cryptography

We will now recall some general cryptographic notions. An excellent rigorous treatment of the theory of cryptography is Oded Goldreich's two volume work: *Foundations of Cryptography*, [Gol04a, Gol04b]. For a more elementary introduction to the topic see [Sti06, Sma02, Mao04, MOV97].

We should first meet our friends *Alice*, *Bob* and *Eve* who have been serving the cryptographic community and have become the de facto characters for illustrating the different cryptographic scenarios. Alice and Bob want to communicate between them but they know that Eve eavesdrops on their communications and may even have control over the channel joining them. In fact, since Eve may have full control on the communication channel she can insert, delete, modify, delay or replay any message of her choice. The Internet is a good example of such an insecure communication channel.

In order to keep the communication *confidential*, Alice and Bob use *encryption* to render their messages unintelligible to Eve. They also want to detect if Eve tampers with their communication, and for this Alice and Bob *authenticate* their communications using *message authentication codes* (MACs) or *digital signatures*.



It is imperative to keep in mind that Eve is not predictable nor does she stick to any rules. She may act *passively* but she can also act *actively* and may adaptively try to break the system. She is intelligent, malicious, devious and may have or will develop better knowledge and technology than expected. Furthermore, in the multi-users setting, she may even be one or some of the users!

Defending against all existentially possible attacks is practically impossible, so it is important to ask the question: What are we trying to protect or prevent? The answer to this question decides the correct level of security needed. For example, if we are trying to prevent forgery of money then it is *enough* to make the cost

of forged money higher than they are worth. In general, a good measure of how costly an attack is is to compare it to the cost of generic attacks such as brute force or birthday-type attacks – then, the cryptosystems designers' aim should be to give Eve no advantage over these generic attacks.

A *cryptosystem* is characterised by five elements $\mathcal{P}, \mathcal{C}, \mathcal{E}, \mathcal{D}$ and $\mathcal{K}$, where

- $\mathcal{P}$ is the *plaintext space* from which messages are drawn
- $\mathcal{C}$ is the *ciphertext space* where the encrypted messages live
- $\mathcal{E}$ and $\mathcal{D}$ are respectively the *encryption* and *decryption* algorithms family parametrised by keys drawn at random from the *key space* $\mathcal{K}$.

The cryptosystem should satisfy the *soundness* condition

$$\forall k \in \mathcal{K}, \forall m \in \mathcal{P}: D_k(E_k(m)) = m$$

or at least to hold for the overwhelming majority of keys and messages, as is the case for the NTRU cryptosystem for example [HPS98]. The algorithms are also required to be computationally efficient, and not to leak information about the secrets (plaintext or key) that can be extracted by another algorithm in polynomial time.

Note that in the case of *symmetric cryptography* the encryption and decryption keys are the same or easily derivable from each other – hence the symmetric property; as opposed to *asymmetric cryptography* (public-key cryptography) where the key is composed of a *private key* for decryption which is kept secret and a *public-key* for encryption which is published, with the assumption that deriving the private key from the public key is computationally infeasible unless given access to some *trapdoor* information (e.g. the secret-key itself).

The public key and private key can be thought of as being related via a *one-way function* (OWF). Set $\Sigma = \{0, 1\}$ and let $f: \Sigma^* \to \Sigma^*$ be a bijective efficiently computable function. Then, we say that f is an OWF iff there is no polynomial-time algorithm that, given $y \in \Sigma^*$, can find $x \in \Sigma^*$ such that $y = f(x)$ or rejects if no such x exists (inversion). Furthermore, we insist that the image of a string under f should be at

most polynomially longer or shorter then the input. If the OWF can be efficiently inverted given access to an additional (secret) string then it is called a *trapdoor one-way function*. Currently, there is no published proof that such functions exist, and in fact, if they do then that will imply that $\mathcal{P} \neq \mathcal{NP}$ which will be a breakthrough in Complexity Theory.

The symmetric and asymmetric approaches to cryptography both have their own advantages and disadvantages. In practice, symmetric cryptography is orders of magnitude faster than asymmetric, but key distribution and management prove to be difficult problems. Asymmetric cryptography on the other hand offers a neat solution to this latter problem and also provides the useful primitive of digital signatures. Furthermore, it enjoys the benefits of Provable Security because of its mathematical structure.

The practical approach for harnessing the best of the two approaches is to use the KEM/DEM paradigm: The *Key Encapsulation Mechanism* (KEM) uses an asymmetric cipher to generate a symmetric-key and then encrypts it producing an *encapsulation* of the key. A symmetric cipher is then used, with the generated key, to encrypt the plaintext. This second part is called the *Data Encapsulation Mechanism* (DEM). The ciphertext is then sent as the key-encapsulation together with the symmetrically encrypted plaintext. This approach grants the benefits of both models of encryption and is furthermore provably secure subject to some security requirements on the KEM and DEM, see [CS03].

1.5.1 Hash functions

Encryption and decryption are the most popular cryptographic *primitives* but there are many others. We often need a special type of functions known as *hash functions*, commonly used in conjunction with *digital signatures*. They are also used for commitments, integrity checking (Modification Detection Codes (MDC), unkeyed hash functions) and authentication with integrity (Message Authentication Codes (MAC), keyed hash functions).

A hash function takes a bit-string from $\{0,1\}^*$ and maps it to a fixed length string space $\{0,1\}^n$, for some fixed $n \in \mathbb{N}$ called the *hash length*. Cryptographic

hash functions are a special type of the general hash functions, where besides the basic property of mapping large domains to small ranges, they have some extra information security requirements, such as lower-bounding n with a function of the level of security.

We say that two messages (bit-strings) collide if they have the same hash value under the same hash function. It is clear from the definition of hash functions that collisions are inevitable, but if the cost of finding such collisions is $\Omega(\sqrt{n})$, which is achievable with a generic birthday attack, then we call such a hash function a *collision resistant hash function* (CRHF).

We also impose that the cost of inversion, i.e. finding a pre-image of a given random hash value, should be $\Omega(n)$. We refer to such hash functions as *one-way hash functions* (OWHF).

The are many other properties that may be required for certain protocols such as being 2^{nd} *pre-image resistant*, where we are given a message together with its hash value and we are asked to find another message that hashes to the same value, or to be a *pseudo-random function* (PRF), i.e. computationally indistinguishable from a truly random function. These and other properties are described in more detail in [MOV97, Chapter 9].

Families of hash functions that are used in practice include the MD and the closely related SHA family, both considered weak by virtue of recent attacks [BCJ+05, WLF+05, WY05]. The recommended hash function to be used at the time of this writing is SHA256. Recent proposals for hash functions include VSH [CLS05], LASH [BPS+06] and FFT based hash function [LMPR06].

1.5.2 Some cryptographic primitives

In this section, we will recall the description of some standard practical realisation of cryptographic primitives that are related to this thesis, but we will only describe them in their textbook versions. The actual protocols used in practice are slightly changed so to make them provably secure.

• **RSA.** This is an encryption-method that is widely used in electronic commerce protocols. It was introduced in 1977 by Ron Rivest, Adi Shamir and Leonard

Adleman from MIT [RSA77] – hence the name RSA. Modular arithmetic is at the heart of this encryption-method as it consists of one modular exponentiation, which is its computational bottleneck.

To use (the textbook version of) RSA Bob, who is to receive encrypted messages from Alice and others, first computes a big number N that is a product of two equally-sized primes p and q, and then computes two numbers e and d satisfying $ed \equiv 1 \bmod (p-1)(q-1)$. Bob's public-key is then the pair (N,e), while d is his secret-key and is therefore kept secret. To encrypt a message m, Alice fetches Bob's public-key (N,e) and computes $c \leftarrow m^e \bmod N$ then sends c to Bob, who deciphers it by computing $m \leftarrow c^d \bmod N$ using the secret-key d.

• **Diffie-Hellman key exchange.** This is a primitive for unauthenticated key agreement which is sometimes called *exponential key exchange* as it utilises exponentiation.

Suppose that Alice and Bob had already agreed on a large prime p and a generator g of $\mathbb{Z}_p^\times$. The idea here is to agree a key of the form

$$K = g^{xy} = (g^x)^y = (g^y)^x.$$

If Alice and Bob want to agree on a session key over an open channel then Alice chooses a random $x \in \{2,\ldots,p-2\}$ and sends g^x to Bob, who similarly chooses a random $y \in \{2,\ldots,p-2\}$ and sends g^y to Alice. Now they can both compute the shared key as $K = (g^x)^y = (g^y)^x = g^{xy}$.

• **ElGamal public-key cryptosystem.** The security of the ElGamal cryptosystem is based on the intractability of the DLP (and the DHP).

The public key is a triple (p,g,g^a), where p is a randomly generated large prime, $\langle g\rangle = \mathbb{Z}_p^\times$ and a is the secret key which is a random integer from the set $\{2,\ldots,p-2\}$.

To encrypt a message $m \in \{1,\ldots,p-1\}$, Alice selects an integer k randomly from the set $\{2,\ldots,p-2\}$ and computes the ciphertext $c = (\hat{g},\hat{m}) = (g^k, m\cdot(g^a)^k)$ using Bob's public-key (p,g,g^a). To decrypt $c = (\hat{g},\hat{m}) = (g^k, mg^{ak})$, Bob simply computes $m = \hat{m}\cdot\hat{g}^{-a}$.

Note. It should be emphasised that the versions described here are not secure for practical applications; they only serve as a motivating theoretical tool. A proper implementation of RSA should use the provably secure OAEP padding construction [JK03, FOPS04], and for the elliptic curve version of ElGamal encryption one should use ECIES [BSS04, Sma01]. Authentication is also needed for the Diffie-Hellman key exchange to avoid the man-in-the-middle attack.

1.6 Motivation (Problems addressed in this thesis)

In this section we motivate the problems that we will address in this thesis. Our main source of problems is cryptographic standards and governmental recommendations as they constitute the best source to infer issues that directly relate to cryptographic *practice*. Another direct source of motivation has come from the recent major academic breakthrough in attacking the the MD family of hash functions which includes the popular MD5 hash function.

1.6.1 NIST 15,360-bit recommendation

NIST (*National Institute of Standards and Technology*, USA) has recently recommended using RSA moduli of sizes as big as 15360 bits to match the security level of AES-256 [Nat06, p. 63], see Table 1.1. No prior work has been done to study efficient arithmetic around these operand sizes nor do we know what is the best strategy to proceed with a practical implementation. With this in mind, it is now worthwhile to explore the improvements that can be made by using asymptotically faster multiplication methods in combination with any "tricks" that may render them practical even for moderate sizes.

Security level	Algorithm	Modulus bit-size
80	2TDEA	1024
112	3TDEA	2048
128	AES-128	3072
192	AES-192	7680
256	AES-256	15360

Table 1.1: Comparable strengths

Chapter 4 is based on [BS07] and aims to investigate a set of possibilities from straight Montgomery and Barrett arithmetic through to combining them with Karatsuba and Toom-Cook style techniques. We will see that a novel use of an error detection technique called *wooping* [FS03] will allow us to overcome the difficulties that arise when trying to go beyond the obvious simple substitution of classical multiplication methods with faster ones. These difficulties are mainly due to carry-propagation when computing upper-half products with recursive methods, a problem that does not arise when using traditional combinations such as the Karatsuba-Comba-Montgomery (KCM) method [GAST05, Sco96].

Using a formal computational cost model, we estimate the exact cost of the Montgomery and Barrett modular reduction algorithms. We then introduce some variants using the Karatsuba and Toom-3 multiplication methods, and analyse the savings that can be theoretically achieved. These variants have been implemented in C using the GMP library (GNU Multiple Precision arithmetic library) [Gra07a], and the relevant results are reported here and compared with the theoretical estimates.

1.6.2 NIST cryptographic hash project

Due to the recent attacks that were first discovered and described by X. Wang on the MD family of hash functions, which includes the popular MD5, RIPEMD and SHA-1 [BCJ+05, WLF+05, WY05], NIST has initialised an international effort to develop a few new cryptographic hashing algorithms through public academic competition, similar to the competition that contributed to the development process for the Advanced Encryption Standard (AES).

NIST has held two workshops to review and assess the status of the previously NIST-approved hash functions, to discuss possible future options and to discuss hash function research in preparation for launching such a competition.

We note that all of the recently broken hash functions are essentially derived from the same design and are constructed using somewhat ad-hoc techniques. In contrast, other areas of cryptography have replaced ad-hoc construction with well defined sets of design principles. Examples include the wide-trail design strategy of

AES [DR02, Chapter 9], or the rigorous application of reductionist provable security techniques as in the context of RSA-OAEP [BR94, FOPS01].

Given the popularity of provable security and the development of a *provably collision resistant* hash function called VSH [CLS05], the time could not be any better for trying to devise a similar method by attempting to relax previous inefficient attempts so to make them practical. This resulted in a hash function that we called LASH [BPS+06] which will be the topic of Chapter 5. In this chapter, we will show that the lattice based hash function that was previously suggested by Goldreich, Goldwasser and Halevi [GGH96] is not secure as a cryptographic hash function when we fix any *concrete* set of parameters. We then adapt the GGH construction to give our concrete proposal LASH, [BPS+06]. Various recent attacks on this construction are briefly sketched with comments on their significance, as well as various implementation tricks.

1.6.3 SECG/NIST standards for curves

A number of cryptographic standards for elliptic-curve cryptography (ECC) have been developed in the few past decades. These ease the task of adopting the latest cryptographically sound techniques while keeping the different engineered components inter-operable, which is of prime importance to the already deployed industrial applications.

Some of the standardising bodies that showed interest in ECC are the *Standards for Efficient Cryptography Group* (SECG), *National Institute of Standards and Technology* (NIST), *American National Standards Institute* (ANSI), *International Organisation for Standardisation* (ISO) and the *Institute of Electrical and Electronics Engineers* (IEEE).

Of special interest to us are the SECG and NIST standards as they both recommend a common set of 15 elliptic curves. The respective publications can be retrieved from `http://www.secg.org/index.php?action=secg,docs_secg` (SEC2: Recommended Elliptic Curve Domain Parameters) and `http://www.itl.nist.gov/fipspubs/by-num.htm` (FIPS 186-2: Digital Signature Standard (DSS)– 00 January 27).

According to FIPS 186-2, the sets of recommended curves were pseudo-randomly generated over prime fields $\mathbb{F}_p$ (for p of bit-size 192, 224, 256, 384, 521) and binary fields $\mathbb{F}_{2^m}$ (for extension degrees $m \in \{163, 233, 283, 409, 571\}$). The choices of these field parameters were made to match the standard security levels, see Table 1.2 for the correspondence.[2]

Security level	Algorithm	Bit size of p for $\mathbb{F}_p$	Degree m for $\mathbb{F}_{2^m}$
80	SKIPJACK	192	163
112	Triple-DES	224	233
128	AES Small	256	283
192	AES Medium	384	409
256	AES Large	512	571

Table 1.2: Field parameters.

On the theoretical side, the equivalence between the DLP and DHP problems was shown by Maurer in 1994 but subject to an existence condition of auxiliary groups with a smooth order [Mau94]. His work was then reexamined by Muzereau *et al.* [MSV04] for the special case of elliptic curves used in practical cryptographic applications, namely the curves from the SECG and NIST standards. Chapter 6 improves on the latter and gets very close to the tightest possible reduction, and we prove that our results are unlikely to be significantly improved upon using Maurer's method [Ben05a].

1.7 Overall structure of the thesis

The two chapters 2 and 3 will review some standard generic algorithms over groups, modular arithmetic, arithmetic of elliptic curve, exponentiation and asymptotically faster integer multiplication methods. These chapters review the background material that is necessary for the developments in the next chapters. We also introduce

[2]The elliptic curve domain parameters over a given prime field $\mathbb{F}_p$ are given by a sextuple (p, a, b, G, n, h), where p is the characteristic of the filed, a and b define the elliptic curve E over $\mathbb{F}_p$: : $y^2 = x^3 + ax + b$, $G = (X_G, Y_G)$ is a base point on $E(\mathbb{F}_p)$ of prime order n, and h is the group order cofactor i.e. $nh = \#E(\mathbb{F}_p)$. In the case where the elliptic curve is defined over a binary finite field $\mathbb{F}_{2^m}$, the domain parameters become a septuple $(m, f(X), a, b, G, n, h)$, where the parameters a, b, G, n, h keep their meaning from the prime field case but the elliptic curve is here defined by $y^2 + xy = x^3 + ax^2 + b$, m is the extension degree and $f(X)$ is a degree m irreducible polynomial over $\mathbb{F}_2$ that defines the extension field $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$.

our wooping technique in Chapter 3 which we will use to speed up the RSA operation in Chapter 4.

Chapters 4 and 5 address the issue of efficiency when designing a practical cryptographic primitive. Chapter 4 relies mainly on Chapter 3 and studies the possible ways of implementing modular arithmetic at very large operand sizes, motivated by the NIST recent key sizes recommendation. Chapter 5 takes the (inefficient and insecure) GGH hash function proposal [GGH96] and tries to design a practically efficient hash function, which despite being better in terms of efficiency loses out on the provable security side as we cannot argue about its security contrary to its predecessor [GGH96].

Chapter 6 is theoretical and is concerned with the DLP and DHP problems and their reduction to each other. There, we show the equivalence of the two problems and establish lower bounds on the difficulty of the elliptic curves DHP based on the generally accepted hardness assumption of the DLP. This is achieved by optimising the reduction method and parameters using material from Chapter 2.

We finally conclude in Chapter 7 where we summarise our results and comment on them. We then list a number of open problems that we feel are of interest for future research and suggest some possible solutions that need to be developed and investigated further.

Chapter 2

Fundamental algorithms

"The mathematician's pattern, like a painter's or the poet's, must be beautiful [. . .] Beauty is the first test; there is no permanent place in the world for ugly mathematics."

— Godfrey Harold Hardy

In this chapter we recall and introduce a few generic algorithms for modular arithmetic, modular square roots, elliptic curve arithmetic and exponentiation. The study of fast multiplication of arbitrary precision integers is delayed to the next chapter.

We start with the central operation of modular reduction, as it is a shared component between all the subsequent chapters of this thesis and is very important to many asymmetric cryptographic primitives. Recall that if we are given two integers z and m then we can divide them using Euclidean division to get

$$z = qm + r, \quad \text{where } q \in \mathbb{N} \cup \{0\} \text{ and } 0 \leq r < m.$$

We call q the *quotient* and r the *remainder* (both exist and are unique). In modular reduction, we only want to calculate the remainder as we do not need the quotient. We will now explore some practical methods for this computation.

2.1 Modular reduction and multiplication

The obvious way of reducing using Euclidean division is a good choice when the modulus is small, but is inefficient for the moderate or large operand sizes used in practice. We will describe three main methods that are commonly used in practice for this purpose. Some adaptations of these methods are developed in Chapter 4 for the case of very large operands.

Let us consider a machine where we represent large integers as arrays of integers in base b, where $b = 2^\beta$ and β is the word length. If z is a $2n$ words long integer then we use subscripts ℓ and u to denote the lower and upper halves:

$$z_\ell = z \bmod b^n \quad \text{and} \quad z_u = \lfloor z/b^n \rfloor .$$

2.1.1 Special moduli

Suppose that the modulus m is equal to $b-1$ (the largest number that can fit in a word) and $z = z_0 + z_1 b + \cdots + z_m b^m$ then note that $b \equiv 1 \pmod{b-1}$ implies that

$$b^k \equiv 1 \pmod{b-1} \quad \forall k \in \{0, 1, \ldots, m\}.$$

This remark allows us to bring reduction modulo $b-1$ to a number of simple addition of word-sized integers, namely

$$z \bmod (b-1) = z_0 + z_1 + \cdots + z_m \bmod (b-1).$$

Depending on the number of terms and their sizes, a second or more similar reductions may be necessary. This technique can easily be generalised to moduli of the form $b^t - a$ for a small integer a as shown in Algorithm 1.

For the next two reduction methods, namely the Barrett and Montgomery methods, we consider the problem of reducing $2n$-word integers modulo a given fixed n-word modulus m. The Barrett and Montgomery reduction methods are techniques used when the *modulus is fixed*, as is the case in RSA where many modular operations are carried out modulo the same fixed modulus $N = pq$. These special methods

Algorithm 1 Reduction modulo $m = b^t - a$
Input: Integer z.
Output: $z \bmod m$.

1: $q_0 \leftarrow \lfloor z/b^t \rfloor, r_0 \leftarrow z - q_0 b^t, r \leftarrow r_0, i \leftarrow 0$.
2: **while** $q_i > 0$ **do**
3: $\quad q_{i+1} \leftarrow \lfloor aq_i/b^t \rfloor, r_{i+1} \leftarrow aq_i - q_{i+1}b^t$.
4: $\quad i \leftarrow i + 1, r \leftarrow r + r_i$.
5: **end while**
6: **while** $r \geq m$ **do**
7: $\quad r \leftarrow r - m$
8: **end while**
9: return r

exploit this fact to decrease the cost of modular reduction by pre-computing some values to avoid the expensive divisions.

2.1.2 Barrett reduction

Let z be a $2n$-word integer and m be a fixed n-word modulus. As was mentioned at the beginning of this section, we can reduce z modulo m using Euclidean division of z by m: $z = qm + (z \bmod m)$. Barrett's idea is to avoid division by computing a good estimate for the quotient $q = \lfloor z/m \rfloor$ as follows

$$q = \left\lfloor \frac{z_u b^n + z_\ell}{m} \right\rfloor \approx \frac{b^n z_u}{m} = \frac{b^{2n}}{m} \cdot \frac{z_u}{b^n} \approx \left\lfloor \frac{\mu z_u}{b^n} \right\rfloor =: \tilde{q} \quad \text{where } \mu = \left\lfloor \frac{b^{2n}}{m} \right\rfloor.$$

Note that $\tilde{q} \leftarrow \lfloor (\mu z_u)/b^n \rfloor = (\mu z_u)_u$, and so it can be computed as an upper-half product (with the help of our wooping technique introduced in section 3.3 on page 66).

It can be shown that if $z < m^2$ then $q - 2 \leq \tilde{q} \leq q$. So a good estimate for the remainder is $z - \tilde{q}m$ which we can correct by subtracting m from it at most twice. Algorithm 2 describes this method in detail, [MOV97, p. 604].

2.1.3 Montgomery reduction and multiplication

Notice that if we represent modular residues modulo m as xR and yR for some fixed R satisfying $\gcd(R, m) = 1$, then their (integer) product is xyR^2, which can easily be brought to the canonical residual form xyR by dividing it by R. If we choose

Algorithm 2 Barrett reduction
Input: n-word modulus m, $\mu = \lfloor b^{2n}/m \rfloor$ and $z < m^2$.
Output: $z \bmod m$.

1: $z' \leftarrow \lfloor z/b^{n-1} \rfloor, \quad \tilde{q} \leftarrow \lfloor z'\mu/b^{n+1} \rfloor$
2: $r \leftarrow (z \bmod b^{n+1}) - (\tilde{q}m \bmod b^{n+1})$
3: **if** $r < 0$ **then**
4: $\quad r \leftarrow r + b^{n+1}$
5: **end if**
6: **while** $r \geq m$ **do**
7: $\quad r \leftarrow r - m$
8: **end while**
9: Return r

$R = b^n$, where b is the word size, then this modular division turns out to be easy to implement and amounts to about two modular multiplications only. This is the principle that is behind the Montgomery reduction [Mon85], which we will now introduce in detail.

We will first introduce a simple version of the Montgomery reduction which operates at the bit level and is suitable for bit-serial hardware implementation. Suppose we want to compute $z/2^n \bmod m$, where m is an *odd* modulus and z is a number less than $m2^n$. To divide by 2^n we simply halve n times, and the trick to do this cheaply is as follows:

- If x is even then we shift z to the right by one bit.
- If x is odd then $x \equiv x + m \pmod{m}$ which is even! So we shift $x + m$ to the right by one bit.

Let us for example compute $1303455736/2^{16} \bmod 2133$, which is equal to 20155. Figure 2.1 illustrates the previous operations step by step by showing the bit representations of the operands after each stage of the reduction, using the shapes ■ and □ to denote bit values 1 and 0 respectively.

In general, let $R = 2^n$ and m be an n-bit modulus and suppose that we want to compute $z/R \bmod m$, where z is less than mR. Note that if we add a multiple of m to z then it remains the same modulo m. Furthermore, if we add a suitable multiple of m to z such that the lower n-bits of the sum are all zero then we can exchange the modular division by R for a very cheap bit-shift operation. That is to say, we want

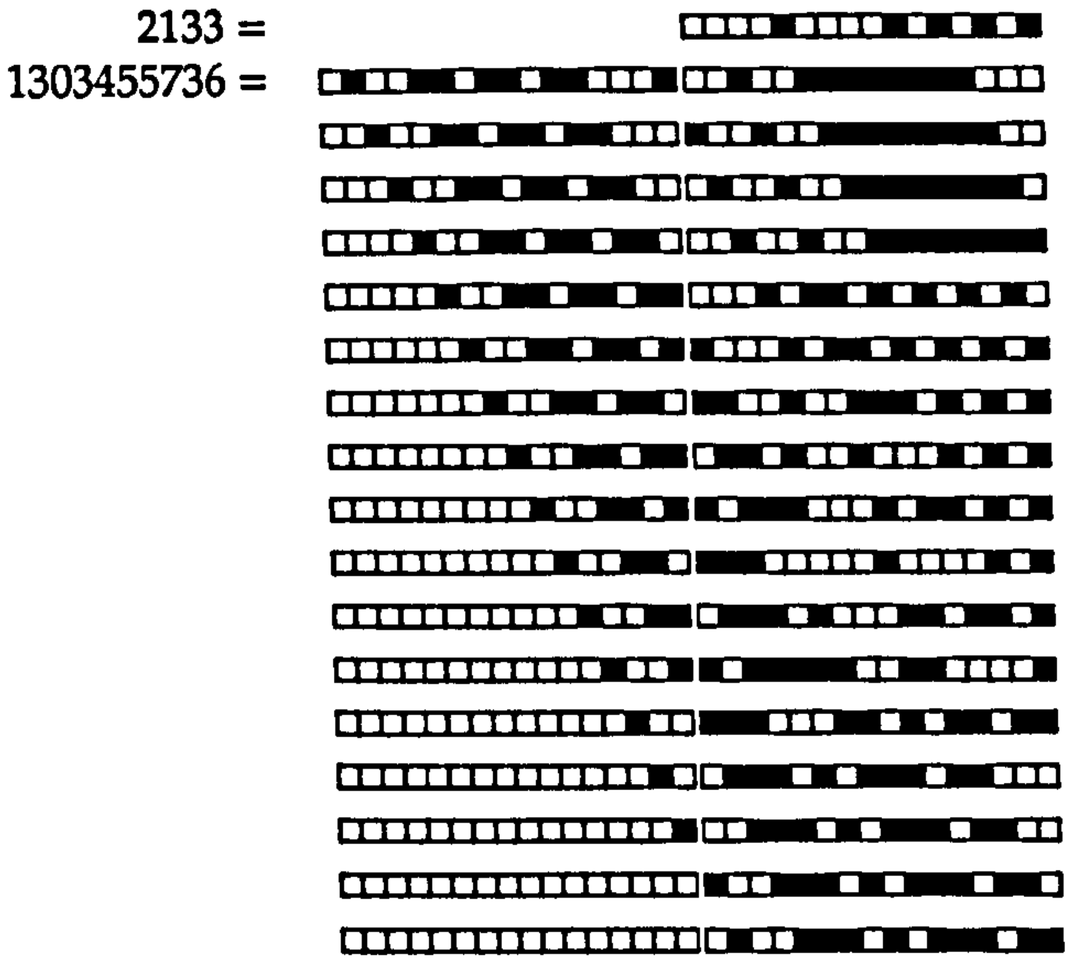


Figure 2.1: Computation of $1303455736/2^{16} \bmod 2133 = 20155$.

to find some u such that

$$z + um \equiv 0 \pmod{R}.$$

Solving for u, we get

$$u = (-m^{-1}) \cdot z \pmod{R}.$$

This suggests that we should pre-compute $-m^{-1} \bmod R$; then the modular division z/R can be exchanged for the computation of $z + um$ and then shifting the result n bits to the right. Algorithm 3 describes Montgomery reduction as suggested by the previous analysis.

Algorithm 3 Montgomery reduction
Input: n-word integer m, $-m^{-1} \bmod R$ where $R = b^n$, and $z < mR$.
Output: $zR^{-1} \bmod m$.

1: $u \leftarrow (-m^{-1})z \bmod R$
2: $x \leftarrow (z + um)/R$
3: **if** $x \geq m$ **then**
4: $\quad x \leftarrow x - m$.
5: **end if**
6: Return x

Note that computing $u = (-m^{-1}) \cdot z \bmod R$ and the product um requires multi-precision multiplication, but it turns out that we can modify the Montgomery reduction method to work on word-size integers as follows: If $R = b^n$ then we can divide z by $R = b^n$ through n divisions by b.

Here again, for the modular division by b, we compute u such that $z + um \equiv 0 \pmod{b}$ i.e.

$$\begin{aligned} u &= -(m \bmod b)^{-1} \cdot (z \bmod b) \bmod b \\ &= -m_0^{-1} \cdot z_0 \bmod b \end{aligned}$$

which requires word-sized operations only, see Algorithm 4. This variant is commonly *fused* with multiplication to produce what is known as *Interleaved Montgomery multiplication* [KAK96], described in the next subsection.

Algorithm 4 Montgomery reduction (word-level)
Input: $R = b^n$, $\hat{m} = -m^{-1} \bmod b$ and $Z = (zR \bmod m) < mR$ as an n-word integer.
Output: $ZR^{-1} \bmod m$.

1: $z \leftarrow Z$
2: **for** $i = 0, \ldots, n-1$ **do**
3: $\quad u \leftarrow z_i \hat{m} \bmod b$
4: $\quad z \leftarrow z + umb^i$ *(Multiplication and division by b correspond to shifts)*
5: **end for**
6: $z \leftarrow z/b^n$
7: **if** $z \geq m$ **then**
8: $\quad z \leftarrow z - m$
9: **end if**
10: Return z

Montgomery multiplication

Montgomery multiplication aims to achieve fast multiplication and reduction in one go. That is, given $X = xR \bmod m$ and $Y = yR \bmod m$ as n-word integers, $R = b^n$ and $\hat{m} = -m^{-1} \bmod b$, we want to compute the Montgomery product of X and Y which is given by $xyR \equiv XYR^{-1} \pmod{m}$.

Algorithm 5 presents the efficient *interleaved* Montgomery multiplication where multiplication and division by $R = b^n$ are interleaved and performed at the word level. This approach keeps the memory costs minimal and makes implementation easier and more efficient.

We will develop better versions of the Montgomery and Barrett reductions, in chapter 4, to be used when the operand sizes are bigger than the currently deployed sizes (1024–4096 bits).

Algorithm 5 Interleaved Montgomery multiplication
Input: $X = xR \bmod m$ and $Y = yR \bmod m$ as n-word integers, $R = b^n$ and $\hat{m} = -m^{-1} \bmod b$.
Output: $XYR^{-1} \bmod m$.

```
1: z ← 0
2: for i = 0,...,n − 1 do
3:     u ← (z_0 + X_i Y_0) m̂ mod b
4:     z ← (z + X_i Y + um)/b
5: end for
6: if z ≥ m then
7:     z ← z − m
8: end if
9: Return z
```

2.2 Quadratic residuosity and square roots in $\mathbb{Z}_p$

In chapter 6 we will need algorithms to test for quadratic residuosity and to compute square roots in $\mathbb{Z}_p$, so we will describe some suitable methods for these specific computations.

Let p be an odd prime. The equation $x^2 \equiv a \pmod{p}$, where a is a given integer, can have at most two roots in $\mathbb{Z}_p$. The element a is called a *quadratic residue* if the number of solutions to $x^2 \equiv a \pmod{p}$ is non-zero, and *quadratic non-residue* otherwise. This property is expressed by the Legendre symbol $\left(\frac{a}{p}\right)$, which is defined as follows.

$$\left(\frac{a}{p}\right) = \begin{cases} -1 & \text{if } a \text{ is a quadratic non-residue modulo } p, \\ 0 & \text{if } a = 0 \bmod p \\ +1 & \text{if } a \text{ is quadratic residue modulo } p. \end{cases}$$

The Legendre symbol $\left(\frac{a}{p}\right)$ is a multiplicative arithmetic function in a, and some of its properties that can help in computing it are

$$\left(\frac{a}{p}\right) = a^{(p-1)/2} \pmod{p}. \tag{2.1}$$

$$\left(\frac{q}{p}\right) = (-1)^{(p-1)(q-1)}\left(\frac{p}{q}\right) \quad \text{if } q \neq p \text{ is an odd prime.} \tag{2.2}$$

This latter property is due to Gauss and is known as the *quadratic reciprocity law.*

Computing the Legendre symbol allows us to decide if a number has a modular square root or not. To compute the actual square root granted its existence we can use the following methods.

We first treat an easy case which applies to half the odd primes. Suppose that $p \equiv 3 \pmod 4$. Then the modular square roots of a modulo p are given by

$$x \equiv \pm a^{(p+1)/4} \pmod p.$$

This can easily be checked as $(\pm a^{(p+1)/4})^2 = a^{(p+1)/2} = a \cdot \underbrace{a^{(p-1)/2}}_{=1} \equiv a \pmod p$, because $a^{(p-1)/2} \bmod p = \left(\frac{a}{p}\right) = +1$.

Another interesting special case occurs when $p \equiv 5 \pmod 8$. We first compute $s = a^{(p-5)/8}$, $u = a \cdot s$ and $t = s \cdot u$. Then it can be checked, in a similar way to the previous case, that the answer is u if $t = 1$ and $2^{(p-1)/4} \cdot u$ otherwise.

We can devise similar formulae for other more special cases, but they would give little advantage over the general probabilistic Tonelli-Shanks algorithm [Coh93, p. 32] which is described in Algorithm 6. Its expected running time is $O(\log^4 p)$.

Algorithm 6 Square root extraction modulo an odd prime p
Input: Odd prime p and $a \in \mathbb{Z}_p$ such that $\left(\frac{a}{p}\right) = +1$.
Output: x such that $x^2 \equiv a \pmod p$.

1: Find a random integer n such that $\left(\frac{n}{p}\right) = -1$
2: Write $p - 1 = 2^e q$ where q is odd
3: $y \leftarrow n^q \bmod p, r \leftarrow e$
4: $x \leftarrow a^{(q-1)/2} \bmod p, b \leftarrow ax^2 \bmod p, x \leftarrow ax \bmod p$
5: **while** $b \not\equiv 1 \pmod p$ **do**
6: Find the smallest $m \geq 1$ such that $b^{2^m} \equiv 1 \pmod p$
7: $t \leftarrow y^{2^{r-m-1}} \bmod p, y \leftarrow t^2 \bmod p, r \leftarrow m$
8: $x \leftarrow xt \bmod p, b \leftarrow by \bmod p$
9: **end while**
10: Return x

2.3 Elliptic curves

We have already introduced elliptic curves in the introductory chapter (§1.2.2). Here, we will comment on some aspects of curve representation, coordinate systems

that are usually used in practical implementation and point counting. This material will be used in chapter 6 where we improve the cost of the reduction DLP ≤ DHP by tuning the coordinate system together with some other parameters.

2.3.1 Coordinate systems

The coordinate system that was used in the introductory chapter to represent points on elliptic curves is known as the *affine coordinate system*. Other commonly used coordinate systems which are mathematically more elegant, as they allow a natural representation of the point at infinity, are called *projective* coordinate systems and are used in Projective Geometry to describe these curves more naturally. There are many other coordinate systems each with its computational advantages and disadvantages. In fact, there is a whole dedicated database for them called "Explicit-Formulas Database" at `http://hyperelliptic.org/EFD/`.

We will now list some of the popular equivalent representations of an affine point (x, y) on an elliptic curve given by the Weierstrass-form $y^2 = x^3 + ax + b$, over a prime field of characteristic $p > 3$, together with the corresponding cost of an elliptic curve addition and doubling respectively.

Affine coordinates. This system was introduced in §1.2.2 (p. 10), so we only quote the cost of addition and doubling in these coordinates. These are respectively

$$I + 2M + S, \quad I + 2M + 2S,$$

where I, M, S respectively denote the inversion, multiplication and squaring operations in the base field over which the elliptic curve is defined.

Projective coordinates. Points are represented as a triple $(X : Y : Z)$ satisfying the equation $Y^2Z = X^3 + aXZ^2 + bZ^3$, where the equivalence $(X : Y : Z) \equiv (sX : sY : sZ)$ holds for all nonzero s. A point $(X : Y : Z)$ corresponds to the affine point $(X/Z : Y/Z)$ when $Z \neq 0$ and to $O = (0 : 1 : 0)$ otherwise. The negative of $(X : Y : Z)$ is given by $(X : -Y : Z)$. The costs of an add and double operations are respectively

$$12M + 2S, \quad 7M + 5S.$$

Jacobian coordinates. This is a *weighted* projective coordinate system where the point $(X : Y : Z)$ satisfies $Y^2 = X^3 + aXZ^4 + bZ^6$ and $(X : Y : Z) \equiv (s^2X : s^3Y : sZ)$ for all nonzero s. A point $(X : Y : Z)$ corresponds to the affine point $(X/Z^2 : Y/Z^3)$ when $Z \neq 0$ and to $O = (1 : 1 : 0)$ otherwise. The negative of $(X : Y : Z)$ is given by $(X : -Y : Z)$.

The costs of an add and double operations are respectively

$$12M + 4S, \quad 4M + 6S.$$

More complicated methods of optimising the cost of elliptic curve arithmetic involve using mixed coordinate systems, where a set of different coordinates are used to try and decrease the total cost of an exponentiation for example. We will not pursue their description here, but the interested reader may refer to [CFA+06, Chapter 13].

2.3.2 Point counting and construction of elliptic curves

In this section we sketch the main methods used for these tasks. The description of these methods is lengthy and beyond the scope of this thesis. For more details on the mentioned algorithm see the relevant chapters in [BSS99, BSS04, CFA+06].

Point counting. Given an elliptic curve E over a finite field $\mathbf{F}_q$, the task of computing the order of the elliptic curve group $E(\mathbf{F}_q)$ is commonly referred to as *point counting*. Recall that, by the Hasse Theorem, the group order of $E(\mathbf{F}_q)$ is given by $|E(\mathbf{F}_q)| = q + 1 - t$ where $|t| \leq 2\sqrt{q}$. For elliptic curves over large-prime fields $\mathbf{F}_p$, one should use the $O(\log^6 p)$ Schoof-Elkies-Atkin's Algorithm (SEA), which is an improvement of the original $O(\log^8 p)$ algorithm suggested by Schoof in 1985 [Sch85]. The SEA algorithm is an ℓ-adic method, meaning that the order is first computed modulo different small primes ℓ_i such that their product is greater than the group order. This then can be reconstructed using the Chinese Remainder Theorem (CRT).

For fields with a small characteristic, there are faster methods which are p-adic in nature, such as Satoh's algorithm and the Arithmetic-Geometric-Mean (AGM) algorithms.

Construction of elliptic curves. Another task that is of importance in ECC is to build an elliptic curve group over a finite field $\mathbb{F}_q$ with a prescribed size n or with some specific properties. Of the few available options, we are interested in the following methods.

1. If we want to generate an elliptic curve with a given fixed size then we can use the *complex multiplication* technique for the construction. This method takes a fundamental discriminant $-D$ and constructs an elliptic curve over the given field which has complex multiplication by the maximal order of $\mathbb{Q}(\sqrt{-D})$. Note however that the running time depends exponentially on the class number h_D which grows like $O(\sqrt{D})$, so D should be as small as possible for the method to be efficient.

2. If the group order is only required to satisfy some easily testable property that holds with a non-negligible probability over the choices of the curve or field parameters, then randomly generating elliptic curves over $\mathbb{F}_q$ and counting their points until one is found with the desired order will yield a Las Vegas algorithm with an expected polynomial running time.

2.4 Exponentiation

Exponentiation is a time consuming operation that will be needed in chapters 4 and 6. This section introduces the techniques that we will be using and gives the conditions under which they may be suitable.

Naively, a general exponentiation g^e can be done with a cost of $e-1$ multiplications by computing $g^e = g \cdot g \cdot \cdots \cdot g$ (e factors) but, given that the exponents that are in common use in cryptography have sizes that certainly exceed 2^{80}, we need to use faster methods to make any exponentiation efficiently computable in practice. Primarily, we need to reduce the total number of multiplications needed for this

task, and we may also need to use faster multiplication algorithms such as the ones introduced in chapter 3.

When studying the problem of raising a group element g to a power e, two special cases arise depending on whether one of g or e is fixed or not.

1. The case where the exponent is always the same, as in the case of RSA where any message m to be encrypted to some party is always raised to the same power e.

2. The case where the element to be raised to a power is always the same, as is the case with the Diffie-Hellman key agreement scheme where a fixed group generator g (defined by some standard) is raised to many different powers.

In the exponentiation methods that we will introduce next, we can save on the cost by making the appropriate precomputations beforehand and once-for-all according to which case of the above we are in.

2.4.1 Binary and k-ary exponentiation algorithms

If we write the binary expansion of the exponent $e = \sum_{i=0}^{n} e_i 2^i$, with $e_i \in \{0, 1\}$, in its Horner's form:

$$e = ((\cdots((e_n \cdot 2) + e_{n-1}) \cdot 2 + e_{n-2}) \cdot 2 + \cdots + e_1) \cdot 2 + e_0$$

then we see that we can compute g^e in the following fashion

$$g^e = ((\cdots((g^{e_n})^2 \cdot g^{e_{n-1}})^2 \cdot g^{e_{n-2}})^2 \cdots)^2 \cdot g^{e_0}.$$

This form amounts to evaluating the successive terms of the sequence

$$g^{e_n}, g^{2e_n+e_{n-1}}, g^{2^2e_n+2e_{n-1}+e_{n-2}} \ldots, g^{2^n e_n+\cdots+e_0} = g^e,$$

which costs n squarings and at most n multiplications. This gives an upper bound of $O(\log e)$ modular multiplications on the cost of *modular* exponentiation, making it a problem in $\mathcal{P}$. Note that the average cost of this method is $\frac{3}{2} \lg e$ multiplications

because the average hamming weight of an integer is 1/2, so only half of the bits e_i are set to 1 giving rise to about $\frac{1}{2}\lg e$ multiplications on the top of the $\lg e$ squarings.

This method is usually referred to as the left-to-right binary exponentiation. There is a similar method called right-to-left binary exponentiation where the bits e_i are used starting from e_0 until e_n.

Fixed window method (k-ary method)

In this method, which is a generalisation of the previous, we first precompute a set of small powers of g. This then allows us to divide the exponent into chunks (windows) of size k bits, and then the exponentiation effort will be mainly k squarings and only one multiplication per k bits.

Fix a window size k. If we write the exponent e in the base 2^k as $\sum_{i=0}^{n} e_i(2^k)^i$ where $e_i \in \{0, 1, \ldots, 2^k - 1\}$ then we see that

$$g^e = \prod_{i=0}^{n} (g^{e_i})^{(2^k)^i}$$

or written slightly differently and more concretely (similar to the binary exponentiation)

$$g^e = ((\cdots((g^{e_n})^{2^k} \cdot g^{e_{n-1}})^{2^k} \cdot g^{e_{n-2}})^{2^k} \cdots)^{2^k} \cdot g^{e_0}.$$

Thus we have the method shown in Algorithm 7.

Algorithm 7 Fixed-window exponentiation (Left-to-right k-ary method)
Input: Group element g and $e = (e_{n-1} \ldots e_0)_{2^k}$ where $k \geq 1$.
Output: g^e.

1: $g_0 \leftarrow 1$
2: **for** $i = 1, \ldots, 2^{k-1} - 1$ **do**
3: $\quad g_i \leftarrow g \cdot g_{i-1}$ $\langle g_i = g^i \rangle$
4: **end for**
5: $A \leftarrow 1$
6: **for** $i = n - 1, \ldots, 0$ **do**
7: $\quad A \leftarrow A^{2^k}$ ⟨*square k times*⟩
8: $\quad A \leftarrow A \cdot g_{e_i}$
9: **end for**
10: **return** A

The cost of this approach is $2^{k-1}-1$ multiplications for the precomputation plus n multiplications and kn squarings, making the total number of multiplications equal to (assuming that squarings cost the same as general multiplications)

$$2^{k-1} + (k+1)n - 1.$$

Now, if the bit length of e is fixed and equal to d then $n = \lceil d/k \rceil$. Then one can find the best value for k by minimising the the number of multiplications

$$2^{k-1} + (k+1)\left\lceil \frac{d}{k} \right\rceil - 1.$$

If g is fixed the we can do the precomputation of small powers of g beforehand and reuse it for any subsequent exponentiation with respect to the same base g.

A generalisation of this method that halves the number of precomputed values and speeds it up a bit is known as the *sliding window* exponentiation method and is given in section 4.2 on page 75.

2.5 Pseudo-random number generation

In chapter 5, we will need a method to generate a "random lattice." We recapitulate on some possible techniques to generate pseudo-random sequence of elements which may be used as entries to the lattice basis matrix. For more in-depth treatment of this topic see the first chapter of [Knu98].

What we want is to quickly generate pseudo-random sequences of integers which should be cryptographically secure. The usual deterministic method of producing a sequence of pseudo-random numbers is to take a truly random seed value x_0 and then iterate some function on it so to "extract" more randomness from it. Some of the popular pseudo-random number generators (PRNGs) in the literature are

- **Linear Congruential Generators (LCG).** This is a classical PRNG that is both lightweight and very fast. The sequence's element are computed via the

iteration

$$x_+ = ax + b \pmod{m},$$

where m is a fixed modulus and a, b are carefully chosen constants to try and avoid statistical bias.

- **Pollard type generators.** These look like the famous iteration used in the Pollard ρ factoring method:

$$x_+ = x^2 + 2 \pmod{p},$$

for some large prime p.

In particular, the Blum-Blum-Shub (BBS) generator is a cryptographically secure pseudorandom *bit generation* generator (CSPRBG) under the assumption that integer factorisation is intractable. It uses the iteration

$$x_+ = x^2 \pmod{m},$$

where m is a product of two large primes each congruent to 3 mod 4, and outputs the least significant bit of x in each iteration.

- **Modular inversion generators.** For a large prime p, the iteration for these is similar to

$$x_+ = x^{-1} + c \pmod{p}.$$

There are many more flavours and exotic approaches to design PRNGs, which range from simply using shift registers to using elliptic curves, but since our needs are modest we will be satisfied with the methods that we have just described. In particular, the Pollard type generators are known to be cryptographically strong and were experimentally observed to yield good lattices for our hash function construction in chapter 5.

2.6 The GMP and NTL libraries

In our implementation of the adapted modular reduction methods presented in chapter 4, we used the GMP library [Gra07a]. The GNU Multi-Precision (GMP) arithmetic library is a portable library written in C that implements arbitrary precision arithmetic on integer, rational, and floating-point numbers. It is generally regarded to be the the most efficient such library. GMP is highly optimised and is designed to give a good performance for both small and large operand sizes. This achieved through the use of appropriate algorithms for the different operand sizes and by carefully implementing them while keeping any overheads at a minimum. The base operations are written in assembly for a wide range of platforms[1] while the rest of the library is written in portable C. The official website for the GMP library is `http://swox.com/gmp`, from where the latest version of the library can be downloaded in source code form. There are also three related mailing lists (Release announcements, general questions and discussions about usage of the GMP library, and bug reports), `http://swox.com/mailman/listinfo`.

The NTL library (Number Theory Library) [Sho06] is a high-performance library written in C++ which is developed and maintained by V. Shoup (`http://www.shoup.net/ntl`). This library is an extra layer on the top of GMP and provides a useful range of number theoretic functions. It was used in the development process of the hash function LASH presented in chapter 5. We used some of the functions related to lattices and linear algebra, especially the well tuned implementation of the LLL and BKZ (Block Korkin-Zolotarev) [SE91] lattice basis reduction algorithms, to test the lattices associated to LASH for weaknesses.

[1]The list of platforms include: ARM, DEC Alpha 21064, 21164, and 21264, AMD 29000, AMD K6, K6-2, Athlon, and Athlon64, Hitachi SuperH and SH-2, HPPA 1.0, 1.1 and 2.0, Intel Pentium, Pentium Pro/II/III, Pentium 4, generic x86, Intel IA-64, i960, Motorola MC68000, MC68020, MC88100, and MC88110, Motorola/IBM PowerPC 32 and 64, National NS32000, IBM POWER, MIPS R3000, R4000, SPARCv7, SuperSPARC, generic SPARCv8, UltraSPARC, DEC VAX, and Zilog Z8000. Some optimisations also for Cray vector systems, Clipper, IBM ROMP (RT), and Pyramid AP/XP.

Chapter 3

Integer arithmetic

"Many people regard arithmetic as a trivial thing that children learn and computers do, but we will see that arithmetic is a fascinating topic with many interesting facets."
— Donald E. Knuth,

In this chapter we study asymptotically faster integer multiplication algorithms. We will also study the computation of truncated products (short products) where we only compute a portion of the full product. Furthermore, we describe the "wooping" error-detection technique, which we shall use later in Chapter 4 to *correct* errors due to our faulty short product method used with the Montgomery reduction.

We assume that we have a machine that can do arithmetic operations on *word* sized operands, which we will refer to as *base operations*, and that it has access to an unlimited *random access memory*. The first assumption is true for most modern machines whereas, strictly speaking, the second is not true; as memory is always limited in practice and there is some cost associated with fetching or moving data – a cost that depends on the size and location of the data and also on the speed, size and architecture of the RAM and cache. However, if enough care is taken then a good implementation should be able to bring this extra cost to a minimum. Also, in order to simplify the task of analysing algorithms, we will limit ourselves to the study of *sequential* machines and do not consider any aspect of parallelism.

We represent large integers as arrays of machine words, with the basic arithmetic operations done with the usual classical schoolbook methods, unless otherwise

mentioned. A cost expression of the form $x\mathcal{M} + y\mathcal{A}$ denotes the cost of performing x base multiplications and y base additions. In order to make comparison feasible, we introduce a parameter μ such that $1\mathcal{M} = \mu\mathcal{A}$. This parameter depends on the machine's architecture and implementation details. To keep our notation light, we will omit the unit $\mathcal{A}$ in formulae of the form $a\mathcal{M}+b\mathcal{A} = (a\mu+b)\mathcal{A}$ and would simply write $a\mu + b$.

Let us now estimate the cost of schoolbook addition and multiplication in our model. We have $\mathcal{A}(n) = n$ for the cost of adding two n-word integers, and $\mathcal{M}(n) = n^2\mathcal{M} + 2n(n-1)\mathcal{A}$ for the cost of multiplying two n-word integers, i.e. we have

$$\mathcal{A}(n) = n \qquad \text{and} \qquad \mathcal{M}(n) = (\mu + 2)n^2 - 2n. \tag{3.1}$$

$\mathcal{M}_u(n)$ and $\mathcal{M}_\ell(n)$ will denote the cost of computing the *upper* and *lower* halves of the product of two n-word integers, respectively. The cost of computing the lower half product is $\mathcal{M}_\ell(n) = \frac{1}{2}n(n+1)\mathcal{M} + n(n-1)\mathcal{A}$, so

$$\mathcal{M}_\ell(n) = (\frac{\mu}{2} + 1)n^2 + (\frac{\mu}{2} - 1)n. \tag{3.2}$$

In principle, we have $\mathcal{M}_u(n) = \mathcal{M}_\ell(n)$ but there is a small extra cost due to the fact that we need to keep track of carries from the lower half of the product, a fact which will be crucial in our work in Chapter 4 on RSA with very large operands. We also set R to be the least power of the basis that is greater than n-words i.e. if a word holds ω bits then the basis is

$$b = 2^\omega \qquad \text{and} \qquad R = b^n = 2^{\omega n}. \tag{3.3}$$

Then, the subscripts ℓ and u respectively denote the lower and upper half parts of a number in the sense that

$$x_\ell = x \bmod R \qquad \text{and} \qquad x_u = \lfloor x/R \rfloor. \tag{3.4}$$

We will assume that the word size is $\omega = 32$ bits, which is the standard word-size in most present computers. This means that when dealing with 15,360-bit RSA modular arithmetic in Chapter 4 we will need $n = 480$ words. If the word size is 64 bits then n drops to 240.

3.1 Asymptotically faster multiplication algorithms

The next two subsections will review the Karatsuba and Toom-Cook fast integer multiplication algorithms and analyse their cost according to the model presented at the beginning of this chapter. A more comprehensive treatment of these and other methods can be found in [Knu98, p. 294–311]. We will also consider the computation of upper and lower halves of products [Har05, Har07], as these will save us on the overall cost of the reduction algorithms considered in Chapter 4.

Recall that, according to our computational cost model, we will not take the cost of memory operations into account and we will assume that they are for free.

3.1.1 The Karatsuba integer multiplication

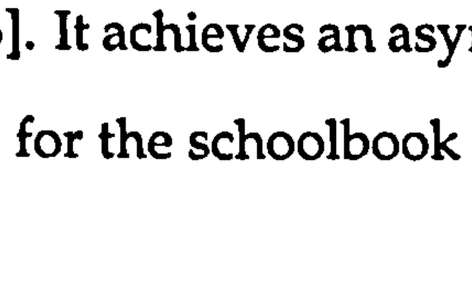
This is a popular divide-and-conquer algorithm for faster multiplication introduced by Karastuba and published by Ofman [OK63]. It achieves an asymptotic complexity of $O(n^{\lg 3}) = O(n^{1.585})$, as opposed to $O(n^2)$ for the schoolbook method (classical multiplication).

Let $u, v \in \mathbb{N}$ be represented as n-word integers in base b, where $n = 2t$. Write $u = u_1 b^t + u_0$ and $v = v_1 b^t + v_0$, where u_0, u_1, v_0, v_1 are t-word integers. Then

$$uv = w_2 b^{2t} + w_1 b^t + w_0,$$

where

$$\begin{aligned} w_2 &= u_1 v_1 \\ w_1 &= (u_0 + u_1)(v_0 + v_1) - w_0 - w_2 \\ w_0 &= u_0 v_0 \end{aligned}$$



In practice, computing $u_0 + u_1$ and $v_0 + v_1$ may result in an overflow, so extra care has to be taken when computing these values. Alternatively, one can compute

$$w_1 = w_0 + w_2 - (u_0 - u_1)(v_0 - v_1),$$

which uses subtraction instead of addition and hence avoids overflows, this however necessitates dealing with signed operands.

If we use the Karatsuba method recursively to multiply operands greater than or equal to a fixed threshold value T and switch to schoolbook multiplication thereafter then the cost function can be written as

$$\mathcal{K}(n) = \begin{cases} 3\mathcal{K}(n/2) + 4n & \text{for } n \geq T \\ M(n) & \text{for } n < T \end{cases} \tag{3.5}$$

Applying the general solution of such recurrence equations which we worked out in the introductory chapter (§1.3.1 on page 16) to this equation we get (for $n \geq T$)

$$\mathcal{K}(n) = \underbrace{[(\mu+2)\frac{T}{2^{\lfloor \lg(T/n) \rfloor}} + 6]\left(\frac{3}{4}\right)^{\lfloor \lg(T/n) \rfloor} T}_{\text{Bounded by a constant } (\mu, T \text{ are fixed})} \cdot \left(\frac{n}{T}\right)^{\lg 3} - 8n = O(n^{\lg 3}). \tag{3.6}$$

The case where n is odd can be dealt with by letting $t = \lceil n/2 \rceil$, but it is more efficient to set $t = \lfloor n/2 \rfloor$ allowing u_1, v_1 to be (t+1)-word integers while keeping u_0, v_0 as t-word integers. The extra bits need to be treated explicitly but it is worth the hassle as it will save some running time. With this latter approach, the cost obeys the following extra recurrence equation when n is odd

$$K(n) = 2\mathcal{K}((n+1)/2) + K((n-1)/2) + 4n.$$

With this optimisation, it becomes very difficult to write down a closed form of the solution, if feasible to start with. So, we will be satisfied with a sample plot. The graph in Figure 3.1 shows the ratio $M(n)/\mathcal{K}(n)$ for $\mu = 1.2$ and $T = 23$, hence illustrating the savings that can be made by using the Karatsuba multiplication method instead of the schoolbook method.

The threshold values used in the graphs of this chapter ($T = 23$ and $T' = 133$ which will be introduced later) are those of the Pentium-4 machines (2.80GHz, 512KB cache, model 2) that were used for the timing experiments of Chapter 4 (Section 4.3), as estimated by GMP's tuning program tuneup. These can easily be estimated for other architectures using the same tuning procedure (see Section 2.6 for the GMP library).

Note, however, that the exact value of μ is hard to pin down because execution times depend on the ordering of instructions and data, which may lead to significant savings through pipelining. Luckily, it turns out that small variations in μ have little theoretical impact on the cost ratios considered here, as μ essentially only affects the leading coefficient which varies slowly as a function of μ. The value 1.2 for μ was experimentally chosen from a set of possible values in the range $(1, 1.5)$. These were obtained using loops to measure the average times for word operations on a few Pentium-4 computers with the same specifications mentioned previously, and then fitting the collected data to estimate the value of μ. Values for μ can also be estimated theoretically through the tables presented in [Gra07b].

3.1.2 Toom-Cook multiplication

This method also uses a divide-and-conquer strategy and can be considered as a generalisation of the Karatsuba method. The general framework here is to treat integers as polynomials and then exploit some properties of polynomials to speed up calculations.

We first write the two integers u, v that we want to multiply as two degree r polynomials $u(x), v(x)$ whose coefficients are the base b^t digits of u and v, for some fixed $t \in \mathbb{N}$. We then evaluate the polynomials at as many points as needed to uniquely define their product $w(x) = u(x)v(x)$ through interpolation, namely $2r + 1$ points. Now, multiplying the values of the two polynomials $u(x), v(x)$ at the chosen points, we get the values of the product $w(x)$ at the same points. Given these $2r + 1$ values, we can now recover $w(x)$ by interpolation; and to get the product of the original integers we simply evaluate $w(x)$ at the base b^t (release the carries). This yields a multiplication method having complexity $O(n^{\log(2r+1)/\log(r+1)})$. Note that

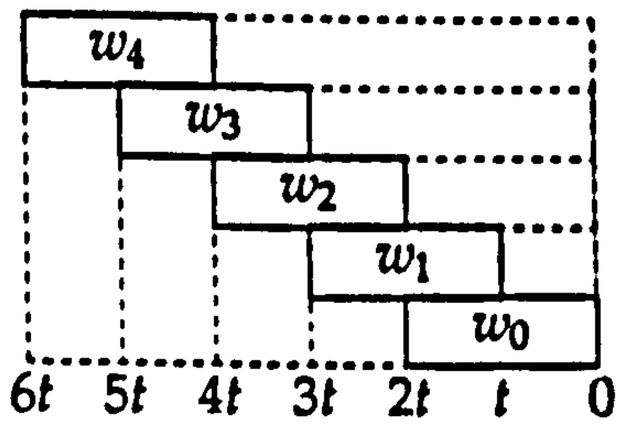
the Karatsuba method can viewed as a special case of this framework when $r = 1$ (linear polynomial).

We will describe a popular instance of this family of multiplication methods known as Toom-3 multiplication in more detail. Toom-3 achieves a complexity of $O(n^{\log_3 5}) = O(n^{1.465})$ by taking the polynomials $u(x)$ and $v(x)$ to be quadratic. Suppose we want to multiply two n-word integers u and v, where $n = 3t$. First, we represent them as quadratic polynomials evaluated at $x = b^t$

$$\begin{aligned} u &= u(x)|_{x=b^t} = u_0 + u_1 b^t + u_2 b^{2t}, \\ v &= v(x)|_{x=b^t} = v_0 + v_1 b^t + v_2 b^{2t}. \end{aligned}$$

Now, to evaluate $w = uv$, we first evaluate $w(x) = u(x)v(x)$ at $x = 0, 1, -1, 2, \infty$. Then, knowing the values of $w(x) = w_4 x^4 + w_3 x^3 + w_2 x^2 + w_1 x + w_0$ at five points, we interpolate the coefficients of w. We have

$$\begin{aligned} w_4 &= u_2 v_2, \\ w_3 &= u_2 v_1 + u_1 v_2, \\ w_2 &= u_2 v_0 + u_1 v_1 + u_0 v_2, \\ w_1 &= u_0 v_1 + u_1 v_0, \\ w_0 &= u_0 v_0. \end{aligned}$$



$$\begin{aligned} w(x)|_{x=0} &= u_0 v_0 &&= w_0, \\ w(x)|_{x=+1} &= (u_2 + u_1 + u_0)(v_2 + v_1 + v_0) &&=: \alpha, \\ w(x)|_{x=-1} &= (u_2 - u_1 + u_0)(v_2 - v_1 + v_0) &&=: \beta, \\ w(x)|_{x=2} &= (4u_2 + 2u_1 + u_0)(4v_2 + 2v_1 + v_0) &&=: \gamma, \\ w(x)|_{x=\infty} &:= \lim_{x\to\infty} u(x)v(x)/x^4 = u_2 v_2 &&= w_4. \end{aligned}$$

So we get w_0 and w_4 right away, and what remains is to find w_1, w_2, w_3. Solving the previous system of equations we get

$$\begin{aligned} w_2 &= (\alpha + \beta)/2 - w_4 - w_0 \\ w_3 &= +w_0/2 - 2w_4 + (\gamma - \beta)/6 - \alpha/2 \\ w_1 &= -w_0/2 + 2w_4 - (\gamma + 2\beta)/6 + \alpha \end{aligned}$$

Hence, the cost function for Toom-3 is $\mathcal{T}(n) = 5\mathcal{T}(n/3)+[3\mathcal{A}(t)+4\mathcal{A}(t)+3\mathcal{A}(t)]+4\mathcal{A}(t)$, i.e.

$$\mathcal{T}(n) = 5\mathcal{T}(n/3) + 14n/3.$$

When n is not a multiple of 3, we set $t = \lceil n/3 \rceil$ and allow u_2 and v_2 to be shorter than t words, as is done in the code of the GMP library – This makes implementation easier.

We introduce a second threshold value $T' > T$ such that if $n < T$ then we use schoolbook multiplication, if $T \leq n < T'$ then we use Karatsuba multiplication, and if $n > T'$ then we use Toom-3 multiplication recursively.

Figure 3.1 shows the plots of the ratios $\mathcal{M}(n)/\mathcal{K}(n)$ and $\mathcal{M}(n)/\mathcal{T}(n)$ for $\mu = 1.2$, $T = 23$ (as before) and $T' = 133$, hence showing the speedup that is made over the schoolbook multiplication method in this case.

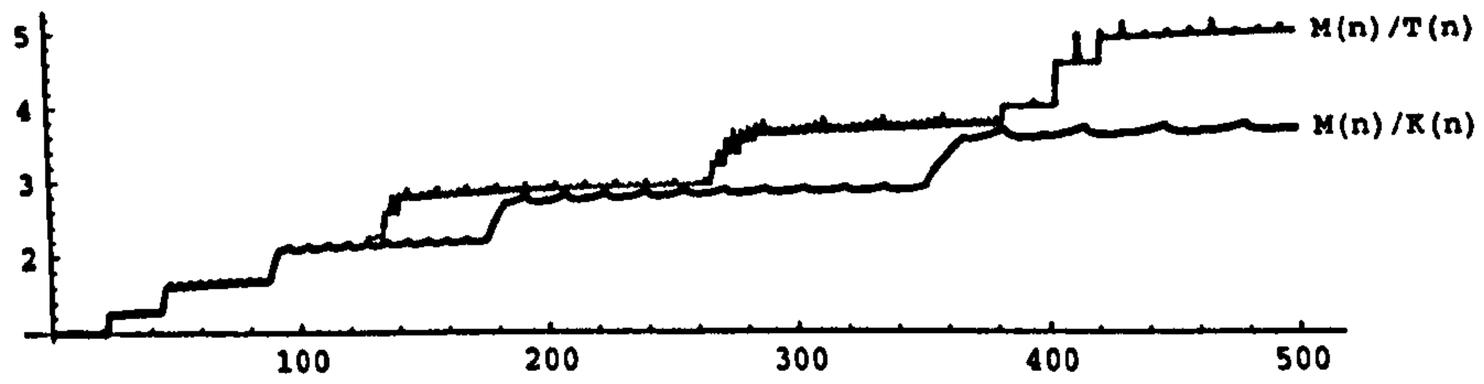


Figure 3.1: Plots for $\mathcal{M}(n)/\mathcal{K}(n)$ and $\mathcal{M}(n)/\mathcal{T}(n)$.

3.1.3 Fast Fourier Transforms (FFT) based multiplication

Suppose we want to multiply two multi-precision integers u and v of length n a power of 2. We first represent both operands as polynomials $u(x)$ and $v(x)$ evaluated at $x = b$

$$\begin{aligned} u &= u(x)|_{x=b} = u_0 + u_1 b + \cdots + u_{n-1}b^{n-1} = (u_0, \ldots, u_{n-1})_b, \\ v &= v(x)|_{x=b} = v_0 + v_1 b + \cdots + v_{n-1}b^{n-1} = (v_0, \ldots, v_{n-1})_b. \end{aligned}$$

Now, to evaluate uv, we first compute the *linear convolution* of $u(x)$ and $v(x)$ by FFT-transforming their coefficients vectors and point-multiplying the results. We then evaluate the FFT-inverse of the latter result at the base b to obtain the integer

product uv, i.e.

$$uv = \mathrm{FFT}_{2n}^{-1}(\mathrm{FFT}_{2n}(u) * \mathrm{FFT}_{2n}(v))|_{x=b},$$

where $*$ denotes point-wise multiplication of two vectors

$$(a_0, \ldots, a_k) * (b_0, \ldots, b_k) = (a_0 b_0, a_1 b_1, \ldots, a_k b_k).$$

This approach fits with the Toom-Cook paradigm of fast multiplication since FFT_{2n} and FFT_{2n}^{-1} correspond to evaluation and interpolation at the $2n$th roots of unity, respectively. We will next investigate the possibility of using FFTs to compute short products, then give an exact description of how to compute FFTs efficiently using complex arithmetic.

FFT-based short products

There does not seem to be there any way of computing half products using FFTs without computing the whole product. However, it is shown in [PG05] that a closely related result can be computed using what is called a *cyclic convolution*: Let $w = uv = (w_u \| w_\ell)_b$, where w_u and w_ℓ are the upper and lower halves of w in the b-base representation. The cyclic polynomial convolution of $u(x)$ and $v(x)$ is

$$\begin{aligned} w(x) \bmod x^n - 1 &= w_u(x)x^n + w_\ell(x) \bmod x^n - 1 \\ &= w_u(x) + w_\ell(x). \end{aligned}$$

This can be computed using FFT on n-point, instead of $2n$ points, as follows:

$$w_u(x) + w_\ell(x) = \mathrm{FFT}_n^{-1}(\mathrm{FFT}_n(w_u) * \mathrm{FFT}_n(w_\ell)).$$

The prior knowledge of either the upper or lower half of a product combined with this result will help us determine the other half. In fact, for our purposes, a good approximation to these halves will suffice. This fact will enable us to trade full FFTs for half-sized FFTs when only a half product is needed knowing that the

result has a special shape. The details can be found in the later discussion about FFT Montgomery and Barrett reduction (§7.2.2 on page 142) or alternatively see [PG05].

Computing FFTs using complex arithmetic

The more precise term used for Fourier Transforms for *vectors* is Discrete Fourier Transform (DFT), but it is common to use the general term FFT for it when there is no ambiguity in its usage.

Without loss of generality, assume that n is a power of 2, $n = 2^k$. We call $\omega_n = e^{2\pi i/n}$ the principal nth root of unity. Let $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ be a polynomial of order less than n. We identify the polynomial A with its coefficients vector $a = (a_0, a_1, \ldots, a_{n-1})$.

The Discrete Fourier Transform (DFT) of a polynomial A (represented by its vector of coefficients $(a_0, a_1, \ldots, a_{n-1})$) is defined by

$$\mathrm{DFT}_n(A) = (y_0, y_1, \ldots, y_{n-1}) \quad \text{where} \quad y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{jk}.$$

A recursive algorithm to compute FFTs. Algorithm 8 describes how to recursively compute compute the FFT of a vector a, [CLRS01, p. 788].

Algorithm 8 FFT of a vector a (Recursive Algorithm)
Input: A vector a of length n, a power of 2, and $\omega_n = e^{2\pi i/n}$.
Output: $\mathrm{FFT}_n(a)$

1: **if** $n = 1$ **then**
2: Return a
3: **end if**
4: $\omega \leftarrow 1$
5: $a^{[0]} \leftarrow (a_0, a_2, \ldots, a_{n-2})$ ⟨*Even indices*⟩
6: $a^{[1]} \leftarrow (a_1, a_3, \ldots, a_{n-1})$ ⟨*Odd indices*⟩
7: $y^{[0]} \leftarrow FFT_{n/2}(a^{[0]})$
8: $y^{[1]} \leftarrow FFT_{n/2}(a^{[1]})$
9: **for** $k = 0, \ldots, \frac{1}{2}n - 1$ **do**
10: $t \leftarrow \omega y_k^{[1]}$ ⟨$1\mathcal{M}$⟩
11: $y_k \leftarrow y_k^{[0]} + t$ ⟨$1\mathcal{A}$⟩
12: $y_{(n/2)+k} \leftarrow y_k^{[0]} - t$ ⟨$1\mathcal{A}$⟩
13: $\omega \leftarrow \omega\omega_n$ ⟨*Can be precomputed...*⟩
14: **end for**
15: Return y

This costs $T(n) = 2T(n/2) + (n/2)\mathcal{M} + n\mathcal{A}$. Rewriting this recurrence equation as

$$T(n) = 2T(n/2) + (\mu/2 + 1)n$$

and using the general solution from 1.3.1 (page 16) we get

$$T(n) = (\mu/2 + 1)n \lg n = \Theta(n \lg n).$$

An iterative algorithm to compute FFTs. This is described in Algorithm 9, [CLRS01, p. 794]. The cost is the same but implementation may be easier and more efficient with this approach.

Algorithm 9 FFT of a vector a (Iterative Algorithm)
Input: A vector a of length n, a power of 2, and $\omega_n = e^{2\pi i/n}$.
Output: $\mathrm{FFT}_n(a)$

```
    "Bit-reverse" copy a into A
1:  for k = 0,...,n-1 do
2:      A_{rev(k)} ← a_k                    ⟨rev(k) is the (lg n)-bit integer reverse of k⟩
3:  end for
    Compute the FFT iteratively
4:  for s = 1,...,lg n do
5:      m ← 2^s
6:      ω_m ← e^{2πi/m}
7:      for k = 0,...,n-1 by m do
8:          ω ← 1
9:          for j = 0,...,½m-1 do
10:             t ← ωA_{k+j+m/2}             ⟨1M⟩
11:             u ← A_{k+j}
12:             A_{k+j} ← u + t              ⟨1A⟩
13:             A_{k+j+m/2} ← u - t          ⟨1A⟩
14:             ω ← ωω_m                     ⟨Can be precomputed...⟩
15:         end for
16:     end for
17: end for
18: Return A
```

The cost of this algorithm is $(m/2\mathcal{M} + m\mathcal{A})(n/m)\lg n = (n/2)\lg n\mathcal{M} + n\lg n\mathcal{A}$, i.e.

$$(\mu/2 + 1)n \lg n.$$

Inverting the FFT (FFT^{-1}). For the inversion, we note that $\mathrm{DFT}_{\omega_n}^{-1} = \frac{1}{n}\mathrm{DFT}_{(\omega_n^{-1})}$. So the cost of the inversion is the same as the cost of a DFT plus a division by $n = 2^k$, which is a simple shift operation.

3.2 Short products

We will make use of methods for computing the lower and upper half products (short products), so we will study their costs next. We start with a general method that applies to all multiplication algorithms [Mul97, Har05] then present some specific solutions specific to the Karatsuba method.

3.2.1 A general method

First, we will introduce a visual aid that will make explaining this method easier and more intuitive. When multiplying two numbers using schoolbook multiplication we stack the partial products in a shape similar to the one on the left in Figure 3.2 prior to adding them up; and to find the lower half product, for example, we only need to compute the results in the shaded triangle.

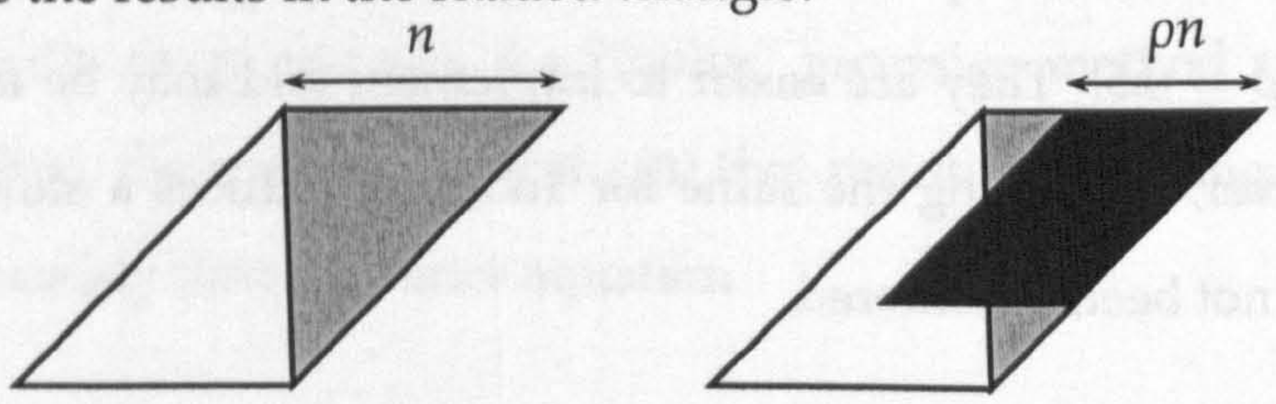


Figure 3.2: Calculation of short products.

Let $S(n)$ be the cost of computing a short product of two n-word integers. If we take a portion ρn, where $0.5 \leq \rho < 1$, of both operands and compute their full product, corresponding to the darker shaded area on the right in Figure 3.2, and then compute the remaining terms using short products again, corresponding to the two light shaded triangles, then we find that this method would cost

$$S(n) = M(\rho n) + 2S((1-\rho)n).$$

Since the multiplication methods we are considering, except FFT-based multiplication, all cost $M(n) = O(n^\alpha)$ for some $\alpha \in (1,2]$ we find that[1]

$$S(n) \leq \underbrace{\frac{\rho^\alpha}{1-2(1-\rho)^\alpha}}_{C_\rho} M(n).$$

The factor C_ρ in the inequality $S(n) \leq C_\rho M(n)$ is minimal at $\rho = 1 - 2^{-1/(\alpha-1)}$, and the following table summarises the results for the methods that we are interested in. It should be noted that these are the best asymptotically, and as such there may be better choices for ρ when n is small or moderate.

Method	α	ρ	C_ρ
Schoolbook	2	0.5	0.5
Karatsuba	$\lg 3$	0.694	0.808
Toom-3	$\log_3 5$	0.775	0.888

Note that if we fix n and look for the best value of ρ we may get a slightly different value. For the case where $n = 480$, the value of ρ turns out to be about 0.80 for Karatsuba and 0.88 for Toom-3.

The next Karatsuba-specific methods are actually special cases of this general setup with $\rho = 0.5$. They are easier to implement and may be faster in practice. Note, however, that doing the same for Toom-3 produces a slower method and hence it has not been considered.

3.2.2 Lower half products using the Karatsuba method

Recall that we have taken $u = u_0 + b^t u_1$ and $v = v_0 + b^t v_1$ to be $n = 2t$ words long integers, and set $w = uv = w_0 + w_1 b^t + w_2 b^{2t}$ where $w_0 = u_0 v_0$, $w_1 = u_0 v_1 + u_1 v_0$ and $w_2 = u_1 v_1$.

[1]First, note that $M(n) = O(n^\alpha)$ implies that $M(an) = a^\alpha M(n)$ for any $a \in \mathbb{R}$. So we have

$$\begin{aligned} S(n) &= M(\rho n) + 2S((1-\rho)n) \\ &= M(\rho n) + 2M(\rho(1-\rho)n) + 2^2 M(\rho(1-\rho)^2 n) + 2^3(\ldots) \\ &= (\rho^\alpha + 2\rho^\alpha(1-\rho)^\alpha + 2^2\rho^\alpha(1-\rho)^{2\alpha} + \cdots)M(n) \\ &\leq \frac{\rho^\alpha}{1-2(1-\rho)^\alpha} M(n). \end{aligned}$$

For the lower product $(uv)_\ell$, we now see that we need to compute

$$w_\ell = (w_0 + w_1 b^t + w_2 b^{2t}) \bmod b^{2t} = (u_0 v_0 + [(u_0 v_1 + u_1 v_0) \bmod b^t] b^t) \bmod b^{2t}$$
$$= (u_0 v_0 + [(u_0 v_1)_\ell + (u_1 v_0)_\ell] b^t)_\ell,$$

which costs $\mathcal{K}_\ell(n) = \mathcal{K}(t) + 2\mathcal{K}_\ell(t) + 2\mathcal{A}(t)$, i.e.

$$\mathcal{K}_\ell(n) = \mathcal{K}(n/2) + 2\mathcal{K}_\ell(n/2) + n. \tag{3.7}$$

3.2.3 Upper half products using the Karatsuba method

This time, we have to compute

$$w_u = \left\lfloor \frac{w_2 b^{2t} + w_1 b^t + w_0}{b^{2t}} \right\rfloor = u_1 v_1 + \left\lfloor \frac{u_0 v_1 + u_1 v_0}{b^t} \right\rfloor + carry$$
$$\approx u_1 v_1 + (u_0 v_1)_u + (u_1 v_0)_u.$$

The carry results from adding w_0 to $w_1 b^t$, in the full multiplication, and hence we have that $carry \in \{0, 1\}$.

If we ignore the carry and use the "faulty" recursive method suggested by this formula. Then, the maximum error $\epsilon(n)$ that results from using this method recursively will satisfy the recurrence equation

$$\epsilon(n) = 2\epsilon(n/2) + 2 \quad \text{and} \quad \epsilon(n) = 0 \text{ for } n < T.$$

By the result of section 1.3.1 (page 16) we deduce that

$$\epsilon(n) = 2 \cdot (2^{\lceil \lg(n/T) \rceil} - 1) \leq \frac{4}{T} n - 2.$$

So, computing upper-half products, up-to an error of order $O(n)$, can be done at the cost of $\mathcal{K}_u(n) = \mathcal{K}(t) + 2\mathcal{K}_u(t) + 2\mathcal{A}(t)$, i.e.

$$\mathcal{K}_u(n) = \mathcal{K}(n/2) + 2\mathcal{K}_u(n/2) + n = \mathcal{K}_\ell(n). \tag{3.8}$$

It turns out that, when the faulty result of this method is used in the reduction algorithms, we can correct the computation by using a nice technique, known as *wooping*, which is due to Bos [FS03, p. 281–284]. This idea is explained in the next section 3.3, whereas the correction steps as applied in our modification of the Montgomery reduction are detailed in section 4.1.1 (page 71).

To see how much faster these methods are, we plot $\mathcal{M}_\ell(n)/\mathcal{K}_\ell(n)$ (using both the general and the specific method) and $\mathcal{M}_\ell(n)/\mathcal{T}_\ell(n)$ – see Figure 3.3. The same speed-ups apply to the upper-half product methods too as they essentially have the same cost.

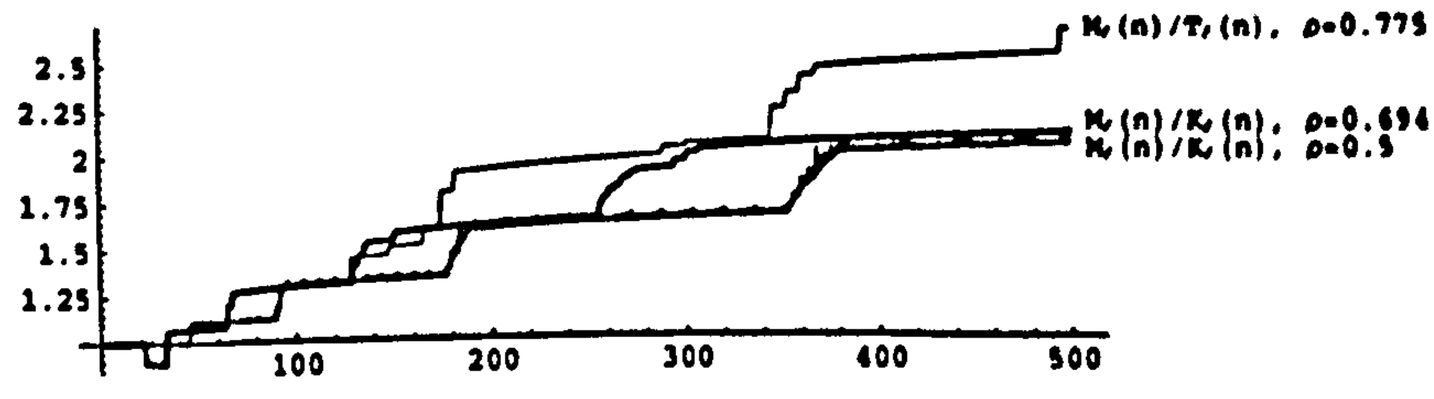


Figure 3.3: Plots for $\mathcal{M}_\ell(n)/\mathcal{K}_\ell(n)$ and $\mathcal{M}_\ell(n)/\mathcal{T}_\ell(n)$.

3.3 Wooping

The wooping technique allows us to *verify* the outcome of a set of integer operations via a clever probabilistic test. The idea as introduced by Bos and explained in [FS03, p. 281–284] relies on the fact that if p is a prime number and we are given the result of an integer operation together with the inputs, then we can detect any modification of the result, with probability $1 - \frac{1}{p}$, by reducing the given full result modulo p and comparing it with the recalculated answer modulo p.

For example, if the operation is $z \leftarrow x \cdot y$ then we randomly choose a small prime p and compute $\tilde{x} \leftarrow x \bmod p$ and $\tilde{y} \leftarrow y \bmod p$ first. Next, we compute $\tilde{z} \leftarrow \tilde{x} \cdot \tilde{y} \bmod p$; and for the comparison we reduce z modulo p and compare the result with $\tilde{z}$. If the two reduced results do not agree then there certainly is an error in the full integer computation, assuming the "small" calculation modulo p is correct, but if they agree then there is a low chance that an error has occurred.

That is to say, the general idea is to perform the same operations modulo a *small* prime number and then compare the results. More specifically, we reduce the operands modulo the prime number first then operate on them with the corresponding modular operations. If p is a *randomly chosen prime number* then the probability that this check fails to reveal the error is $1/p$, so one can choose other prime numbers for the wooping test to increase confidence.

We will use this technique in a different and novel manner in Chapter 4, where we already know that there is a linearly bounded small error in our computation which *underestimates* upper half products (using our earlier approach in §3.2.3) – What we want is to *correct* this error. Furthermore, since we are not in an adversarial setup, this correction scheme will be *deterministic* and *always successful*. The solution, in our case, will be to choose a woop modulus that is bigger than the largest possible error, and then correct the integer computation by adding or subtracting the difference between the two reduced values (according to whether the faulty result is an underestimate or an overestimate of the correct result).

As a toy example of how to use the wooping technique for correction, let us consider a device that can multiply integers but sometimes overestimates it and introduces an error of +1 in the result. Suppose that we wanted to compute 4×5 but we got 21 as the answer. First, note that we can choose the woop modulus to be 2 as that is enough to reveal the magnitude of the error. Now, we check that $(4 \bmod 2) \times (5 \bmod 2) = 0 \times 1 = 0$ whereas $21 \bmod 2 = 1$, so we correct the computation by subtracting 1 from 21 to get the correct answer of 20. For the exact details of how to use this technique in our work, see section 4.1.1 on page 71.

On a side note, as an alternative to wooping one may consider computing enough extra words to the right of the truncated upper-product in order to ensure a small probability of a carry being missed. This is in fact suggested in [Har05] and the extra words are referred to as "guard digits." This alternative is more complicated to implement because of the extra storage and will most likely be more expensive, especially if two or more guard digits are needed. Wooping on the other hand requires negligible storage and introduces little computational overhead, especially

if the wooping modulus is chosen to be special in order to speed up the modular reduction.

Chapter 4

Efficient RSA at high security parameters

"The choice is particularly difficult for paranoid organizations whose encrypted messages should remain secret for several decades, since it is almost impossible to predict the progress of factoring algorithms over such a long period of time. The only reasonable course of action is to use huge margins of safety, but this will make the RSA operations extremely slow."
— A. Shamir

NIST has recently recommended using RSA moduli sizes as big as 15360 bits to match the security level of AES-256 [Nat06, p. 63]. With this in mind, it is now worthwhile to explore the improvements that can be made over [Koc94] by using asymptotically faster multiplication methods together with any "tricks" that may render them practical even for moderate operand sizes (4096–8192 bits for example).

We show how the *wooping* technique, described in §3.3, will allow us to overcome the difficulties that arise when trying to go beyond the obvious simple substitution of multiplication methods. These difficulties are due to carry-propagation when computing upper-half products with recursive methods, a problem that does not arise when using traditional combinations such as the Karatsuba-Comba-Montgomery (KCM) method [GAST05, Sco96].

Using the formal computational cost model and material presented in chapters 2 and 3, we estimate the exact cost of the Montgomery and Barrett modular reduction algorithms. We then introduce two variants using the Karatsuba and Toom-3

multiplication methods, described in section 3.1, and analyse the savings that can be theoretically achieved. These variants have been implemented in C using the GMP library (see §2.6), and the relevant results are reported here and compared with the theoretical estimates.

4.1 The Montgomery and Barrett reductions

Given a fixed n-word modulus m, we want to reduce $2n$-word integers modulo m as fast as possible. We will now describe two improved fast reduction algorithms, based on the Montgomery and Barrett methods, using multiplication methods that are faster than the schoolbook method and their adaptations to compute short products.

4.1.1 Montgomery reduction

Let us first recall the general Montgomery reduction algorithm as described in Algorithm 3 on page 41.

Input: n-word integer m, $\hat{m} = -m^{-1} \bmod R$ where $R = b^n$, and $z < mR$.
Output: $zR^{-1} \bmod m$.

1: $u \leftarrow \hat{m}z \bmod R$ $\quad \langle \mathcal{M}_\ell(n) \rangle$
2: $x \leftarrow (z + um)/R$ $\quad \langle \mathcal{M}_u(n) + \mathcal{A}(n) \rangle$
3: **if** $x \geq m$ **then**
4: $\quad x \leftarrow x - m.$ $\quad \langle \mathcal{A}(n) \rangle$
5: **end if**
6: Return x

Note that in Step 1, it is sufficient to compute a lower half product; and in Step 2 we can compute an upper half product of u and m then add the result to z_u plus a carry. Hence, the cost of this algorithm is $\mathcal{M}_\ell(n) + \mathcal{M}_u(n) + 2n$. So, its cost using schoolbook multiplication is (using $\mathcal{M}_\ell = \mathcal{M}_u$)

$$C_{mr,cl}(n) = (\mu + 2)n^2 + \mu n. \tag{4.1}$$

Here, we have used C with subscript mr,cl to indicate cost of *Montgomery reduction* using *classical multiplication* (schoolbook multiplication method). Later, we

will use *mm* for *Montgomery multiplication*, *br* for *Barrett reduction*, 2 for *Karatsuba multiplication* (Splitting operands into two halves) and 3 for *Toom-3 multiplication*.

The practical word-level version of Montgomery reduction (Algorithm 4) is based on schoolbook multiplication and does not require direct calculation of lower or upper half products, but the quoted cost remains the same.

The Karatsuba variant with wooping

Recall, from section 3.2, that we can compute upper-half products using Karatsuba multiplication with an error of $O(n)$. We will now explain how to use the wooping correction idea in our case. Let $\lambda \in \mathbb{N}$ be a modulus greater than the magnitude of the maximum possible error resulting from ignoring the carry in the "faulty" upper-half Karatsuba method.

We first compute the product $u \leftarrow (-m^{-1})z \bmod R = (\hat{m}z)_\ell$ with a low-half Karatsuba multiplication. Now, for $x \leftarrow (z+um)/R$, note that a good approximation to this value is given by $x_u + (um)_u$, which will be off by at most 1 (carry). An extra error will come from the fact that we are using a faulty Karatsuba multiplication for the upper-half product $(um)_u$. To correct the approximate answer, we now compute $(z + um)/R$ modulo λ and compare it with the reduction of the approximate value: Given that the error magnitude is less than λ then we will be able to deduce the offset from the correct answer by comparing these reduced values, and therefore correct our answer. This is the "trick" that allows us to be satisfied with an approximation to $(um)_u$ and save on its computation.

If we further choose $\lambda = b^l - 1$, for some $l \in \mathbb{N}$, then reduction modulo λ becomes rather efficient (see § 2.1.1, page 38). In fact, to reduce an n-word number modulo $b^l - 1$, we only need about $\lceil n/l \rceil$ additions on numbers of size l words, costing a total of $n\mathcal{A}$. In practice, for $b = 2^{32}$, we take $l = 1$ as this is enough to correct errors for operand sizes $n < b$. Also, note that with this choice of λ and $R = b^n$ we have $R \equiv 1 \pmod{\lambda}$, so the computation of $(z + um)/R \bmod \lambda$ requires no inversion.

With this choice of $\lambda = b - 1$ and $R = b^n$, the correction steps involve computing $z + um \bmod \lambda$, costing about $(2n + n + n)\mathcal{A} + 1\mathcal{M} + 2\mathcal{A}$, and $x \bmod \lambda$, costing about $n\mathcal{A}$, where x is the result of step 2 of the algorithm. Then, computing the offset

and correction will cost $2\mathcal{A}$. So the cost of the Karatsuba variant of Montgomery reduction is about

$$C_{mr,2}(n) = \mathcal{K}_\ell(n) + \mathcal{K}_u(n) + 7n + \mu + 4. \tag{4.2}$$

Algorithm 10 gives a detailed description of our method. We use subscript λ instead of writing "modλ" to lighten notation, e.g. $z_\lambda = z \bmod \lambda$. We also use the notation $(um)_{\sim u}$ to indicate that we are using our faulty upper-half product method to approximate $(um)_u$.

Algorithm 10 Montgomery reduction with wooping
Input: n-word modulus m, $\hat{m} = -m^{-1} \bmod R$ where $R = b^n$, $z < mR$, and $\lambda = b - 1$.
Output: $zR^{-1} \bmod m$.

1: $u \leftarrow (\hat{m}z)_\ell$	$\langle \mathcal{K}_\ell(n) \rangle$
2: $x \leftarrow z_u + (um)_{\sim u}$	$\langle \mathcal{K}_u(n) + \mathcal{A}(n) \rangle$
3: $c \leftarrow (z_\lambda + u_\lambda \cdot m_\lambda)_\lambda$	
$x \leftarrow x - (c - x_\lambda)$	$\langle$*Correction (Wooping)*$\rangle$
4: **If** $x \geq m$ **then**	
5: $x \leftarrow x - m$.	$\langle \mathcal{A}(n) \rangle$
6: **end if**	
7: Return x	

The Toom-3 variant (with wooping)

We proceed exactly the same as in the Karatsuba variant (Algorithm 10) but using the Toom-3 multiplication methods. Then, the cost is found to be

$$C_{mr,3}(n) = \mathcal{T}_\ell(n) + \mathcal{T}_u(n) + 7n + \mu + 4. \tag{4.3}$$

Comparison. Figure 4.1 shows the graphs of $C_{mr,cl}/C_{mr,2}$ and $C_{mr,cl}/C_{mr,3}$, and serves to illustrate the improvements that can be made with these two variants of Montgomery reduction.

Crossing point. From the graph we see that the crossing point is at about 40 words, so we expect the Karatsuba variant of Montgomery reduction to start being effective from moduli sizes of about 1280 bits.

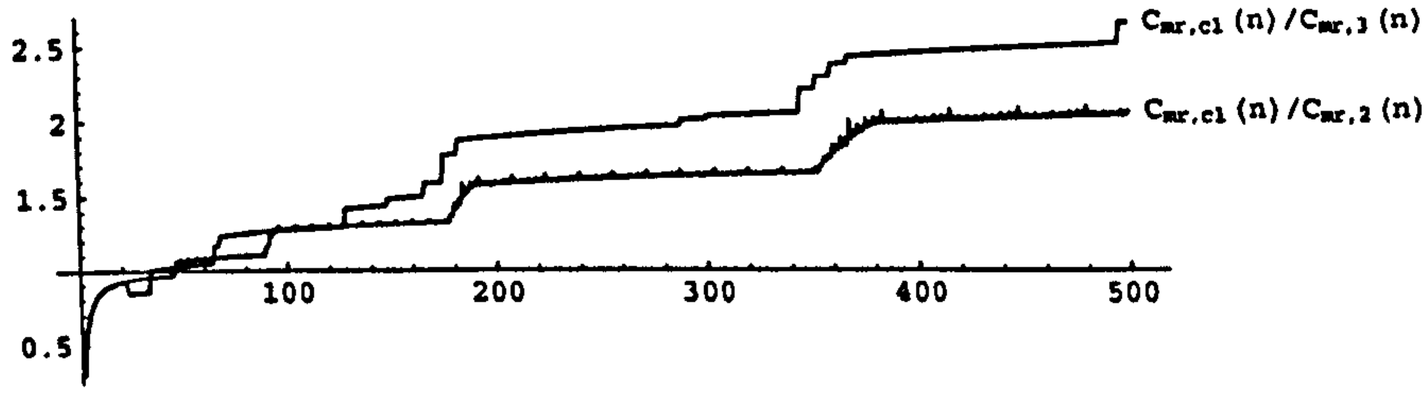


Figure 4.1: Plots for $C_{mr,cl}/C_{mr,2}$ and $C_{mr,cl}/C_{mr,3}$.

4.1.2 Montgomery multiplication

Montgomery multiplication aims to achieve fast multiplication and reduction in one go. There exists an efficient *interleaved* version where multiplications and division by R are interleaved and performed word-by-word as described in Algorithm 5 (quoted below, see page 43). This approach keeps the memory costs minimal and makes implementation easier.

There does not seem to be an easy way in which this can be done with the faster multiplication methods because of their recursive nature. For future research, we propose using an iterative version of Karatsuba in Section 7.2.3 (page 145). We leave this as an open problem.

Let us now analyse Algorithm 5 and find its computational cost.

Input: n-word integers $X, Y \in \mathbb{Z}_m$, $R = b^n$, and $\hat{m} = -m^{-1} \bmod b$.
Output: $XYR^{-1} \bmod m$.

1: $z \leftarrow 0$
2: **for** $i = 0, \ldots, n-1$ **do**
3: $\quad u \leftarrow (z_0 + X_i Y_0)\hat{m} \bmod b$ $\qquad \langle 2\mathcal{M} + 1\mathcal{A} \rangle$
4: $\quad z \leftarrow (z + X_i Y + um)/b$ $\qquad \langle 2\mathcal{M}_1(n) + 2\mathcal{A}(n) \rangle$
5: **end for**
6: **if** $z \geq m$ **then**
7: $\quad z \leftarrow z - m$ $\qquad \langle \mathcal{A}(n) \rangle$
8: **end if**
9: Return z

Let $\mathcal{M}_1(n)$ denote the cost of multiplying an n-word integer by a single word integer. Then we find that

$$\mathcal{M}_1(n) = n\mathcal{M} + (n-1)\mathcal{A} = (\mu + 1)n - 1.$$

So, the cost of the interleaved Montgomery multiplication is $n[2\mathcal{M} + 1\mathcal{A} + 2(n\mathcal{M} + (n-1)\mathcal{A})] + \mathcal{A}(n)$, i.e.

$$C_{mm,cl}(n) = 2(\mu + 1)n^2 + 2\mu n. \tag{4.4}$$

The Karatsuba variant (with wooping)

To compute the Montgomery multiplication of X and Y: $XYR^{-1} \bmod m$, we first multiply X by Y using the Karatsuba method then we Montgomery-reduce the result as described in Section 4.1.1. Montgomery multiplication using Karatsuba will therefore cost us

$$C_{mm,2}(n) = \mathcal{K}(n) + C_{mr,2}(n). \tag{4.5}$$

The Toom-3 variant

Here we also proceed exactly the same as in the Karatsuba variant. The cost this time is found to be

$$C_{mm,3}(n) = \mathcal{T}(n) + C_{mr,3}(n). \tag{4.6}$$

Comparison. Figure 4.2 shows the plots of $C_{mm,cl}/C_{mm,2}$ and $C_{mm,cl}/C_{mm,3}$, which illustrate the gain that is theoretically achievable with these variants of Montgomery multiplication.

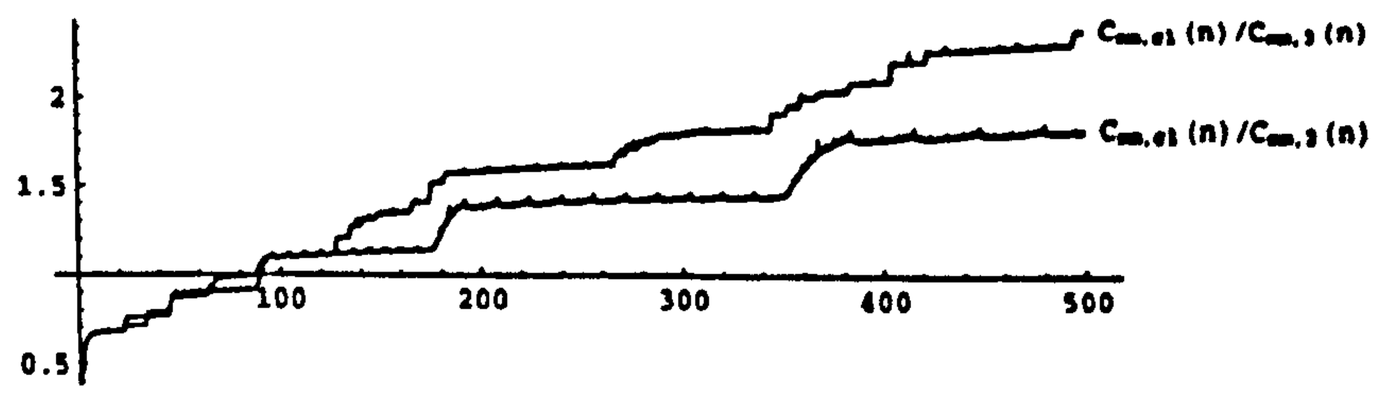


Figure 4.2: Plots for $C_{mm,cl}/C_{mm,2}$ and $C_{mm,cl}/C_{mm,3}$.

Crossing point. From the previous graph we find that the crossing point is at about 90 words, implying that 2880 bits is the point from which our variant starts to be advantageous.

4.1.3 Barrett reduction

Recall Algorithm 2 (page 40) which describes the Barrett reduction method.

Input: n-word modulus m, $\mu = \lfloor b^{2n}/m \rfloor$ and $z < m^2$.
Output: $z \bmod m$.

1: $z' \leftarrow \lfloor z/b^{n-1} \rfloor$, $\tilde{q} \leftarrow \lfloor z'\mu/b^{n+1} \rfloor$ $\langle \mathcal{M}_u(n) \rangle$
2: $r \leftarrow (z \bmod b^{n+1}) - (\tilde{q}m \bmod b^{n+1})$ $\langle \mathcal{M}_\ell(n) + \mathcal{A}(n) \rangle$
3: **if** $r < 0$ **then**
4: $\quad r \leftarrow r + b^{n+1}$ $\langle 1\mathcal{A} \rangle$
5: **end if**
6: **while** $r \geq m$ **do**
7: $\quad r \leftarrow r - m$ $\langle$*Repeated at most twice:* $2\mathcal{A}(n)\rangle$
8: **end while**
9: Return r

From this description, we see that the cost of this algorithm is at most $\mathcal{M}_u(n) + \mathcal{M}_\ell(n) + 3n + 1$. So if schoolbook multiplication is used then this reduction method will cost

$$C_{br,cl}(n) = (\mu + 2)n^2 + (\mu + 1)n + 1, \tag{4.7}$$

and if Karatsuba multiplication is used it will cost

$$C_{br,2}(n) = \mathcal{K}(n) + \mathcal{K}_\ell(n) + 3n + 1 \tag{4.8}$$

and similarly for Toom-3 we get

$$C_{br,3}(n) = \mathcal{T}(n) + \mathcal{T}(n) + 3n + 1 \tag{4.9}$$

Comparison. Figure 4.3 represents $C_{br,cl}/C_{br,2}$ and $C_{br,cl}/C_{br,3}$, and again we can see from it that the cutoff point is at about 90 words.

4.2 Exponentiation using the sliding-window method

The RSA operation consists of a modular exponentiation modulo a large RSA-modulus. In our implementation of the methods developed in this chapter we used full size exponents and therefore needed a fast exponentiation method. We used the *sliding window method* which we now describe.

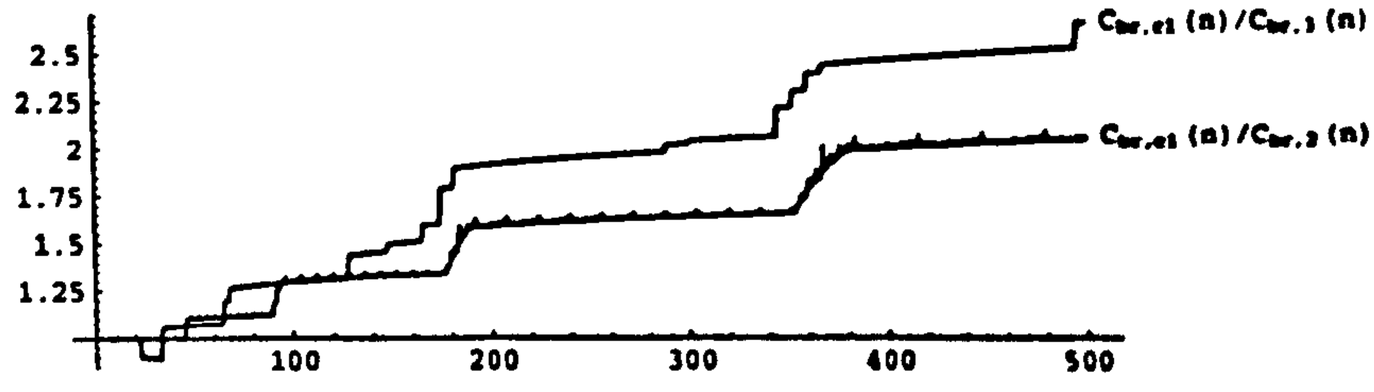


Figure 4.3: Plots for $C_{br,c1}/C_{br,2}$ and $C_{br,c1}/C_{br,3}$.

The sliding window exponentiation method is a generalisation of the fixed window method (k-ary method), where the window is allowed to slide so that the number it represents is odd. This flexibility halves the number of precomputed values and reduces the average number of multiplications (but the number of squarings stays the same). Algorithm 11 gives a concise description of this generalisation.

Algorithm 11 Sliding-window exponentiation
Input: Group element g and integer $e = (e_n e_{n-1} \dots e_0)_{2^k}$ where $k \geq 1$.
Output: g^e.

1: $g_1 \leftarrow g, g_2 \leftarrow g^2$
2: **for** $i = 1, \dots, 2^{k-1} - 1$ **do**
3: $\quad g_{2i+1} \leftarrow g_{2i-1} g_2$
4: **end for**
5: $A \leftarrow 1, i \leftarrow n$
6: **while** $i \geq 0$ **do**
7: $\quad$ **if** $e_i = 0$ **then**
8: $\quad\quad A \leftarrow A^2, i \leftarrow i - 1$
9: $\quad$ **else**
10: $\quad\quad$ Find the longest bit-string $e_i \dots e_\ell$ such that $i - \ell + 1 \leq k$ and $e_\ell = 1$
11: $\quad\quad A \leftarrow A^{2^{i-\ell+1}} g_{(e_i \dots e_\ell)_2}, i \leftarrow \ell - 1$
12: $\quad$ **end if**
13: **end while**
14: **return** A

A careful analysis of this method for bit-size η done by H. Cohen in [Coh05] shows that there exists $\rho > 1$ such that this method requires

$$\eta - \frac{k^2 + k + 2}{2(k+1)} + O(\rho^{-\eta}) \text{ squarings and } \frac{\eta}{k+1} - \frac{k(k+3)}{2(k+1)^2} + O(\rho^{-\eta}) \text{ multiplications.}$$

We note that GMP optimises the window size k depending on the exponent's bit-size η by finding the least k such that $2\eta > 2^k(k^2 + 3k + 2) = 2^k(k+1)(k+2)$. The

following table shows when a window of size k is first used by GMP for $n < 1000$ ($\eta = 32n$).

k	3	4	5	6	7	8	9	10
n	1	3	8	22	57	145	361	881

4.3 Experimental results

We implemented Montgomery Multiplication in three flavours: The classical interleaved version, the new Karatsuba and Toom-3 with wooping variants and, finally, a naive version where we first multiply using the fastest available multiplication method then Montgomery-reduce the resulting product using the efficient word-level version of Algorithm 3 (GMP's redc function). These were implemented in C using the GMP library [Gra07a] with the low-level `mpn` set of functions for speed (as they are SSE2-optimised). We also implemented the RSA exponentiation by adapting GMP's `mpz_powm` function which uses the efficient sliding window method for exponentiation [Gor98, Coh05].

The times needed to perform each of these two computations were averaged for random full size operands of sizes from 64 words (2,048 bits) up to 576 words (18,432 bits), in a step size of 32 words, and then plotted to ease comparison of the different methods. Figure 4.4 shows the timing results for Montgomery Multiplication, and Figure 4.5 summarises the average times obtained for RSA exponentiation using full size random messages and exponents (times are given in milliseconds).

We note that although the experimental cutoff points do not fit very accurately with the theory, because of the parallelism present in modern processors (pipelining), these are not far from the expected theoretical values, and the general trends are indeed as expected.

The experiments were done on Intel Pentium 4 machines (2.80GHz, 512KB cache, model 2). The threshold values that were used are $(T, T') = (23, 133)$ as estimated by GMP's tuning program tuneup. We bring the reader's attention to the fact that GMP uses slightly different threshold values for squaring, for which a more optimised code is used. (For our machines, they are 57 and 131 respectively, as estimated

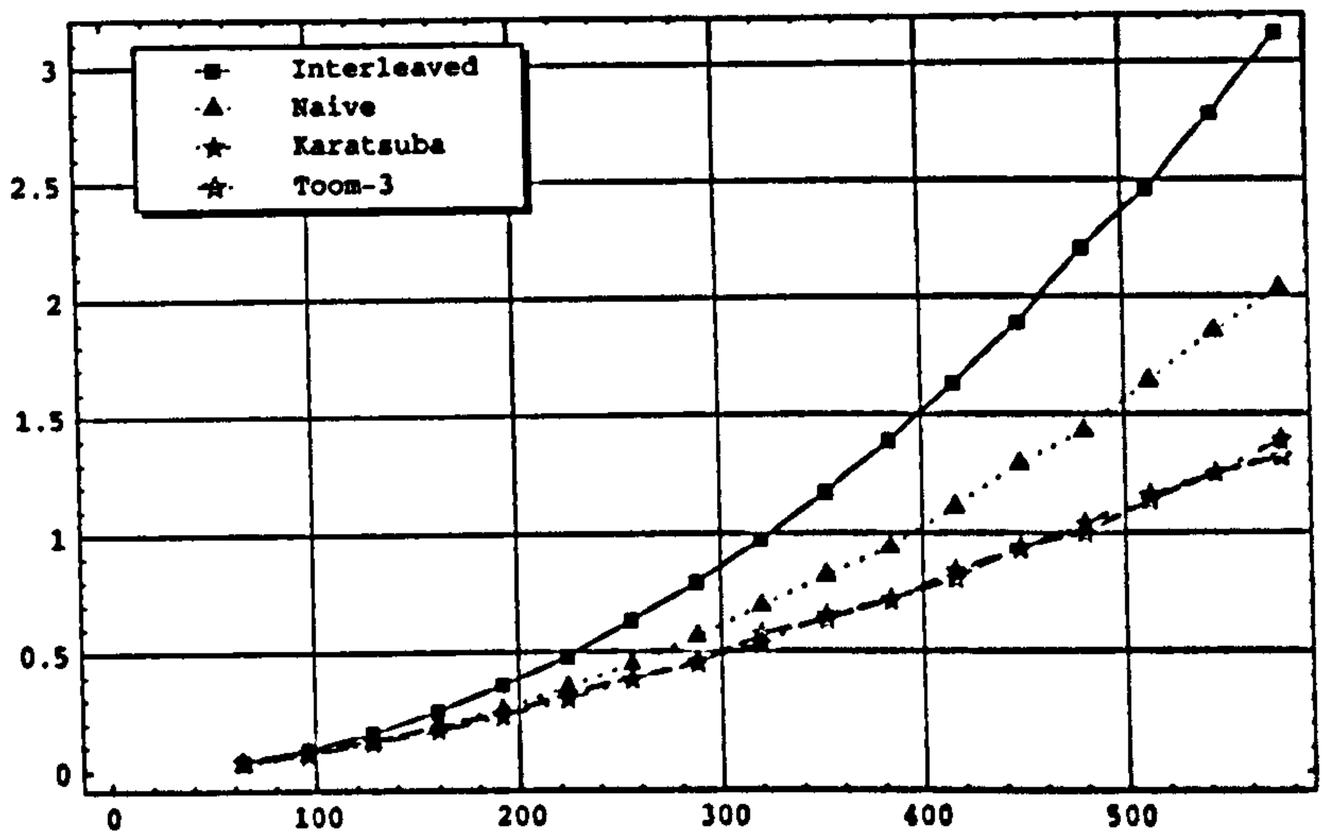


Figure 4.4: Montgomery Multiplication times in milliseconds.

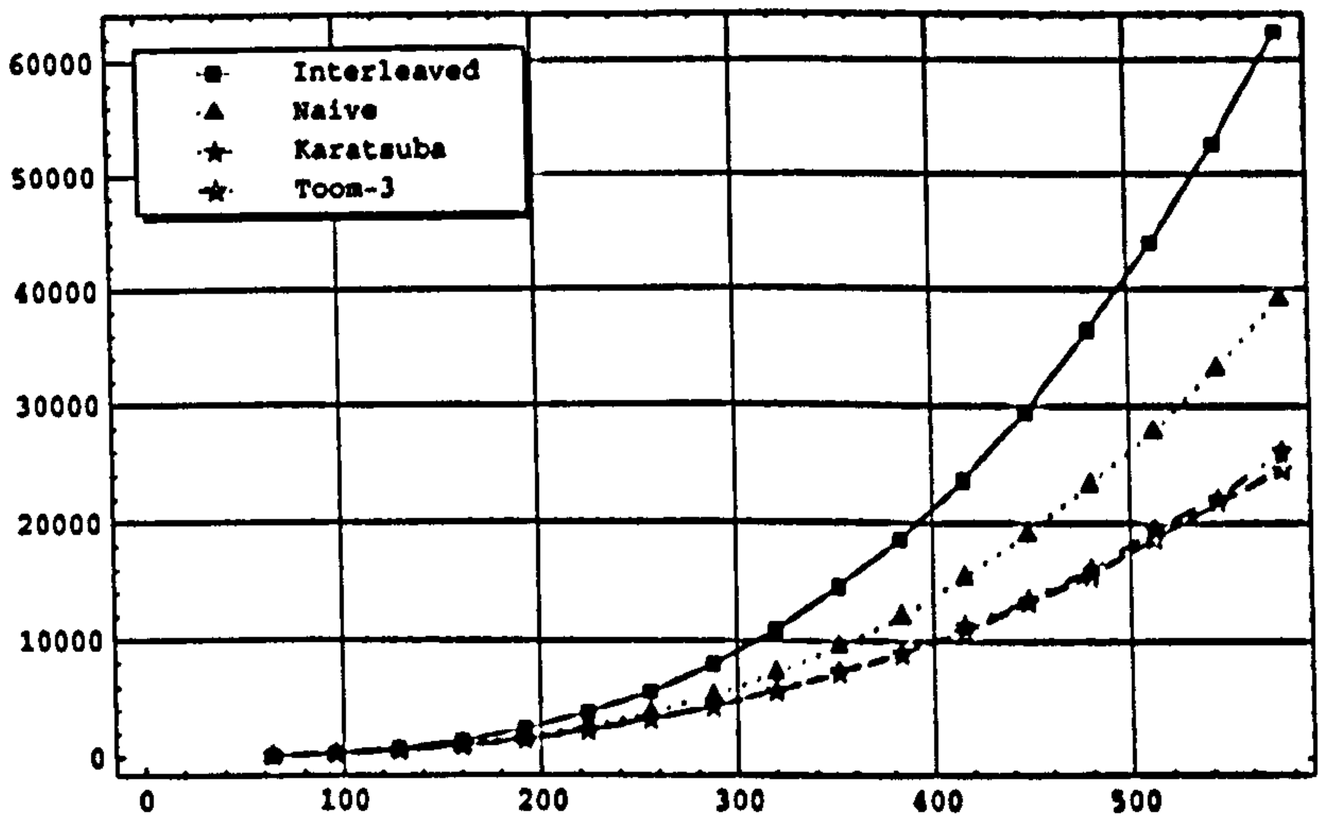


Figure 4.5: RSA exponentiation times in milliseconds.

by tuneup, but there is a large margin of error in them). Note also that, in our implementation of the short products algorithms, we used halves of T and T' for the thresholds.

In particular, we find that an execution of a 15,360-bit RSA exponentiation with full size exponent on these Pentium 4 machines takes 16.1 seconds with the Karatsuba variant and 15.6s with the Toom-3 variant on average, compared to about 23.3s with the naive version and 36.45s for the traditional interleaved Montgomery multiplication.

RSA in practice (Comment on the size of exponents). In practice, the exponents used in the RSA encryption operation are small or of a special form, e.g. $e = 3$ or $e = 2^{16} + 1$. We have produced graphs for the full size exponents case to reflect the very general case, but timings for these special exponents can easily be estimated from the timings of Montgomery multiplication (Figure 4.4), e.g. for $e = 3$ we will only need two multiplications[1] so it would cost about 2 milliseconds seconds only using our new variants. Likewise, for $e = 2^{16} + 1$, the computation will take about 17 milliseconds, which is of course much faster than the very general case of full size exponents.

For the RSA decryption operation, the Chinese Remainder Theorem is usually used to construct $m = c^d \bmod N$ from the simpler operation $c^d \bmod p$ and $c^d \bmod q$, so all operands will be of roughly half the size (240 words in our case).

In our treatment of this problem, we have concentrated on speeding up modular arithmetic. There are other ways of implementing RSA with large public key which we discuss in the conclusion chapter, see §7.2.1 (page 141).

[1] If the squaring operation is optimised then it costs one squaring and one multiplication.

Chapter 5

LASH, a lattice based hash

"Although there is no specific reason to believe that a practical attack on any of the SHA-2 family of hash functions is imminent, a successful collision attack on an algorithm in the SHA-2 family could have catastrophic effects for digital signatures."

— NIST, Federal Register Notice (November 2, 2007)

Hash functions play an important role in cryptography and constitute a sensitive component of many protocols. Traditionally, the used hash functions were picked from the MD family of hash functions, which includes the popular MD5, RIPEMD and SHA1 hash functions. But in light of the recent novel attacks on this family that were discovered by X. Wang in 2005 [WY05], the cryptographic community has been left with very limited choice of hash functions to be used. We are in great need for new design paradigms and better hash function families than the currently available.

These recently broken hash functions are essentially all derived from the same design principles and are built using somewhat ad-hoc techniques, albeit being constructed using solid symmetric cryptography techniques and expertise. In contrast to this practice, other areas of cryptography have replaced ad-hoc constructions with well defined sets of design principles. Examples include the wide-trail design strategy of AES [DR02, Chapter 9], or the rigorous application of reductionist provable security techniques as in the context of RSA-OAEP [BR94, FOPS01].

However, considering the recent attacks on the currently deployed hash functions, Provable Security has become a desirable property and a very important

aspect, if not essential, in the design of hash functions. A slower hash rate is not too much of an issue for many applications. Therefore, it is very important to closely study potential constructions, propose new ones and improve them when possible.

While the SHA2 family of hash functions is not yet known to succumb to the recent attack techniques, its design principles are so similar to SHA1 that we have no guarantee an attack will not appear in the near future. Furthermore, despite the fact that a lot is known theoretically about how to construct hash functions from one-way functions, these theoretical results do not aid one in designing *efficient and practical* realisations. Hence, there is an urgent and pressing need for new radical designs and constructions of families of practical hash functions. Also, considering the damage of the recent attacks, provable security has now become a very desirable, if not essential, property of any new hash function proposal. Speed may be the price to pay for this property, but this should not be too expensive and a successful design should keep the overhead as small as possible.

One problem with previous attempts to design an alternative family of hash functions based on hard computational problems, such as the RSA-like MASH-1 algorithm [ISO96] for instance, has been that the result is not competitive in terms of performance. However, the recent development of a *provably* collision resistant and a somewhat efficient hash function based on the hardness of factoring called VSH [CLS05] has ignited a renewed interest in devising more hash function families of this kind, which may be efficient enough to be used in practice as a replacement to the old ones.

VSH is faster than MASH-1, but it is still significantly slower than any standard hash function. The output block length is fixed to the size of an RSA-modulus, although of course this may be truncated in an actual application, and its design criteria mentions nothing about pre-image resistance or other desirable properties as its only proved property is collision-resistance. Additionally, the design of VSH raises the question as to whom actually generates the hard problem upon which the security is based, i.e. the prime factorisation of the RSA-modulus (We may need to put trust in some third party or parties to generate the secret in a secure multi-party computation).

In this chapter[1] we will explore one possible path to achieve this aim through a relaxation of a previous inefficient proposal. We modify and tune the construction proposed by Goldreich, Goldwasser and Halevi (GGH) in [GGH96], which is based on lattices, to obtain a fairly efficient compression function and a hash function that we call LASH. We will introduce a result by Ajtai [Ajt96], which is the complexity result behind the security of the GGH compression function, and then discuss the design and claimed properties. We show that it is unfortunately insecure for any *practical* instantiation, despite it being secure asymptotically. We then present our modified hash function, LASH, which is partly based on the Miyaguchi–Preneel construction [BRS02] as we replace block ciphers with a modular matrix multiplication of the kind used in the GGH construction. That is to say, LASH uses a relaxed version of the theoretical GGH construction as a core component in its compression function. With these choices, we show that with a suitable selection of parameters we can produce a hash function which is comparable in performance to existing deployed hash functions such as SHA2.

Before going any further, now is probably the right time to explain what the acronym LASH stands for. Actually, it has a number of possible meanings which all reflect the design principles and properties of this family of hash functions:

- **L**inear **A**lgebra based **S**ecure **H**ash: As the main component is simply a matrix-vector product.
- **L**A**t**tice based **S**ecure **H**ash: Because inverting/finding collisions in the linear component of the hash function is closely related to the hard problem of finding short/close vectors in lattices.
- **L**ight-weight **A**rithmetical **S**ecure **H**ash: Because the design is very short and easy to remember.

In this chapter we will repeatedly refer to two special types of vectors, and for convenience we will give them names for ease of reference. A *binary vector* in a

[1]The material presented in this chapter is joint work with D. Page, M.J.O. Saarinen, J.H. Silverman and N.P. Smart, [BPS+06]. My main contribution was in writing code (using the NTL library) to experiment with different ways of generating "hard random lattices" and trying to attack them using lattice reduction techniques, under the supervision of Smart. I also noticed the special form of the lattice basis matrix kernels which turned out to be the basis of an attack on an earlier version of LASH which directly influenced its current design. §5.3.4 is mostly the work of Silverman, §5.4 is mainly the work of Page and Saarinen. Adopting the Miyaguchi–Preneel construction was suggested by Saarinen. The rest of the chapter contains contributions from coauthors to different degrees.

lattice L is defined to be a vector in L whose coordinates are restricted to come from the set $\{0,1\}$. The set of all binary vectors in $\mathbf{R}^n$ will be denoted by $\mathcal{B}_n$. Similarly, we define a *ternary vector* to have coordinates from the set $\{-1,0,1\}$ and let $\mathcal{T}_n$ to denote the set of ternary vectors in $\mathbf{R}^n$.

5.1 The GGH lattice based hash function

Interest in cryptographic primitives based on lattice problems thrived after Ajtai published his seminal paper "Generating hard instances of lattice problems" [Ajt96] in 1996, in which he showed that some variants of the knapsack problem are at least as hard to break on the *average* as the *worst case* instances of a corresponding lattice approximation problem.

In slightly more detail, if we let c be an arbitrary positive constant then assuming that there is no efficient algorithm to approximate SVP in an n-dimensional lattice to within a multiplicative factor n^c in the worst case, then Ajtai's result allows us to build a knapsack-like cryptographic one-way function that is as hard to break on average as to approximate SVP to within a polynomial factor in the worst case, provided that the key is chosen randomly.

In the same year (1996), Goldreich, Goldwasser and Halevi presented in [GGH96] a hash function whose collision resistance could be related to the *worst case* of the problem of approximating small vectors in lattices (SVP). It was shown, in the tradition of reductionist provable security, that any algorithm which could systematically find collisions for such a function can be used to solve the problem of approximating short vectors in an associated lattice to within a polynomial factor (apprSVP). The reduction to the worst case of this latter problem was established using the result of Ajtai [Ajt96].

The problem with the construction of a compression function using the ideas of Goldreich *et al.*is that, with the parameters needed so as to reduce the underlying lattice problem to the worst case scenario, the resulting hash function is not very efficient. In addition it appears hard to directly develop a hash function which meets a specific security guarantee required by the practical community. For example, if

the output hash size is n bits in length then it should require $2^{n/2}$ operations to find a collision. One can show (see later) that collisions can be found in the construction of Goldreich *et al.*using $2^{n/3}$ operations, or $2^{n/4}$ operations if one is using the GGH construction with the MD construction to extend the input domain.

5.1.1 The GGH compression function

Let $H \in \mathbb{Z}^{m\times n}$ be an $m \times n$ integer matrix, and let q be a fixed integer modulus (not necessarily prime). We define a lattice L_H and a map f_H by

$$L_H = \{x \in \mathbb{Z}^n : Hx = 0 \pmod q\} \tag{5.1}$$

and

$$\begin{array}{rccl} f_H: & \{0,1\}^n & \longrightarrow & (\mathbb{Z}/q\mathbb{Z})^m \\ & b & \longmapsto & Hb \pmod q \end{array}, \tag{5.2}$$

where bit-strings from $\{0,1\}^n$ are interpreted as binary vectors from $\mathcal{B}_n \subset \mathbb{Z}^n$.

The map f_H is taken to be the compression function in the hash function construction proposed by Goldreich, Goldwasser and Halevi [GGH96], and the lattice L_H is its associated lattice. Building on the work of Ajtai [Ajt96] they show that, for a suitably chosen $m \times n$ matrix H over $\mathbb{F}_q$, if the map f_H is collision resistant then it is hard to find *small non-zero ternary vectors* in the lattice L_H. More precisely, they show that if m, n, and $q = O(n^c)$, for some constant $c > 0$, satisfy

$$m \log_2 q < n < \frac{q}{2m^4} \tag{5.3}$$

then the difficulty of finding collisions for f_H is equivalent to the *worst case* complexity of the approximate shortest vector problem APPRSVP in a lattice of dimension m.

Goldreich *et al.*suggest that the function f_H is suitable as a cryptographic hash function. However, in practice matters are not as easy and nice. Firstly, as m and n tend to infinity, multiplicative constants and even log factors may not be of great theoretical importance, but when deployed in real life such a cryptographic system

is likely to employ lattices of dimension a few hundreds, if not thousands. In these cases, the constants and log factors are significant and crucial to the efficiency. From equation 5.3, we can show that we must have

$$m < \frac{n}{5 \lg n},$$

which implies, for example, that an algorithm that finds collisions in dimension $n = 500$ can be turned into an algorithm to solve apprSVP in dimension m, but only with $m \leq 11$. Similarly, finding collisions in dimension $n = 1000$ gives an apprSVP algorithm in dimension at most $m = 20$; and even dimension $n = 10000$ only gives an apprSVP algorithm in dimension at most $m = 150$.

Given the efficiency of LLL-type algorithms in low dimension, it thus appears that the practical security of hash functions based directly on the GGH compression function f_H must depend on the *average-case difficulty of solving Ajtai's problem itself in high dimension*, rather than on the derived difficulty of solving worst case apprSVP in much lower dimensions.

Furthermore, if using the output of the linear function f_H as the hash value one does not achieve the concrete security level one would want in practice: The output hash length is $m \lg q$ bits so the size of the hash space is q^m; and thus it is required that the best method for finding collisions will take time no less than the generic birthday attack costing $\sqrt{q^m}$ operations, as is required of all hash functions. We will see in section 5.1.2 that one can find collisions in f_H in time significantly shorter than $\sqrt{q^m}$, and we also describe an even faster attack if the function f_H is used as the compression function in a Merkle–Damgård construction (MD).

However, despite this negative fact of not being able to rely on the asymptotic worst-case/average-case analysis of [GGH96] to derive concrete security guarantees for a practical GGH hash function instantiation, it is not hard to (asymptotically) relate the security of the function f_H to the hardness of certain standard problems in the lattice L_H. The following result is reproduced from [Dwo97] and [GGH96].

Proposition 5.1. (a) *Inversion of f_H is equivalent to finding, for a given vector $\mathbf{a} \in \mathbb{R}^n$, a vector that differs from $\mathbf{a}$ by a binary vector, that is, finding a vector $\mathbf{x}$ satisfying*

$$\mathbf{x} \in L_H \qquad \text{and} \qquad \mathbf{x} - \mathbf{a} \in \mathcal{B}_n.$$

In particular, such a vector $\mathbf{x}$ always satisfies $\|\mathbf{x} - \mathbf{a}\| \le \sqrt{n}$, and on average it will satisfy $\|\mathbf{x} - \mathbf{a}\| \approx \sqrt{n/2}$.

(b) *Finding a collision for f_H is equivalent to finding a nonzero ternary vector in L_H, that is, finding a vector in the intersection*

$$\mathbf{x} \in \mathcal{T}_n \cap L_H \qquad \text{with } \mathbf{x} \neq 0.$$

In particular, such a collision-producing vector always satisfies $\|\mathbf{x}\| \le \sqrt{n}$, and on average a collision gives a vector $\mathbf{x} \in L_H$ satisfying $\|\mathbf{x}\| \approx \sqrt{n/2}$.

Here is an elementary proof of this proposition.

Proof. For (a), suppose that we are given $b \in (\mathbb{Z}/q\mathbb{Z})^m$ and want to solve $f_H(y) = b$ for $y \in \mathcal{B}$. We begin by finding any vector $a \in \mathbb{Z}^n$ satisfying $Ha \equiv -b \pmod{q}$. This is easy to do, since the congruence $Ha \equiv -b$ has more variables than equations. Of course, we are assuming that there is at least one solution. Now the following problems are equivalent:

- Solve $f_H(y) = b$.
- Find $y \in \mathcal{B}_n$ satisfying $Hy = b$. (Since the domain of f_H is $\mathcal{B}$.)
- Find $y \in \mathcal{B}_n$ satisfying $H(y + a) = 0$. (Since $b = -Ha$.)
- Find $x \in L_H$ satisfying $x - a \in \mathcal{B}_n$ (Letting $x = y + a$.)

This completes the proof of (a).

For (b), we first observe that if $f_H(x) = f_H(y)$, then $x - y \in L_H$ and clearly $x - y$ is ternary. Conversely, suppose that $z \in L_H$ is a ternary vector. Then z can be written

as a difference $z = x - y$ of binary vectors, so $f_H(x) = f_H(y)$ and we have produced a collision.

Binary and ternary vectors of dimension n have length at most $\sqrt{n}$, and the average length of a binary vector is $\sqrt{n/2}$. The average length of a ternary vector is $\sqrt{2n/3}$, but the average length of the difference of two binary vectors (which is how the ternary vectors are being produced) is $\sqrt{n/2}$. □

We have made the conservative assumption that solving ApprSVP for the lattice L_H yields a collision for f_H, but this is actually only true if the solution is a ternary vector. A detailed analysis using standard assumptions, e.g., assuming that the collection of lattices $\{L_H\}$ satisfies the Gaussian heuristic (cf. [HPS98, MS01]), yields a more precise statement. One finds that for the suggested parameters, given later, solving ApprSVP in L_H to within a factor of approximately 2.5 is likely to yield a ternary vector, and hence a collision of f_H. In the opposite direction, solving ApprSVP in L_H to within a factor of say 1.8 is unlikely to yield a collision, since almost all vectors of this size in L_H are not ternary vectors, see section 5.3.4 on page 102 for details.

5.1.2 Collisions in the GGH construction in less than $\sqrt{q^m}$ operations

In this section we describe an attack on the plain GGH compression function construction. In particular we show that for fixed parameter sizes one does not achieve the security that is hoped from the Goldreich *et al.*construction.

Before giving the details of the attack, let us first examine the lattice associated with the GGH construction closely and work out some of its properties.

Dimension and discriminant of L_H. Note that $q\mathbb{Z}^n \subset L_H \subset \mathbb{Z}^n$, so we clearly have that

$$\dim(L_H) = n.$$

If the map f_H is surjective, then we have the following *short exact sequence*[2]

$$0 \longrightarrow L_H \longrightarrow \mathbb{Z}^n \xrightarrow{H} (\mathbb{Z}/q\mathbb{Z})^m \longrightarrow 0.$$

This implies that $(\mathbb{Z}/q\mathbb{Z})^m \cong \mathbb{Z}^n/L_H$, which allows us to compute the discriminant of L_H as follows

$$\Delta(L_H) = [\mathbb{Z}^n : L_H] = \#(\mathbb{Z}/q\mathbb{Z})^m = q^m.$$

Bases for L_H. We first recall that a lattice has many matrix bases which are related through multiplication by unimodular matrices. We shall now describe two methods for obtaining a basis for L_H. Since the dimension of the lattice is equal to n, the basis matrix should also be of dimension $n \times n$.

For this section, we adopt the *row-wise* basis convention (For use with the NTL library, see § 2.6), i.e. a matrix $B = (v_1, \ldots, v_m)^T \in \mathbb{R}^{m \times n}$ is a basis for the lattice L_B iff

$$L_B = \left\{ xB : x \in \mathbb{Z}^m \right\}.$$

1. If f_H is already known to be surjective then a basis for the lattice L_H can be derived by first computing the $n \times (n-m)$-kernel matrix of H over the integers, which we denote by K_H. This kernel matrix takes the special form

$$K = \begin{pmatrix} K^* \\ \\ I_{n-m} \end{pmatrix} \quad \text{for some } m \times (n-m) \text{ submatrix } K^*,$$

where I_{n-m} is the $(n-m) \times (n-m)$ identity matrix.

---

[2]A short exact sequence of groups G_1, G_2, G_3, written as $0 \to G_1 \to G_2 \to G_3 \to 0$, is given by two maps $\pi_1 : G_1 \to G_2, \pi_2 : G_2 \to G_3$ where π_1 is injective and π_2 is surjective. An important corollary of this is that kernel of π_2 is the image of π_1 and hence the group G_1 can be viewed as a normal subgroup of G_2 and most importantly we have

$$G_3 \cong G_2/G_1,$$

which we have used here.

A basis for our lattice L_H can then be obtained from the rows of the matrix

$$\begin{pmatrix} (K^*)^T & I_{n-m} \\ qI_m & 0 \end{pmatrix}.$$

where the submatrix $(K^*)^T$ is the transpose of K^*.

2. More generally, a basis matrix for the lattice L_H can be derived by first finding a spanning set of vectors that spans the lattice L_H, then reducing it will provide us with a basis. This can be done as follows: First, form the *Smith Normal Form* (SNF) of H as

$$S_H = UHV,$$

where S_H is diagonal and U, V are square invertible integer matrices.

If we let r denote the rank of H, then the lattice L_H is spanned by the first r rows of V^T. When the corresponding diagonal entry $s_{i,i}$ of S is not equal to one, we multiply the corresponding row of V^T by $q/s_{i,i} \pmod q$. This $r \times n$ matrix is then augmented with the rows of the $n \times n$ matrix qI_n. A basis from this spanning set can then be obtained in the standard manner. We define B_H to be the row-oriented basis matrix obtained in this way.

The attack. Our first attempt to make the GGH construction practical used the linear function f_H directly as the compression function, exactly like Goldreich *et al.*construction. We soon noticed that if we assume that f_H is surjective then the basis of the associated lattice looked surprisingly special, as it can be written in the following form (as has just been explained in the previous note on bases for L_H)

$$\begin{pmatrix} (K_H^*)^T & I_{n-m} \\ qI_m & 0 \end{pmatrix}.$$

An attack related to this idea of the authors was pointed out by an anonymous referee for an earlier version of LASH (which did not use the Miyaguchi–Preneel scheme or the post processing step, that will be sketched later) finds a binary vector

in the lattice associated to f_H in time $q^{m/3}$ and thus can be used to break the collision resistance of a hash function based solely on the GGH construction.

The attack works as follows: To find a collision, we only need to consider vectors of the form $x = (y||0)$ where $y \in \mathcal{B}_{n-m}$ and $0 \in \mathcal{B}_m$. The vector x produces a lattice vector of the form $(y(K_H^*)^T, y)$. If we try to solve $y(K_H^*)^T \pmod q = 0$ then the resulting lattice vector will be a binary vector in the lattice.

However, solving $y(K_H^*)^T \pmod q = 0$ has been studied by Wagner [Wag02] in terms of a k-sum generalisation of the birthday paradox. This problem can be solved as follows: We divide the $n-m$ row vectors of $(K_H^*)^T$ into four lists and form $q^{m/3}$ combinations of the row vectors in each list such that their lower third parts xor to zero (i.e. they are equal in their lower third parts). Then we use the technique of Wagner to find a subset sum equal to zero modulo q. We expect such a subset sum to exist since the entries of $(K_H^*)^T$ are essentially random elements modulo q. Thus the running time is $q^{m/3}$, which is the time to produce the lists and the time to run Wagner's algorithm.

One can extend this method by constructing a list of 2^d partial matrix-vector products by using d message bits in a message block and running through all combinations (i.e. subset sums of rows of $(K_H^*)^T$). By choosing another d message bits, another list of equal size can be produced. It is possible to merge these distinct lists in essentially $O(2^d)$ time to produce a third list of equal size that has the property of having d selected bits as zero. The process can be recursively applied in a tree-like fashion to produce a collision in kd bits of the internal state with the selection of $2^k d$ message bits and $O(2^{k+d})$ effort in optimal conditions.

A Hybrid Attack on the MD construction

We will outline a hybrid attack that combines cycle-based collision finding techniques with linear algebra and a time-memory trade-off against the GGH function applied directly to multi-block messages using the MD construction, i.e. LASH with a different compression function, i.e. the function f_H as the compression function, and no output transform.

The general strategy of the attack is to try to select two-block messages in a way that forces a cycle-based collision finding algorithm such as [OW99] into a smaller cycle, thus producing collisions faster. If the outputs belong to a subset S of possible outputs, collision search will have $O(\sqrt{|S|})$ complexity, assuming that the message selection process is $O(1)$.

The messages are chosen as follows. The first block of the message contains the output of the previous iteration in the collision finding algorithm. The message bits in the second block are chosen in a way that causes a number of bits in the internal state of the hash function be to zero, hence forcing the final output to a smaller subset of possible outputs. The algorithm for selecting the second message block requires $O(1)$ time. The message selection algorithm is as follows:

1. Since carry propagation in addition is from least significant bits towards higher bits, $H \cdot b \pmod 2$ is in fact a system of linear equations in $\mathbf{F}_2$, independent of the 7 higher bits in each byte of H. Using simple linear algebra operations in $\mathbf{F}_2$, bit 0 in each of the m state bytes can be forced to zero by selecting m message bits appropriately. This is an $O(1)$ step.

2. A precomputed lookup table is used to force further c bits to zero. The table has 2^c entries and uses $m + c$ message bits (since the table entries must also have least significant bits as zeros). Each lookup requires $O(1)$ time. The precomputation phase requires $O(2^c)$ time.

Thus, by selecting $2m + c$ message bits in the second block in a certain way, $m + c$ bits in the $8m$-bit internal state are forced to zero. The offline complexity of the attack is $O(2^c)$ and the collision search algorithm is expected to find a collision in $O(2^{\frac{1}{2}(7m-c)})$ steps.

First consider the hypothetical case where LASH would have the standard MD structure. In this case the internal state would have the same size as the final output, i.e. $8m$ bits. If we choose $c = \frac{7}{3}m \approx 2.33m$, the overall complexity of the algorithm will be $O(2^{2.33m})$, which is significantly less than $O(2^{4m})$ expected by direct application of the birthday paradox. However, since the internal state of LASH is

twice as wide as the final output, the security goal of LASH is $O(2^{2m})$. This is the rationale behind the final transformation of LASH.

We note that it is possible to also force bit 1 of each byte to zero if the message block is large enough so that additional m^2 message bits can be selected. This is why a relatively short message block size is being used (larger message blocks would have resulted in greater hashing speed).

5.2 Design of LASH

We now turn to describing the criteria of how we selected the parameters m, n, q and the matrix H that define the function f_H used in our construction.

- Due to the fact that finding collisions in f_H is easier than the naive $q^{m/2}$, we take m to be larger than one needs in our final hash function output. This is also useful to defeat various other generic attacks on hash functions and is consistent with the advice of Lucks [Luc04].

- It turns out to be convenient in our chaining algorithm to select $n = 2m \log_2 q$.

- Whilst a value of $q = 2^{32}$ is more likely to place us in the range of the inequality (5.3), we have found via various experiments that since the output size of the hash function is fixed (and so m is limited), a harder lattice problem is produced if q is smaller. Hence, we select $q = 2^8$.

- The matrix H was chosen so that it does not require too much storage, easy to compute on the fly yet still "random enough".

We give more detailed comments on these criteria in section 5.2.2, after specifying the exact form of our proposed compression and hash function.

To use the GGH construction as a component in our practical proposal, we modify it slightly to avoid its linearity and use it as a compression function. We then extend the domain to an arbitrary length using the standard construction of Merkle and Damgård with strengthening (MD) [Mer90, Dam88]. This construction provides a provably secure collision resistant hash function, under the assumption

that the compression function is itself collision resistant. When combined with the technique of Goldreich *et al.*one obtains a collision resistant hash function which can take arbitrary length inputs. Recent work showing that the MD construction is weak in certain circumstances [Jou04, KS05] can be resolved with minor alterations, see for example [CDMP05, Luc04].

Our approach is to take the idea behind the construction of Goldreich *et al.*and try and obtain an efficient hash function whose security is related to finding short vectors in a particular fixed lattice. We will study whether this lattice behaves as a random lattice, and whether the underlying hard problem is actually secure.

5.2.1 Specification of the LASH hash functions family

Here, we present an efficient (supposedly) collision resistant hash function whose performance is comparable to that of SHA2. The design has been motivated by implementation quality, including issues such as speed and memory footprint, and the ability to fully utilise processor features available in current computer architectures.

LASH-x computes an x-bit hash value from an input bit-string of arbitrary length (less than 2^{2x} bits). There are four concrete proposals which are detailed in the following table.

LASH Variant	Input *bit*-length n	Hash *byte*-length m
LASH-160	640	40
LASH-256	1024	64
LASH-384	1536	96
LASH-512	2048	128

Here, n is the size of the input to compression function in *bits*, and m is the size of the chaining variable in 8-bit *bytes*. We have for all versions that $n = 16m$.

Compression function. We define a compression function f that takes-in two sequences of bytes $r = (r_0, r_1, \ldots, r_{m-1})$ and $s = (s_0, s_1, \ldots, s_{m-1})$ and produces a new byte sequence $t = (t_0, t_1, \ldots, t_{m-1})$. The compression function can be represented as

$$f(r, s) = (r \oplus s) + f_H(r \| s) \pmod{q}, \tag{5.4}$$

where f_H is the linear function obtained from multiplying a matrix H, defined next, by the column vector $(r\|s)^T$, interpreted as a bit vector. Figure 5.1 is a visualisation of the LASH compression function, and Algorithm 12 gives it in more detail for ease of implementation.

Thus the compression function is based on a combination of addition modulo 2^8 and xoring (bitwise exclusive or). This combination helps defeat the attacks on the naive use of the GGH construction on its own.

Algorithm 12 The LASH-m compression function
Input: Chaining variable r and message block s (byte arrays).
Output: Compression $t = f(r, s)$.

for $i = 0, 1, \ldots, m-1$ **do**
  $t_i \leftarrow r_i \oplus s_i$ ⟨*Initialise with XOR*⟩
end for
for $i = 0, 1, \ldots, n$ **do**
  if $i < 8m$ **then**
    $x \leftarrow \left\lfloor r_{\lfloor i/8 \rfloor} / 2^{7-(i \bmod 8)} \right\rfloor \bmod 2$ ⟨*Bit i from r, for $i < n/2$*⟩
  else
    $x \leftarrow \left\lfloor s_{\lfloor i/8 \rfloor - m} / 2^{7-(i \bmod 8)} \right\rfloor \bmod 2$ ⟨*Bit $(i - 8m)$ from s, for $i \geq n/2$*⟩
  end if
  if $x = 1$ **then**
    for $j = 0, 1, \ldots, m-1$ **do**
      $t_j \leftarrow t_j + a_{((n+j-i) \bmod n)} \bmod 2^8$ ⟨*Add column*⟩
    end for
  end if
end for
return t

$$\begin{bmatrix} t_0 \\ \vdots \\ t_{m-1} \end{bmatrix} = \left(\begin{bmatrix} r_0 \\ \vdots \\ r_{m-1} \end{bmatrix} \oplus \begin{bmatrix} s_0 \\ \vdots \\ s_{m-1} \end{bmatrix} \right) + \overbrace{\begin{bmatrix} \cdots & H & \cdots \end{bmatrix}}^{n=16m} \begin{bmatrix} r_0 \\ \vdots \\ r_{m-1} \\ s_0 \\ \vdots \\ s_{m-1} \end{bmatrix}$$

Figure 5.1: Visualising $t = f(r, s) = (r \oplus s) + f_H(r\|s) \pmod{q}$.

Pseudorandom sequence and the matrix H. Consider the following pseudorandom sequence. Start with $y_0 = 54321$ and iterate the following recurrence based on the Pollard generator

$$y_+ = y^2 + 2 \pmod{2^{31} - 1}.$$

We define an additional sequence that results in reducing y_i to byte length integers (truncation), which will serve as elements of the matrix H:

$$a_i = y_i \pmod{2^8}$$

The first eight members of this sequence are

$$a_0 = 49 = \texttt{0x31}, \quad a_1 = 100 = \texttt{0x64}, \quad a_2 = 135 = \texttt{0x87}, \quad a_3 = 237 = \texttt{0xED},$$
$$a_4 = 95 = \texttt{0x5F}, \quad a_5 = 26 = \texttt{0x1A}, \quad a_6 = 139 = \texttt{0x8B}, \quad a_7 = 214 = \texttt{0xD6}.$$

Note that the modulus $2^{31}-1$ in the Pollard generator is a Mersenne prime, which allows us to perform faster modular reduction hence speeding-up the pseudo-random number generation. We point out also that to reduce modulo 2^8 one can simply use bit masks.

We take the matrix H to be the $m \times n$ *circulant matrix* associated to the sequence $a_0, \ldots, a_n$ (Circulant matrices are a special type of Toeplitz matrices)

$$H = \begin{pmatrix} a_0 & a_{n-1} & a_{n-2} & \cdots & a_2 & a_1 \\ a_1 & a_0 & a_{n-1} & \cdots & a_3 & a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_{m+1} & a_m \end{pmatrix}.$$

Hashing the message. Let ℓ be the bit-length of the original message to be hashed. Let us call the individual message bytes $v_0, v_1, v_2, \ldots, v_{\lceil \ell/8 \rceil}$.

We first pad the message with a single '1' bit (in case of byte-aligned data, this corresponds to a single byte with hexadecimal value `0x80`) and then we add enough bytes v_i with a zero value to make the length a multiple of $8m$.

The message is cut into $k = \lceil \ell/8m \rceil$ blocks of m bytes and fed to the compression function, and then a final transform is performed, which involves applying the compression function to the chaining variable and the binary encoding of ℓ, to produce a message digest.

Algorithm 13 describes the overall hash function. We should note that the use of IV = 0 is not secure because of the recent attack presented in [CMP+07], which is

sketched in section 5.5.3. Another fixed value of IV should be studied and carefully chosen as to circumvent similar attacks on IV = 0. At the time of writing, this has still not been resolved and is left as an open problem.

Algorithm 13 The LASH-m hash function
Input: A padded message v (= ...||0x80||0...0) of bit-length ℓ.
output: LASH-$m(v)$.

for $i = 0, 1, \ldots, m-1$ **do**
  $r_i = 0$ ⟨*Initialise chaining variable (IV=0), see 5.5.3*⟩
end for
for $i = 0, 1, \ldots, \lceil \ell/8m \rceil - 1$ **do**
  for $j = 0, 1, \ldots, m-1$ **do**
    $s_i = v_{m \cdot i + j}$ ⟨*Get message-block*⟩
  end for
  $r \leftarrow f(r, s)$ ⟨*Compression function, Algorithm 12*⟩
end for
for $i = 0, 1, \ldots, m-1$ **do**
  $s_i \leftarrow \lfloor \ell / 2^{8i} \rfloor \mod 2^8$ ⟨*ℓ encoded in little-endian*⟩
end for
$r \leftarrow f(r, s)$ ⟨*Final compression*⟩
for $i = 0, 1, \ldots, m/2 - 1$ **do**
  $t_i = 16\lfloor r_{2i}/16 \rfloor + \lfloor r_{2i+1}/16 \rfloor$ ⟨*High 4 bits of output bytes*⟩
end for
return t ⟨*$m/2$-byte hash result.*⟩

5.2.2 Comments on the design of LASH

In this section we go into more detail over the precise design choices we have made. The main goals of the design have been as follows:

- To adopt the large-pipe strategy of Lucks [Luc04] to avoid problems with the Merkle–Damgård construction. The final hash value is then produced from the large-pipe by taking the upper half bits of each byte – these being the bits which depend in the most non-linear manner on the input values.

- To combine two forms of mathematical operations in the compression function: Arithmetic modulo 256 and bitwise exclusive-or (xor). The compression functions consists of two parts: A linear function, motivated by the GGH construction, and an xoring of the chaining variable and the next message block motivated by the construction of Miyaguchi–Preneel [MOI90, Pre93].

- To be able to reason about the ability of the linear function to resist preimages and collisions.

- To be as simple and efficient as much as possible, particularly aiming for application on as wide a range of platforms as possible. Thus the hash function is byte oriented and built from components found most modern processors and which are easy to implement in hardware.

- To enable as much parallelism as possible, thus allowing the hash function to exploit performance enhancing features in modern instruction sets.

- The hash function should be patent free, as such none of the designers have taken out patents on its design.

Linear Function

We chose to use a circulant matrix whose entries are generated with a Pollard type PRNG because the use of a circulant matrix allows more efficient implementations of our function f_H and less storage requirements for the matrix H, and deriving the entries via a pseudorandom number generator allows us to reduce the memory requirements of our hash function even more.

The non-linearity of the generator is crucial in creating a matrix for which the associated lattice problem is hard to solve. For example, we have found that using a linear-congruential PRNG instead of the Pollard PRNG results in a compression function that is easy to break using the LLL algorithm.

The choice of the prime modulus $p = 2^{31} - 1$ in the Pollard generator is made to enable a sequence with period greater than the largest value of n, and so $\sqrt{p}$ should be greater than the largest value of n chosen. In addition, we selected the modulus p for which modular reduction can be performed efficiently because of its special form: We only need a few additions and bit shifts akin to Algorithm 1 in Chapter 2.

Compression Function

Recall that the compression function for LASH is defined, for the m-byte chaining variable r and the next m-byte message block s, by

$$f(r,s) = (r \oplus s) + f_H(r||s) \pmod{q}.$$

The compression function is highly motivated by the hash functions construction from Block Ciphers by Miyaguchi–Preneel [MOI90, Pre93], which is of the form

$$f(r,s) = (r \oplus s) \oplus E_{g(r)}(s),$$

for a block cipher $E_k(m)$ and a function g which takes inputs the size of the chaining variable and outputs keys for the block cipher. That is to say, we are treating the function f_H as equivalent to a block cipher with key r and message s.

We are not claiming that the function f_H can be used as a block cipher. So, the "proof of security" of the Miyaguchi–Preneel construction [BRS02] does not necessarily apply to the LASH compression function. However, the function f_H does have some interesting properties which it shares with a block cipher, as is implied by Proposition 5.1 e.g.

1. Given an output f_H it is hard to invert.

2. It is hard to find collisions in the function f_H.

The difference lies in the *exact* complexity of these problems. Generally speaking, these problems seem to be easier for f_H because of its linearity.

Final Transformation

In the final transform, we need to compress the $8m$ bit chaining variable down to its half to get the output hash value of length $4m$ bits. Recall that each byte of the chaining variable has been obtained by performing a lot of additions modulo $q = 2^8$, which have been dependent on the message bits. To compute the final hash value, we select the upper four bits of each byte of the final value of the chaining variable

(as they are affected to the most unpredictable extent by the carry propagation) and concatenate them together. This produces an output of the correct size.

The reason for taking the upper four bits is that, due to the nature of addition modulo q, these are going to be the bits which are affected in the most non-linear manner by the effect of carry propagation in the modular addition operations. Hence, it is this upper half of the bytes that should enjoy more entropy than the lower half.

5.3 Security considerations

The general structure of LASH, having only linear components, easily leads one to suspect that it is directly vulnerable to differential and linear cryptanalysis. LASH has gone through several evolutionary stages after the idea of a lattice-based hash function was first considered. The current version is a result of combining the traditions of provable complexity-theoretic security with symmetric cryptanalysis.

In determining the security of LASH against these attacks, we note that as a fully parameterisable family of hash function (message block size, state size, and hash result size can all be flexibly chosen), simulation of attacks against LASH is straightforward and meaningful. If an attack can be successfully mounted and simulated on reduced variants of LASH, and the asymptotic behaviour of the security as a function of various parameters established, then concrete evidence about the security of the full-size variants can be obtained. This flexibility also makes it easy to create larger versions of LASH if weaknesses are found in the current proposed versions. This is a clear advantage of LASH over many hash function designs with a more rigid block-cipher like structures.

5.3.1 Differential cryptanalysis

A small input difference (in either the chaining variable and/or the message block) will result in a very large difference in the hash function state. Differential trails are very wide. The propagation of differentials is further amplified in the final iteration

(which does not use message bits), making all output bits differentially dependant on all input bits.

We conjecture that the simple and understandable structure of LASH will make it difficult to find differential anomalies such as the so-called necessary conditions exploited by Wang *et al.*in their attacks on MD5, SHA1, and other hash functions [WLF+05, WY05, WYY05, XW05].

5.3.2 Linear cryptanalysis

All components of the LASH compression function are, in some sense, linear. Furthermore, if we consider a matrix H' that contains the least significant bits of H, then the product function $H' \cdot b$ is a linear equation in $\mathbb{F}_2$ and indeed H' is invertible with a significant probability. This can be exploited in some attacks, as is done in the hybrid attack presented in Section 5.1.2. We note that these attacks are difficult to extend to the full version of LASH, however.

It is unlikely that classical linear cryptanalysis (involving the parity of subsets of bits) can be applied on LASH.

5.3.3 Generalised birthday attack

Wagner's method for solving the generalised birthday problem [Wag02] can directly be applied to the GGH construction, as was shown in section 5.1.2. Using the GGH function f_H on its own implies that we can find collisions in $O(q^{m/3})$ operations as opposed to the $O(q^{m/2})$ operations one would want in practice from a hash function.

Although improvements to this basic version of the attack can be made, this attack does not seem to be applicable to the internal f_H function used in LASH, due to the ratio between the message block size and the size of the internal state. This motivates our choice of a large chaining variable and our output transformation. Our use of the Miyaguchi–Preneel construction, as opposed to using the function f_H directly also helps defeat this attack.

5.3.4 Ternary vectors in lattices

We want to develop some tools needed to analyse whether solutions to an approximate shortest vector problem in a lattice $L \subset \mathbb{Z}^n$ are likely or unlikely to be ternary vectors. This section aims to present an analysis on how hard it is to either invert or find collisions in the internal function f_H via lattice basis reduction.

Before commencing we reiterate that finding collisions or inverting f_H is *not sufficient* to break LASH due to the use of the Miyaguchi-Preneel construction, but may be a first step in some attack on this construction.

Which balls contain many ternary lattice points?

Let $\mathcal{T}_n$ be the set of ternary vectors of dimension n as usual, and let $B_n(R)$ be the ball of radius R centred at 0 in $\mathbb{R}^n$. If R is small, than most of the integral lattice points in $B_n(R)$ will be ternary vectors, while if R is large, then few of them will be ternary. We would like to determine a critical value R_n at which the ternary vectors cease to predominate. This should be roughly the value R such that the number of ternary vectors of norm at most R is equal to the volume of the ball of radius R, i.e., R_n solves the equation

$$\mathrm{Vol}(B_n(R)) = \#\Big(\mathcal{T}_n \cap B_n(R)\Big).$$

Using the formula for the volume of an n-dimensional ball and the counting formula for ternary vectors, we see that R_n solves

$$\frac{\pi^{n/2}}{\Gamma(n/2+1)} R^n = \sum_{d=0}^{\lfloor R^2 \rfloor} \binom{n}{d} 2^d. \tag{5.5}$$

The sum on the right-hand side of (5.5) is a step function, so the equation (5.5) tends to have several solutions. For example, if $n = 100$, then (5.5) has 14 solutions ranging from 4.992 to 6.087. Although this does not give an exact solution, it tells us that a ball of radius 5 in $\mathbb{R}^{100}$ contains mostly ternary lattice points, while a ball of radius (say) 10 contains proportionally very few ternary lattice points. Table 5.1 gives the largest, smallest, and average solutions to (5.5) for a range of dimensions.

n	$R_n^{\min}$	R_n^{mean}	$R_n^{\max}$
50	3.15042	3.90777	4.58992
100	4.99171	5.55618	6.08738
150	6.32237	6.81316	7.28238
200	7.48077	7.90118	8.30731
250	8.48252	8.83002	9.16873
300	9.37782	9.69343	10.0022
350	10.1947	10.4858	10.7715
400	10.9082	11.2014	11.4894
450	11.6179	11.8743	12.1269
500	12.2867	12.5294	12.7689

Table 5.1: Solutions to $\text{Vol}(B_n(R)) = \#\big(\mathcal{T}_n \cap B_n(R)\big)$

It is clear from Table 5.1 that R_n^{mean} does not grow linearly with n. For our data, the regression line of $\log(R_n^{\text{mean}})$ versus $\log(n)$ is

$$\log(R_n^{\text{mean}}) \approx 0.50634 \log(n) - 0.6173 \tag{5.6}$$

with correlation coefficient 0.999996. This suggests that $R_n \approx c\sqrt{n}$.

We next relate the sum on the right-hand side of (5.5) to a binomial distribution and use a normal approximation to prove the validity of this guess and find an asymptotic value for c.

Proposition 5.2. *For large values of n, the equation*

$$\frac{\pi^{n/2}}{\Gamma(n/2+1)} R^n = \sum_{0 \le d \le R^2} \binom{n}{d} 2^d \tag{5.7}$$

has a solution R satisfying $R \approx 0.4332\sqrt{n}$. (This may be compared with the experimental value $R \approx 0.54 \cdot n^{0.506}$ given by (5.6).)

Proof. For any $r > 0$,

$$\sum_{0 \le d \le r} \binom{n}{d} 2^d = 3^n \sum_{d=0}^{r} \binom{n}{d} \left(\frac{2}{3}\right)^d \left(\frac{1}{3}\right)^{n-d}$$

is 3^n times the probability that a binomial distribution (with probabilities 1/3 and 2/3) is smaller than r. If n is large, we can approximate this probability using

the normal distribution

$$\begin{aligned}\Phi(x) &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-t^2/2}\,dt \\ &= \sqrt{\frac{2}{\pi}}\cdot\frac{1}{|x|}e^{-x^2/2}\left(1+O(1/x^2)\right) \qquad \text{for } x<0.\end{aligned}$$

Thus

$$\begin{aligned}\frac{1}{3^n}\sum_{0\le d\le r}\binom{n}{d}2^d &= \sum_{d=0}^{r}\binom{n}{d}\left(\frac{2}{3}\right)^d\left(\frac{1}{3}\right)^{n-d} \\ &\sim \Phi\left(\frac{r-2n/3}{\sqrt{2n/9}}\right) \qquad \text{as } n\to\infty.\end{aligned}$$

To ease notation, we let $r = \alpha n$ and set $\beta = (3\alpha - 2)/\sqrt{2}$, so the above quantity is $\Phi(\beta\sqrt{n})$.

Using the elementary asymptotic expansion for $\Phi(x)$ (valid for $x < 0$) and Sterling's formula to approximate $\Gamma(x)$, the equation (5.7) that we are trying to solve (with $R = \sqrt{r} = \sqrt{\alpha n}$) becomes

$$(2\pi er/n)^{n/2} \approx 3^n\Phi(\beta\sqrt{n})$$

$$(2\pi e\alpha)^{n/2} \approx 3^n\cdot\sqrt{\frac{2}{\pi}}\cdot\frac{1}{|\beta|\sqrt{n}}\cdot e^{-\beta^2 n/2}$$

Taking n^{th} roots and letting n go to infinity gives the equation

$$\sqrt{2\pi e\alpha} = 3e^{-\beta^2/2}$$

to be solved for α, where recall that $\beta = (3\alpha - 2)/\sqrt{2}$. The numerical solution is $\alpha \approx 0.18762$, so we find that the solutions R to (5.7) are given approximately by

$$R = \sqrt{\alpha n} \approx 0.4332\sqrt{n}.$$ □

Which general lattice problems have many ternary solutions?

Let $L \subset \mathbb{Z}^n$ be a lattice of dimension n and let $\lambda(L)$ denote the length of a shortest nonzero vector in L. Proposition 5.2 suggests that if $\lambda(L)$ is significantly smaller

than $R_n \approx 0.4332\sqrt{n}$, then most solutions to APPRSVP will be ternary vectors, but if $\lambda(L)$ is significantly larger than R_n, then only a small proportion of the solutions to APPRSVP will be ternary vectors. Combining this observation with the value of $\lambda(L)$ given by the Gaussian heuristic yields the following result.

Proposition 5.3. *Let $\mathcal{L}$ be a class of lattices for which the Gaussian heuristic is valid and fix $\epsilon > 0$. Then for $L \in \mathcal{L}_n$, solutions $v \in L$ of* APPRSVP *satisfying*

$$\|v\| < (1-\epsilon) \cdot \frac{1.79}{\mathrm{Disc}(L)^{1/n}} \cdot \lambda(L)$$

are quite likely to be ternary vectors, while solutions $v \in L$ of APPRSVP *satisfying*

$$\|v\| > (1+\epsilon) \cdot \frac{1.79}{\mathrm{Disc}(L)^{1/n}} \cdot \lambda(L)$$

are unlikely to be ternary vectors.

In particular, if Disc(L) *is significantly larger than* 1.79^n*, then even a shortest vector in L (i.e., a solution to* SVP*) is unlikely to be a ternary vector.*

Proof. The Gaussian estimate says that the shortest nonzero vector in a "typical lattice" has length

$$\lambda(L) \approx \sqrt{n/2\pi e}\,\mathrm{Disc}(L)^{1/n}.$$

(See, e.g., [HPS98, MS01].) Solving APPRSVP in L yields a vector of length $C\lambda(L)$ for some $C \geq 1$. Proposition 5.2 says that this vector is quite likely to be a ternary vector if $C\lambda(L) < 0.4332(1-\epsilon)\sqrt{n}$ and that it is not very likely to be a ternary vector if $C\lambda(L) > 0.4332(1+\epsilon)\sqrt{n}$. Thus the critical value for C is

$$\begin{aligned} C &= \frac{0.4332\sqrt{n}}{\lambda(L)} \approx 0.4332 \cdot \sqrt{2\pi e} \cdot \mathrm{Disc}(L)^{-1/n} \\ &\approx 1.79 \cdot \mathrm{Disc}(L)^{-1/n}. \end{aligned}$$

□

Which lattice problems arising from f_H have many (or mostly) ternary solutions?

If we are to base a hash function upon the linear function f_H, then we would want the difficulty of finding binary (resp. ternary) vectors in L_H to be at least as hard as inversion (resp. finding collisions) of f_H via generic methods. An interesting aspect of the lattices we shall use is that for a fixed output size of the linear function, the value $\Delta^{1/n}$ of the associated lattice tends to one as we increase the dimension of the lattice, i.e. the input block size of the linear function.

As indicated by Proposition 5.1 (page 86), the ability of finding collisions in f_H depends on the difficulty of finding special sorts of short vectors in the circulant lattice L_H. The NTRU cryptosystem [HPS98] is also based on the difficulty of finding short vectors in certain lattices (called convolution modular lattices in [MS01]) that are built up out of circulant matrices. However, the matrices (and lattices) underlying LASH are rather different from those underlying NTRU, so the associated lattice problems are also different.

We now apply the results of the previous section to the lattices L_H used by LASH. Recall that $\dim(L_H) = n$ and $\mathrm{Disc}(L_H) = q^m$. Notice that if we make the assumption that $q^m < 2^n$, which is required if f_H is to be a compression function, then $1 < \mathrm{Disc}(L_H)^{1/n} < 2$.

Proposition 5.4. *Assume that the Gaussian heuristic holds for the LASH lattices* (5.1).

(a) *If $q^m > 1.8^n$, then solving* apprSVP *in L_H is unlikely to give a ternary vector.*

(b) *If $q^m < 1.78^n$, then solving* apprSVP *in L_H to within a factor of $1.79/q^{m/n}$ is quite likely to give a ternary vector.*

Proof. This is immediate from Proposition 5.3 using the values $\dim(L_H) = n$ and $\mathrm{Disc}(L_H) = q^m$. □

Finally, we apply Proposition 5.4 to the specific LASH parameters $q = 256$ and $n = 16m$. We find that

$$q^m = (2^8)^{n/16} = (2^{1/2})^n \approx 1.414^n.$$

Hence, q^m is less than 1.78^n, which implies that all of the LASH lattices are likely to contain many ternary vectors. The crucial quantity is the approximation factor

$$\frac{1.79}{q^{m/n}} = \frac{1.79}{(2^8)^{1/16}} \approx 1.27,$$

which tells us how closely we need to solve APPRSVP in order to (probably) find a ternary vector.

The conclusion is that in order to find a collision in the linear function for the suggested parameters, it is probably necessary to find a vector in L_H that is no more than about 2.5 times as long as the shortest nonzero vector. However, we note once more that finding collisions in the linear function f_H is not *sufficient* to find collisions in LASH itself.

5.4 Implementation

Now that we have presented the new hash function proposal, it will be very informative to implement it in practice and get concrete performance figures out. In this section, we will comment on some aspects of the implementation of LASH in software and produce some benchmarks to allow us to compare it against the currently recommended hash functions. Some thoughts and comments on the possible hardware implementation of LASH and some of its variations that should be immune to side channel analysis (SPA and DPA) is given in [Pag07] but no actual implementation has been reported.

Storage of the pseudorandom data. We have several options as regards storage of the pseudorandom matrix. A compromise seems the most attractive option, that is to store only part of the matrix. Due the circulant nature, there is no real benefit in storing the whole matrix since each row is essentially a rotation of the first. Therefore, we can simply store one row and be able to access all the required elements by shifting a window from right to left; at each of n steps, the window contains the elements for the corresponding column.

The circulant nature of the matrix has an additional property in that neighbouring columns differ only in one element. Therefore, one can imagine storing only a single column of the matrix and updating it by computing a new entry at each step. This creates a computational overhead in that we need to generate a total of n matrix entries, but offers a saving in storage overhead since there are far less rows than columns in the matrix.

Parallelism in the compression function. The basic algorithm for executing the compression function offers parallelism in two directions. Firstly, since the matrix columns do not affect each other in the matrix-vector multiplication, one can operate on them at once summing the partial vector dot-products to form the final result. Secondly, one can add different elements of a given column into the state in parallel. These two method combine to offer a high degree of scalability. This is easy to exploit in hardware or where a dedicated SIMD instruction set is available.

We can manually apply a similar technique on processors which do not have SIMD instruction sets but do have a native word size greater than 8-bits. For example, on a 32-bit processor we can pack four 8-bit sub-words into one 32-bit value. We cannot add packed values using native 32-bit addition since carries from one sub-word may overflow into another. However, we can construct a suitable method for addition by masking the top bits of the packed bytes to prevent carries before using 32-bit addition and patching up the result. The resulting packed addition of x and y to produce the result r can be described as

$$\begin{aligned} x' &\leftarrow x \wedge \texttt{0x7F7F7F7F} \\ y' &\leftarrow y \wedge \texttt{0x7F7F7F7F} \\ r' &\leftarrow x' + y' \\ r &\leftarrow ((x \oplus y) \wedge \texttt{0x80808080}) \oplus r' \end{aligned}$$

with a similar construction possible for other word sizes.

Specialisation of the compression function. Considering how the compression function is used to process arbitrary length messages, the first and last invocations can be considered special. In the first invocation the chaining variable is zero; in the

last invocation the message block is mostly zero with only a few bytes representing the message length. In both cases, only a small portion of the compression function input is relevant and in the first case the initial mixing stage is redundant since $t_i = r_i \oplus s_i = s_i$ for all i.

The saving afforded from capitalising on these features by using specialised versions of the compression function is amortised over all invocations. For short messages, the saved computation can be significant since the first and last invocations of the compression function comprise the majority of the total.

5.4.1 Results

Table 5.2: Comparing the performance of LASH with standardised hash functions.

Name	Implementation options SIMD	Matrix storage	Storage (bytes)	Cycles/Byte
SHA1-160	✘ [Den]		0	26.29
SHA1-160	✓ [Gau]		64	16.86
LASH-160	✘	All matrix	25600	689.64
LASH-160	✘	One row	640	774.42
LASH-160	✓	All matrix	25600	392.83
LASH-160	✓	One row	640	523.26
SHA2-256	✘ [Den]		256	55.16
SHA2-256	✘ [Gay]		288	31.34
SHA2-256	✓ [Gau]		256	45.20
LASH-256	✘	All matrix	65536	859.83
LASH-256	✘	One row	1024	1027.74
LASH-256	✓	All matrix	65536	344.81
LASH-256	✓	One row	1024	597.01
SHA2-384	✘ [Den]		640	124.57
SHA2-384	✘ [Gay]		704	117.45
LASH-384	✘	All matrix	147456	1078.58
LASH-384	✘	One row	1536	1355.09
LASH-384	✓	All matrix	147456	805.47
LASH-384	✓	One row	1536	1090.41
SHA2-512	✘ [Den]		640	124.98
SHA2-512	✘ [Gay]		704	117.52
LASH-512	✘	All matrix	262144	1351.39
LASH-512	✘	One row	2048	1730.14
LASH-512	✓	All matrix	262144	1036.70
LASH-512	✓	One row	2048	1220.54

We recompiled and tested publicly available source code for the SHA1 and SHA2 hash functions [Den, Gau, Gay], as well as preliminary implementations of

LASH, on our experimental platform. This platform housed a 2.8GHz Pentium 4 processor running the 2.4.21 Linux kernel. All source code was written in C, making use of GCC 4.0.1 and the intrinsics feature to access the SIMD functionality of the processor. Measurement of the number of cycles elapsed during execution was performed using the `rdtsc` instruction in the normal way.

Table 5.2 shows the results of the experiment and compares SHA1 and SHA2 with equivalent parameterisations of LASH. The results were averaged over a large number of random inputs; it is vital to note that LASH performance is variable depending on the input. Also note that the storage requirement is intended to detail only the amount of pre-computed material rather than the total memory footprint.

The results show an encouraging ratio between the fastest implementations of LASH versus SHA1 and SHA2. In particular, LASH is potentially at most only about 30 times slower than SHA1 with the ratio improving significantly for SHA2 with LASH being only 10 to 20 times slower. This is comparable, at the lower security levels, with an implementation of VSH; although results for this latter clearly depend on how large one takes the modulus in ones VSH implementation (before truncation at the end of the computation, if this method is used to produce shorter digests).

5.4.2 Test vectors

We provide test vectors for each variant of LASH (with `IV` = 0), for the purpose of testing one's own implementation. The vectors are computed over two test messages A and B. The message A is a 24 bits string which consists of three lower-case ASCII characters "abc", whose corresponding hexadecimal bytes are 61 62 63. The message *B* consists of 100000 repetitions of the ten ASCII characters "0123456789", with corresponding hexadecimal bytes 30 31 32 33 34 35 36 37 38 39. The message length of B is 8 million bits ($100000 \times 10 \times 8 = 8 \cdot 10^6$).

```
LASH-160(A) =
        67 58 25 ec  f3 ba f5 c9  4f fe 38 a1  5b c0 ab 40  77 9b 96 4d
LASH-160(B) =
        43 68 df 33  4f ce b9 e7  99 d2 77 22  12 fc 44 f2  ce ec 04 1e
```

```
LASH-256(A) =
        39 ff b7 84  0b 6b 3b 71  89 fc 5e dc  9e 24 33 9e
        77 8c f4 be  bf 94 df 00  c3 53 d0 bf  37 30 b3 2f
LASH-256(B) =
        e9 57 75 d4  53 d6 36 1e  3c 9c 88 8c  dc eb 3c 8a
        ab 49 cd ad  43 56 b5 ba  97 98 38 6b  b6 dc 95 e9

LASH-384(A) =
        11 d0 9c 55  cb ba 6f 31  10 bf 87 7f  ab cf b6 30
        10 52 0c 30  76 e1 dc d2  7b af dc a8  38 5e 25 0e
        4e fa 42 97  a1 6c 69 23  b9 a1 33 3d  8d ca 1d a7
LASH-384(B) =
        41 7e cb d6  dd 54 2f 82  e4 29 e4 ec  93 e6 c0 78
        3d 81 7c 5e  38 4d d2 e4  97 61 6c b1  0f 32 6e b6
        10 5c ef 9e  32 ba 2f 97  9b 5e 94 8b  31 e7 8c 75

LASH-512(A) =
        c5 bb 7c f4  c1 ca c6 38  43 94 66 65  7c 8d ed 14
        bb ab f8 28  e4 b3 69 99  86 11 64 b9  79 2d 88 fd
        48 eb 0f aa  aa f4 e0 33  19 fc bd 4d  4e 5c 2c 06
        82 5a 85 97  35 98 69 dd  1e 84 0b 12  15 96 19 c8
LASH-512(B) =
        07 02 25 1f  85 b4 5a a7  78 0d f4 9d  69 b2 de b0
        20 12 c5 e3  20 46 7e 3b  04 a3 4f fa  75 a0 19 0d
        c8 f5 41 20  c2 33 a5 08  38 26 a8 e6  47 68 2c 5b
        59 c0 9e d2  52 c7 1e 81  66 f6 2e 59  ef fb 24 57
```

To help with finding bugs in implementations of LASH, we further give a trace of the internal variables when hashing the three-byte ASCII string "abc" with LASH-160 in Appendix A.2 on page 151.

5.5 Attacks on LASH

5.5.1 Some weak matrix dimensions

First, note that this attack does not apply to the parameters set that we have proposed. This attack is on the LASH compression function and was given at the end of a presentation during the NIST Second Cryptographic Hash Workshop [LMPR06]. I thank V. Lyubashevsky and C. Peikert for explaining the attack to me (by correspondence through email).

The attack only applies to matrices with $m = 232, 368, 1056, 2096, \ldots$ because the sum of the elements of each row is zero (If the sum of the first row's elements is zero then so is sum for all the other rows as their elements are permutations of the first row). This immediately implies that the two bit-strings 1^n and 0^n will collide

with a hash value 0. Here is how to find collisions for the mentioned dimensions. Starting from the seed 54321, the matrix H will have row entries that sum up to 0 modulo 256. So now, if we take $r = s = 1^{n/2}$ then $f_H(r||s) = 0$. Also, $r \oplus s = 0$. And this is the exact same value we get if $r = s = 0^{n/2}$, and so we have a collision.

There are various possible generalisations of the attack sketched above that also work (Peikert):

- For any dimension, the probability that the attack above works is about 1/256, taken over the random choice of the seed (or a completely random choice of H's first row).
- For $r = s = (1, 0, 1, 0, \ldots)$ and any even dimension, there is a probability about 2^{-16} of this input colliding with the all-zeros input.
- Various other periodic 0-1 patterns also work, with probabilities that drop off with the length of the period.

5.5.2 LASH is not a pseudo-random function (PRF)

First, we note that the only claimed properties of LASH are collision and pre-image resistance. Hash functions are used for a variety of purposes, and in some cases they are assumed to be pseudo-random functions, but it is noted in [CMP+07] that LASH is not and the following attack is given to show this fact.

First, separate the matrix H into its left and right halves $H = (H_L||H_R)$, then the compression function can be written as

$$f(r, s) = (r \oplus s) + H_L r + H_R s.$$

Now, note that for $s = 0$ we have

$$f(r, 0) = r + H_L r.$$

and for $s' = (2^7, 0, \ldots, 0)$ we get

$$f(r, s') = (r_0 \oplus 2^7, r_1, \ldots, r_{m-1}) + H_L r + H_{R_0},$$

where H_{R_0} is the first column of H_R. Notice that the difference between these two values is constant and independent of r

$$f(r,s') - f(r,0) = H_{R|0} + (2^7, 0, \ldots, 0)^T.$$

This fact allows us to distinguish between a family of truly random functions and the LASH compression function.

5.5.3 Exploiting zero IV

This attack is also given in [CMP+07], and it is the most serious attack so far as it shows that LASH with $IV = 0$ is not collision resistant and furthermore that it is not pre-image resistant either. The presented attack uses a time/memory tradeoff and exploits the zero IV to cleverly "absorb" the xor operation into the linear function f_H. This trick yields an attack costing $2^{\frac{4}{11}m} < 2^{m/2}$ for finding collisions and $2^{\frac{4}{7}m} < 2^m$ for finding pre-images.

Another heuristic collision attack based on lattice reduction using two approaches, solving either an SVP or a CVP, is also given and is supposed to cost less than $2^{m/2}$. However, this attack produces colliding messages which are very long.

It is also shown that if one changes the value of IV to be non-zero then LASH still suffers from being vulnerable to pre-image attacks costing about $2^{\frac{7}{8}m} < 2^m$ space and time. However, the pre-images produced by this attack are of a very special type as they are 1 block messages only.

5.5.4 Attacks on the final compression

This attack is presented in [CMP+07] for the final LASH compression. It uses a generalisation of Wagner's method to solve multi-birthday problems. The cost of the attack is $O(m2^{m/(4+4/105)}) \approx O(m2^{m/4})$.

However, we note that all of these attacks require a comparable amount of storage to their running time, which questions the validity of these attacks. But

these attacks remain acceptable from the academic point of view as they show that this construction is not as secure as previously believed.

Chapter 6

The equivalence between the DLP and DHP

"It is important to understand that an asymptotic result–such as my theoretical argument that established the inefficiency of xedni *in the limit as the size of the group increases – cannot be relied upon as any kind of guarantee of security. Rather, one must analyze the algorithm for elliptic curves of the size employed in cryptography."*

— Neal Koblitz

The theoretical equivalence between the Discrete Logarithm Problem (DLP) and the Diffie-Hellman Problem (DHP) over a cyclic group of prime order $p > 3$ was first shown to hold by Maurer [Mau94] in 1994, subject to a mild existence condition of a smooth order elliptic-curve group over the finite field $\mathbb{F}_p$. His reduction was later used by Muzereau, Smart and Vercauteren [MSV04] to study the special case of elliptic curves used in practical cryptographic applications as recommended in the SECG standard [SEC00], which encompasses most of the other Elliptic-Curve Cryptography (ECC) standards (see §1.6.3).

In this chapter, we will build on the Muzereau *et al.*work and try to establish the tightest possible reduction from DLP to DHP using Maurer's reduction. We achieve this aim in two ways, first by using *projective coordinates* instead of affine coordinates and secondly by exploiting a special type of DH-oracles that allow arbitrary choice of the group generator.

For the rest of this chapter, we let G be a cyclic group with prime order $p > 3$ and a fixed generator g, unless otherwise indicated. We write $G = \langle g \rangle$ and $|G| = p$.

We will use $\mathfrak{M}$ and $\mathfrak{I}$ to denote multiplications and inversions in G, respectively, and $\mathfrak{DH}$ for DH-oracle calls. Formulae of the form $x\mathfrak{DH} + y\mathfrak{I} + z\mathfrak{M}$ mean: Cost is x DH-oracle calls, y inversions and z multiplications in G.

6.1 Maurer's reduction method in $\mathbb{F}_p$

Note that, since solving any instance of the DHP given access to a DL-oracle is trivial[1], we only concentrate on the reverse implication for the equivalence to hold: If we suppose the DHP turns out to be easy, we wish to know if this implies that the DLP is easy as well.

Maurer and Wolf proved, in a series of papers, that for every cyclic group G with prime order $p > 3$, the DLP and DHP over G are equivalent if there exists an elliptic curve, called *auxiliary elliptic curve*, over $\mathbf{F}_p$ with a smooth order [Mau94, MW96a, MW96b, MW00].

More concretely, the following result is shown in [Mau94] and [MW00].

Theorem 6.1. *Let G be a group. If each large prime factor p of $|G|$ is single and if for every such p a cyclic elliptic-curve group over $\mathbf{F}_p$ is known with smooth order then breaking DHP and DLP are equivalent for G.*

Here "single" means that the prime factor is not a repeated factor, or in other words: It only appears to the first power in the prime factorisation of $|G|$. Note also that we require the auxiliary elliptic curve groups for the large primes only as we can afford to use the traditional methods for the smaller primes without really affecting the overall cost.

Muzereau *et al.* [MSV04] showed that such auxiliary elliptic-curve groups are highly likely to exist for almost all elliptic curve groups. It is however remarked that it gets extremely hard to construct them as the order of G increases. They explicitly generated auxiliary groups with smooth orders for most of the curves in the SECG standard, hence making Maurer's proof concrete and applicable to most of the groups used in practical ECC.

[1] Given $g^a, g^b \in G$, we compute $a = \mathcal{DL}(g^a)$ and then compute $g^{ab} = (g^b)^a$.

The idea behind the reduction method introduced by Maurer [Mau94] rests on the concept of *implicit representation*: The *implicit representation* of an integer a (modulo p) is defined to be $g^a \in G$. The algorithm proceeds by doing computations in the implicit representation instead of the usual explicit representation. For example, to compute $a+b$ in implicit form, $g^a \cdot g^b$ is computed instead which costs one multiplication. For $a-b$, we compute $g^a \cdot (g^b)^{-1}$ costing one inversion and one multiplication. To compute $a \cdot b$ in implicit form, one call to an *DH-oracle*, that computes g^{ab} given g^a and g^b, is needed. For the implicit form of a^{-1}, one uses the fact that $a^{p-1} \equiv 1 \pmod{p}$, so $g^{a^{p-2}} = g^{a^{-1}}$, which would cost $O(\lg p)$ calls to the DH-oracle. Hence, granted access to a DH-oracle for the group G, all algebraic algorithms in $\mathbb{Z}_p$ can be converted to work in the implicit representation in G.

In this chapter we will build on the work in [MSV04] by tightening the reduction and trying to extend the result to the remaining curves that were missed. Our goal is to show that, for the elliptic-curve cryptosystems described in the various standards, the number of group operations and DH-oracle calls required to reduce the DLP to the DHP is reasonably "small." Say for example that the number of calls to the DH-oracle is less than 2^r then, if we believe that the much more extensively studied DLP over the same group takes *at least* 2^ℓ operations to solve then an algorithm for solving the DHP, and thus breaking the DHP protocol, would require a minimum of $2^{\ell-r}$ group operations. Our target is therefore to minimise the value of r, in order to get the tightest possible security reduction.

In [MSV04], affine coordinates were used to represent the points on the auxiliary elliptic curve groups. This representation requires division and hence a *DH-inversion oracle* was needed, which was implemented via repeated calls to the DH oracle using the relation $g^{a^{-1}} = g^{a^{p-2}}$ at the cost of $O(\lg p)$ calls to a DH-oracle to compute the exponentiation. This approach is clearly an expensive choice as it leads to a large increase in the number of DH-oracle calls. To avoid this extra cost, we use *projective coordinates* to avoid division, and as a further refinement we also use a specially tailored optimised square root extraction algorithm.

The reduction method involves lots of exponentiations, so one may also consider using *addition chains* to reduce the cost of exponentiation. However, it turns out that

this saves very little and only complicates the analysis. So it was decided to only use a more generic method of exponentiation and concentrate on the other critical areas of the reduction algorithm. Section 6.4 expands on this point and justifies this decision.

Appendix A.1 provides a list of auxiliary elliptic-curve groups that give almost the tightest possible reduction from the DLP to the DHP, using the Maurer method.

We need to address two cases that depend on the way we define the DH-oracle that will be used in the reduction. The first one will be an oracle with respect to a fixed generator of the cyclic group G, while the second is when the generator can be freely chosen by the environment i.e. the generator is part of the input to the DH-oracle, i.e the oracle is given a triple (g, g^a, g^b) where $\langle g \rangle = G$ as input instead of just (g^a, g^b) when g is fixed.

6.1.1 Case 1: Fixed base DH-oracle

We now define the problems DLP and DHP in the case of a *fixed* generator g of a cyclic group G.

Definition 6.1 (DLP and DHP). *Let G be a cyclic group. Fix a generator g of G and write $G = \langle g \rangle$.*

- *Given $h \in G$, the problem of computing the integer $\alpha \in [0, |G|)$ such that $g^\alpha = h$ is called the* Discrete Logarithm Problem *(DLP) with respect to g.*
- *Given two elements $g^a, g^b \in G$, where a and b are unknown, we call the problem of computing g^{ab} the* Diffie-Hellman Problem *(DHP) with respect to g.*

In the definition of the DLP, the existence and uniqueness of the integer α are implied by the fact that the group G is cyclic of prime order $|G| = p$. Next, we formalise the notions of Diffie-Hellman (DH) and Discrete Logarithm (DL) oracles.

Definition 6.2 (DL and DH oracles). *Let G be a cyclic group. Fix a generator g of G and write $G = \langle g \rangle$.*

- *A* DH-oracle *takes as input two elements $g^a, g^b \in G$ and returns g^{ab}. We write $\mathcal{DH}(g^a, g^b) = g^{ab}$.*

- *A* DL-oracle *takes as input an element* $h = g^a \in G$ *and returns* $a \bmod |G|$. *We write* $\mathcal{DL}(h) = \mathcal{DL}(g^a) = a$.

Both oracles return answers in unit time (by definition of oracles).

The equivalence between the two problems for any group was theoretically established by Maurer and Wolf in the nineties [Mau94, MW96a, MW96b, MW99, MW00], but it relies on the existence of some auxiliary elliptic curves whose orders must be smooth. These auxiliary elliptic curves are not necessarily easy to build and it seems they are exceptionally hard to find in general. Hence, a more concrete treatment for the elliptic curve groups used in practice proved necessary and this was done in [MSV04]. The paper discussed the *computational equivalence* between the DLP and DHP, and it also presented an explicit list of auxiliary elliptic curves needed for the reduction.

The optimised reduction algorithm

Given an element $h \in G$ and granted access to a DH-oracle for G, we want to find the unique integer α modulo p such that $h = g^\alpha$. We assume that we have an elliptic curve E over $\mathbb{F}_p$, given by the Weierstrass equation $y^2 = x^3 - 3x + b$, with smooth order that can be written as a product of *coprime integers*

$$|E| = \prod_{j=1}^{s} q_j, \tag{6.1}$$

with $q_j < B$ of roughly the same size, where B is a fixed smoothness bound that is polynomial in $\log p$.

The specific choice of $y^2 = x^3 - 3x + b$ for the defining equation of E saves $1\mathfrak{DS}$ while adding points on it as we can optimise the addition formulae to save one multiplication.

To solve a DLP in G, Maurer's approach is to use the given DH-oracle and solve the problem in the implicit representation over the elliptic curve E, which is supposed to have a smooth order (Hence we can use the Pohlig-Hellman simplification, see below for the details).

So, given $h = g^{\alpha} \in G$ and the elliptic curve E, as above, we check whether $g^{y^2} = g^{\alpha^3 - 3\alpha + b}$ can be solved for y. If so then we have found a point Q on E in its implicit form, otherwise we replace α by $\alpha + d$ for some random, small, integer d and do the checking again until we get a point Q on E.

Note that, at this stage, we know Q in its implicit representation only. The idea now is to solve $Q = kP$ over E, where P is a generator of E. Upon finding the value of k, we then compute kP in the explicit representation and hence recover the value of α, from the explicit first coordinate of Q. Given that E has a smooth order, we simply use the naive Pohlig-Hellman method of first solving the problem in the subgroups of E of prime power order, and then recovering k using the Chinese Remainder Theorem (CRT). The reader is referred to Algorithm 14 for the detailed description of the algorithm.

The crucial point to note is that we have a wide choice of curves over $\mathbf{F}_p$ that have sizes distributed in the Hasse interval $[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$. So, with a bit of luck, one hopes that one of these sizes is smooth enough and hence the corresponding auxiliary elliptic curve would make solving our DLP easy, granted access to an appropriate DH-oracle. We draw the reader's attention to the fact that this is the same reason that makes the ECM factoring method so successful.

In the description of Algorithm 14, note that for the comparison step (12:) to test whether a point $(X : Y : Z)$, in projective Jacobian coordinates, is equal to a point (x, y), in affine coordinates, we simply check whether $(X/Z^2, Y/Z^3) = (x, y)$ i.e. $xZ^2 = X$ and $yZ^3 = Y$. In the implicit representation this becomes

$$(g^{Z^2})^x = g^X \quad \text{and} \quad (g^{Z^3})^y = g^Y.$$

This use of projective coordinates gives our greatest improvement over [MSV04]. We also make extra savings by storing precomputed values and using them throughout the algorithm. The next two subsections will describe the other improvements.

Algorithm 14 Solve a DLP in a group G given access to a DH-oracle for G
Input: A cyclic group $G = \langle g \rangle$ of prime order p, an elliptic curve $E/\mathbb{F}_p : y^2 = x^3 - 3x + b$, generated by P, $|E| = \prod_{j=1}^{s} q_j$ and $h = g^\alpha \in G$
Output: $\alpha = \mathcal{DL}(h)$

Step 1. Compute a valid implicit x-coordinate related to the DL α

1: **repeat**
2: Choose d randomly, and set $g^x \leftarrow hg^d$ $\langle g^x \leftarrow g^{\alpha+d} \rangle$
3: $g^z \leftarrow g^{x^3-3x+b}$
4: **until** $g^{z^{(p-1)/2}} = g$ $\langle$*Test quadratic-residuosity of* $z \pmod p$$\rangle$

Step 2. Compute g^y from $g^z = g^{y^2}$:

5: Extract the square root of z in implicit representation, to obtain g^y
Now, $Q = (x, y)$ is a point on E known implicit only: (g^x, g^y)
Step 3. Compute k: $Q = kP$ in $E(\mathbb{F}_p)$: $\langle$*Pohlig-Hellman*$\rangle$
6: **for** $j = 1, \ldots, s$ **do**
7: Compute $Q_j = (g^{u_j}, g^{v_j}, g^{w_j})$, where $(u_j, v_j, w_j) = \frac{|E|}{q_j} Q$ $\langle$*Projective*$\rangle$
8: Set $i \leftarrow 0$, $(u, v) \leftarrow O$, $P_j \leftarrow \frac{|E|}{q_j} P$ $\langle$*Affine*$\rangle$
9: **repeat** $\langle$*Solve* $Q_j = k_j P_j$ *in the subgroup of* $E(\mathbb{F}_p)$ *of order* $q_j$$\rangle$
10: $i \leftarrow i + 1$
11: $(u, v) \leftarrow (u, v) + P_j$ $\langle (u, v) \leftarrow iP_j = i\frac{|E|}{q_j} P \rangle$
12: **until** $(g^{w_j^2})^u = g^{u_j}$ and $(g^{w_j^3})^v = g^{v_j}$ $\langle$*Test if* (g^u, g^v) *equals* $(g^{u_j}, g^{v_j}, g^{w_j})$$\rangle$
13: $k_j \leftarrow i$
14: **end for**
Step 4. Construct α
15: Compute $k \pmod{|E|}$ such that $\forall j \in \{1, \ldots, s\} : k \equiv k_j \pmod{q_j}$ $\langle$*CRT*$\rangle$
16: Compute $kP = Q$ in affine coordinates
17: Then $x \pmod p$ is the abscissa of Q, and $\alpha = x - d$

Square root extraction

We will now describe the special cases in the explicit notation. The next formulae are used by Algorithm 15, in the implicit representation, to compute g^y from $g^z = g^{y^2} = g^{x^3-3x+b}$.

Suppose a is known to be a quadratic residue modulo p, using the Legendre symbol for example (§ 2.2), and we want to compute $x \in \mathbb{F}_p$ such that $x^2 \equiv a \pmod p$. Then, besides the general Tonelli and Shanks algorithm used in [MSV04], we also treat two special cases:

1. If $p \equiv 3 \pmod 4$ then $x \equiv a^{(p+1)/4} \pmod p$,

2. If $p \equiv 5 \pmod 8$ then do the following: Compute $s = a^{(p-5)/8}$, $u = a \cdot s$, $t = s \cdot u$.
If $t = 1$ then $x = u$ otherwise $x = 2^{(p-1)/4} \cdot u$.

Algorithm 15 Implicit square roots in a group G using a DH-oracle for G.
Input: A cyclic group $G = \langle g \rangle$ of odd prime order p, and $g^z = g^{y^2} \in G$.
Output: g^y.

1: **if** $p \equiv 3 \pmod 4$ **then**
2: $g^y \leftarrow g^{z^{(p+1)/4}}$. ⟨*First case:* $p \equiv 3 \pmod 4$⟩
3: **else if** $p \equiv 5 \pmod 8$ **then**
4: $g^s \leftarrow g^{z^{(p-5)/8}}, g^u \leftarrow g^{zs}, g^t \leftarrow g^{su}$. ⟨*Second case:* $p \equiv 5 \pmod 8$⟩
5: **if** $g^t = g$ **then**
6: $g^y \leftarrow g^u$.
7: **else**
8: $g^y \leftarrow g^{u \cdot 2^{(p-1)/4}}$.
9: **end if**
10: **else**
11: Write $p - 1 = 2^e \cdot w$, w odd. ⟨*Tonelli and Shanks algorithm for* $p \equiv 1 \pmod 8$⟩
12: Set $g^s \leftarrow g, r \leftarrow e, g^y \leftarrow g^{z^{(w-1)/2}}, g^b \leftarrow g^{zy^2}, g^y \leftarrow g^{zy}$. ⟨*Initialise*⟩
13: **while** $g^b \neq 1$ **do**
14: Find the smallest $m \geq 1$ such that $g^{(b^{2^m})} \equiv g$. ⟨*Find exponent*⟩
15: Set $g^t \leftarrow g^{(s^{2^{r-m-1}})}, g^s \leftarrow g^{t^2}, r \leftarrow m, g^y \leftarrow g^{yt}, g^b \leftarrow g^{bs}$. ⟨*Reduction*⟩
16: **end while**
17: **end if**

Treating these special cases is worthwhile since half the primes are congruent to 3 modulo 4, and half of the remaining primes are congruent to 5 modulo 8. The only remaining primes are all congruent to 1 modulo 8. We gain no advantage by using similar methods for this case, so we simply use the Tonelli-Shanks algorithm for the remaining primes as described in Chapter 2 (Section 2.2 on page 43).

Explicit and implicit point multiplication

As already stated, we use the projective coordinate system in Step 3 of Algorithm 14 instead of the affine coordinate system. The formulae for addition and doubling of points[2] in the implicit representation follow from their standard explicit counterparts [BSS99, p. 59–60] as follows (Recall that we are assuming that $a = -3$ and $p > 3$ in the defining equation of our elliptic curve E).

[2]Recall that "doubling" is the operation of adding a point to itself: $2P = P + P$.

Doubling. Let $P = (X : Y : Z)$ and $Q = 2P = (X' : Y' : Z')$. Then

$$
\begin{aligned}
\lambda_1 &= 3X^2 + aZ^4 & g^{X^2} &= \mathcal{DH}(g^X, g^X)\\
\lambda_2 &= 4XY^2 & g^{Y^2} &= \mathcal{DH}(g^Y, g^Y)\\
\lambda_3 &= 8Y^4 & g^{Y^4} &= \mathcal{DH}(g^{Y^2}, g^{Y^2})\\
X' &= \lambda_1^2 - 2\lambda_2 & g^{Z^2} &= \mathcal{DH}(g^Z, g^Z)\\
Y' &= \lambda_1(\lambda_2 - X') - \lambda_3 & g^{Z^4} &= \mathcal{DH}(g^{Z^2}, g^{Z^2})\\
Z' &= 2YZ & g^{\lambda_1} &= (g^{X^2})^3 \cdot (g^{Z^4})^a\\
 & & g^{\lambda_2} &= (\mathcal{DH}(g^X, g^{Y^2}))^4\\
 & & g^{\lambda_3} &= (g^{Y^4})^8\\
 & & g^{X'} &= \mathcal{DH}(g^{\lambda_1}, g^{\lambda_1}) \cdot (g^{\lambda_2})^{-2}\\
 & & g^{Y'} &= \mathcal{DH}(g^{\lambda_1}, g^{\lambda_2} \cdot (g^{X'})^{-1}) \cdot (g^{\lambda_3})^{-1}\\
 & & g^{Z'} &= \mathcal{DH}((g^Y)^2, g^Z)
\end{aligned}
$$

So the cost of explicit doubling is $8\mathfrak{M}$ and that of implicit doubling is $8\mathfrak{DH}+4\mathfrak{I}+14\mathfrak{M}$.

Addition. Let $P = (X_1 : Y_1 : Z_1)$, $Q = (X_2 : Y_2 : Z_2)$ and $R = P+Q = (X_3 : Y_3 : Z_3)$. Then

$$
\begin{aligned}
\lambda_1 &= X_1 Z_2^2 & g^{Z_1^2} &= \mathcal{DH}(g^{Z_1}, g^{Z_1})\\
\lambda_2 &= X_2 Z_1^2 & g^{Z_1^3} &= \mathcal{DH}(g^{Z_1^2}, g^{Z_1})\\
\lambda_3 &= \lambda_1 - \lambda_2 & g^{Z_2^2} &= \mathcal{DH}(g^{Z_2}, g^{Z_2})\\
\lambda_4 &= Y_1 Z_2^3 & g^{Z_2^3} &= \mathcal{DH}(g^{Z_2^2}, g^{Z_2})\\
\lambda_5 &= Y_2 Z_1^3 & g^{\lambda_1} &= \mathcal{DH}(g^{X_1}, g^{Z_2^2})\\
\lambda_6 &= \lambda_4 - \lambda_5 & g^{\lambda_2} &= \mathcal{DH}(g^{X_2}, g^{Z_1^2})\\
\lambda_7 &= \lambda_1 + \lambda_2 & g^{\lambda_3} &= g^{\lambda_1} \cdot (g^{\lambda_2})^{-1}\\
\lambda_8 &= \lambda_4 + \lambda_5 & g^{\lambda_4} &= \mathcal{DH}(g^{Y_1}, g^{Z_2^3})\\
 & & g^{\lambda_5} &= \mathcal{DH}(g^{Y_2}, g^{Z_1^3})\\
 & & g^{\lambda_6} &= g^{\lambda_4} \cdot (g^{\lambda_5})^{-1}\\
 & & g^{\lambda_7} &= g^{\lambda_1} \cdot g^{\lambda_2}\\
 & & g^{\lambda_8} &= g^{\lambda_4} \cdot g^{\lambda_5}\\
 & & g^{\lambda_3^2} &= \mathcal{DH}(g^{\lambda_3}, g^{\lambda_3})\\
 & & g^{\lambda_3^3} &= \mathcal{DH}(g^{\lambda_3^2}, g^{\lambda_3})\\
Z_3 &= Z_1 Z_2 \lambda_3 & g^{Z_3} &= \mathcal{DH}(\mathcal{DH}(g^{Z_1}, g^{Z_2}), g^{\lambda_3})\\
X_3 &= \lambda_6^2 - \lambda_7 \lambda_3^2 & g^{X_3} &= \mathcal{DH}(g^{\lambda_6}, g^{\lambda_6}) \cdot (\mathcal{DH}(g^{\lambda_7}, g^{\lambda_3^2}))^{-1}\\
\lambda_9 &= \lambda_7 \lambda_3^2 - 2X_3 & g^{\lambda_9} &= \mathcal{DH}(g^{\lambda_7}, g^{\lambda_3^2}) \cdot (g^{X_3})^{-2}\\
Y_3 &= (\lambda_9 \lambda_6 - \lambda_8 \lambda_3^3)/2 & g^{Y_3} &= \{\mathcal{DH}(g^{\lambda_9}, g^{\lambda_6}) \cdot (\mathcal{DH}(g^{\lambda_8}, g^{\lambda_3^3}))^{-1}\}^{1/2}
\end{aligned}
$$

For the implicit square root extraction in the computation of g^{Y_3}, we pre-compute $2^{-1} \pmod p$ and use exponentiation. Hence the cost of explicit addition is $16\mathfrak{M}$ and that of implicit addition is $16\mathfrak{DH} + 5\mathfrak{I} + (8 + \frac{3}{2}\lg p)\mathfrak{M}$.

The cost of each operation is summarised in the following table.

	Point doubling		Point addition	
	Explicit	Implicit	Explicit	Implicit
$\mathfrak{D}$		8		16
$\mathfrak{I}$		4		5
$\mathfrak{M}$	8	14	16	$\frac{3}{2}\lg p + \frac{13}{2}$

For the affine coordinates, note that we only need the explicit case (In the j-loop).

The costs are (see 2.3.1 and take $\mathcal{M} = \mathcal{S} = \mathfrak{M}$):

$$1\mathfrak{I} + 4\mathfrak{M} \text{ for doubling and } 1\mathfrak{I} + 3\mathfrak{M} \text{ for addition.}$$

Exponentiation

Since we will need to compute kP for different values of k but a fixed P, precomputing the values $2^1P, 2^2P, \ldots, 2^{\lfloor \lg k \rfloor}P$ will save us some computation. Then, using the right-to-left binary method, we expect only $\frac{1}{2}\lg k$ elliptic curve additions. We now summarise the costs of exponentiation.

Implicit exponentiation in projective coordinates. The cost of the precomputation is about

$$(8\mathfrak{D} + 4\mathfrak{I} + 14\mathfrak{M})\lg k \tag{6.2}$$

and then each exponentiation would cost about

$$\left(8\mathfrak{D} + \frac{5}{2}\mathfrak{I} + \frac{1}{4}(3\lg p + 13)\mathfrak{M}\right)\lg k. \tag{6.3}$$

Explicit exponentiation in affine coordinates. The precomputation cost is

$$(1\mathfrak{I} + 4\mathfrak{M})\lg k \tag{6.4}$$

and then each exponentiation would cost

$$\frac{1}{2}(1\mathfrak{I} + 3\mathfrak{M})\lg k. \tag{6.5}$$

Complexity of the optimised reduction algorithm (Algorithm 14)

The average case complexity analysis of Algorithm 14, presented next, yields the following theorem.

Theorem 6.2. *Let G be a cyclic finite group of prime order p. Assume an elliptic curve E over* $\mathbb{F}_p$ *has been found, whose B-smooth order is*

$$|E| = \prod_{j=1}^{s} q_j,$$

where q_j *are not necessarily prime but are coprime of roughly the same size. Then, solving a given instance of the DLP in G requires on average about*

$$O\left(\frac{\log^2 p}{\log B}\right)\mathfrak{D}\mathfrak{H} + O\left(\frac{B\log^2 p}{\log B}\right)\mathfrak{M}.$$

For comparison, we quote below the asymptotic costs obtained by [MSV04]

$$O\left(\frac{\log^3 p}{\log B}\right)\mathfrak{D}\mathfrak{H} + O\left(\frac{B\log^2 p}{\log B}\right)\mathfrak{M}.$$

While the number of multiplications has remained the same, the number of DH-oracle calls has now become quadratic in the size of the group G instead of the previously cubic cost.

Note that, in order to get a lower bound on the cost of solving a DHP instance, we no longer require the auxiliary elliptic curves' orders to be smooth. This is because as long as we assume that the DLP is an exponentially hard problem then we do not mind if the reduction from the DHP to the DLP is exponential too. This remark will allow us to choose $s = 3$ later, and then the task will be to find smooth elliptic curves whose orders are product of three coprime numbers. This is a significant relaxation of the smoothness condition.

Analysis of the average case complexity of Algorithm 14

To simplify this task, each step of Algorithm 14 (see page 121) will be studied separately and then the partial results will be summed up to obtain the total average cost of the algorithm.

Step 1:

We first precompute g^{2^i} for $i = 1, \ldots, \lfloor \lg p \rfloor$. This will allow us to compute any power g^k with an average cost of $\frac{1}{2} \lg k \mathfrak{M}$, using the double-and-add algorithm of exponentiation. The precomputation requires $\lfloor \lg p \rfloor$ squarings, which costs

$$\lg p \mathfrak{M}.$$

Without loss of generality, we set $d = 0$ at the start of this step. Then, evaluating $g^z \leftarrow g^{x^3-3x+b} = g^{x^3} \cdot ((g^x)^3)^{-1} \cdot g^b$ requires

$$2\mathfrak{D}\mathfrak{H} + +1\mathfrak{I} + (4 + \frac{1}{2} \lg b)\mathfrak{M}.$$

Note that

$$g^{(x+d)^3-3(x+d)+b} = g^{x^3-3x+b} \cdot (g^{x^2})^{3d} \cdot (g^x)^{3d^2} \cdot g^{d^3-3d}.$$

So for a second evaluation, we only need an extra

$$\left(3 + \frac{3}{2} \lg(3d) + \frac{3}{2} \lg(3d^2) + \frac{1}{2} \lg(d^3 - 3d)\right)\mathfrak{M} \sim (3 + 3 \lg 3 + 6 \lg d)\mathfrak{M}.$$

For the quadratic residuosity check we need to compute $g^{z^{(p-1)/2}}$. First precompute $g^{z^{2^i}}$ for $i = 1, \ldots, \lfloor \lg \frac{p}{2} \rfloor$, then the total cost is

$$\left(\lg \frac{p}{2} + \frac{1}{2} \lg \frac{p-1}{2}\right)\mathfrak{D}\mathfrak{H} \sim \left(\frac{3}{2} \lg p - \frac{3}{2}\right)\mathfrak{D}\mathfrak{H}.$$

Now, let ν be the number of iterations for Step 1. Since $\mathbb{F}_p$ has $(p-1)/2$ quadratic non-residues, the probability for having $\nu = k$ iterations is

$$\Pr[\nu = k] = \left(\frac{p-1}{2p}\right)^{k-1} \cdot \frac{p+1}{2p}.$$

Hence, the expected number $\bar{v}$ of iterations for Step 1 is

$$\bar{v} = \sum_{k=1}^{\infty} k \cdot \Pr[v = k] = \frac{p+1}{2p} \sum_{k=1}^{\infty} k \left(\frac{p-1}{2p}\right)^{k-1} = \frac{2p}{p+1} \approx 2.$$

Thus the total average cost of this first step is $\lg p\mathfrak{M} + [2\mathfrak{D}\mathfrak{H} + 1\mathfrak{I} + (4 + \frac{1}{2}\lg b)\mathfrak{M}] + [(3 + 3\lg 3 + 6\lg d)\mathfrak{M}] + 2 \times (\frac{3}{2}\lg p - \frac{3}{2})\mathfrak{D}\mathfrak{H}$. That is

$$(3\lg p - 1)\mathfrak{D}\mathfrak{H} + 1\mathfrak{I} + (\lg p + \frac{1}{2}\lg b + 6\lg d + 7 + 3\lg 3)\mathfrak{M}. \tag{6.6}$$

Step 2: Following Algorithm 15, we treat three cases:

1. If $p \equiv 3 \pmod 4$ then, using the precomputations from the previous step, we can compute $g^{z^{(p+1)/4}}$ in an average of

$$\frac{1}{2}\lg\frac{p+1}{4}\mathfrak{D}\mathfrak{H} \sim (\frac{1}{2}\lg p - 1)\mathfrak{D}\mathfrak{H}.$$

2. If $p \equiv 5 \pmod 8$ then the computation of $g^{z^{(p-5)/8}}, g^{zs}$ and g^{su} costs $(2 + \frac{1}{2}\lg\frac{p-5}{8})\mathfrak{D}\mathfrak{H} \sim (\frac{1}{2}\lg p + \frac{1}{2})\mathfrak{D}\mathfrak{H}$ on average.

   If $t = 1$ then no further computation is needed and the total cost is $(\frac{1}{2}\lg p + \frac{1}{2})\mathfrak{D}\mathfrak{H}$. Otherwise, $t \neq 1$ and then computing

$$g^{u \cdot 2^{(p-1)/4}} = \mathcal{DH}(g^u, g^{(2^{(p-1)/4} \bmod p)})$$

   will cost an extra $1\mathfrak{D}\mathfrak{H} + (\frac{3}{2}\lg\frac{p-1}{4} + \frac{1}{2}\lg p)\mathfrak{M}$.

   Since t behaves like a random variable, the average cost for this case is then

$$(\frac{1}{2}\lg p + \frac{1}{2})\mathfrak{D}\mathfrak{H} + \frac{1}{2}\big(1\mathfrak{D}\mathfrak{H} + (2\lg p - 3)\mathfrak{M}\big).$$

3. Otherwise, we use the general (implicit) Tonelli and Shanks algorithm. We first write $p - 1 = 2^e \cdot w$, where w is odd.

   The initialisation step requires roughly $(\frac{1}{2}\lg\frac{w-1}{2} + 2)\mathfrak{D}\mathfrak{H}$. Finding the exponent and reducing it requires $(r + 2)\mathfrak{D}\mathfrak{H}$ per iteration, and at most e iterations are

expected. Since $r \le e$, we will need $e \cdot (r+2) \le e \cdot (e+2)$ calls to the DH-oracle. Hence, the total number of the DH-oracle calls is about

$$\left(\frac{1}{2}\lg\frac{w-1}{2} + 2 + (e+2)e\right)\mathfrak{DH}.$$

Since p is odd, we can easily see that the expected value of e is

$$\sum_{k=1}^{\infty} k \cdot \Pr[e = k] = \sum_{k=1}^{\infty} k \cdot (1/2)^k = 2.$$

Bearing this in mind, we get $w = p/2^e = p/4$ and the total cost is then estimated to be

$$\left(\frac{1}{2}\lg p + \frac{17}{2}\right)\mathfrak{DH}.$$

Note. When concluding, we will use the weighted average of the costs above, which is

$$\left(\frac{1}{2}\lg p + \frac{15}{8}\right)\mathfrak{DH} + \frac{1}{8}(2\lg p - 3)\mathfrak{M}. \tag{6.7}$$

Step 3: Before entering the j-loop, we first pre-compute

$$2^i Q \quad \text{for } i = 1, \ldots, \lfloor \lg |E|^{1-1/s} \rfloor.$$

This is enough since q_j are of roughly the same size, so $q_j \approx |E|^{1/s}$ and then $\frac{|E|}{q_j} \approx |E|^{1-1/s}$.

Using equation (6.2), the cost of precomputation is found to be about

$$(8\mathfrak{DH} + 4\mathfrak{I} + 14\mathfrak{M})\left(1 - \frac{1}{s}\right)\lg |E|.$$

We also pre-compute $2^i P$ for $i = 1, \ldots, \lfloor \lg |E| \rfloor$ in affine coordinates[3]. According to equation (6.4), this costs about

$$(1\mathfrak{I} + 4\mathfrak{M})\lg |E|.$$

[3]We need i up to $\lg |E|$ as we will use these precomputed values in Step 4 too.

Now, let j be fixed (We want to analyse the cost of one j-loop). The cost for computing $Q_j = (g^{u_j}, g^{v_j}, g^{w_j})$ such that $(u_j, v_j, w_j) = \frac{|E|}{q_j}Q$, given by equation (6.3), is about

$$\left(8\mathfrak{D}\mathfrak{H} + \frac{5}{2}\mathfrak{I} + \frac{1}{4}(3\lg p + 13)\mathfrak{M}\right)\gamma_j,$$

where we have set $\gamma_j = \lg(|E|/q_j)$. For the evaluation of $P_j = \frac{|E|}{q_j}P$, in affine coordinates, equation (6.5) gives

$$\left(\frac{1}{2}\mathfrak{I} + \frac{3}{2}\mathfrak{M}\right)\gamma_j.$$

For the i-loop, we note that $g^{w_j^2}$ and $g^{w_j^3}$ need to be computed only once for each j-loop, which costs $2\mathfrak{D}\mathfrak{H}$.

Now fix i. Computing $iR = (i-1)R + R$, in affine coordinates, can be achieved with one elliptic curve addition costing $1\mathfrak{I} + 3\mathfrak{M}$, since $(i-1)R$ has been computed and $1R = R$ is trivial.

The cost of comparison is about $2 \times \frac{3}{2}\lg p\mathfrak{M} = 3\lg p\mathfrak{M}$.

On average there will be $q_j/2$ i-loops for each j-loop, and therefore the average cost of the i-loop is

$$\frac{q_j}{2}\left(1\mathfrak{I} + 3(\lg p + 1)\mathfrak{M}\right).$$

Hence, the cost per one j-loop is

$$(8\gamma_j + 2)\mathfrak{D}\mathfrak{H} + (\frac{1}{2}q_j + 3\gamma_j)\mathfrak{I} + \left(\frac{3}{2}(\lg p + 1)q_j + \frac{1}{4}(3\lg p + 19)\gamma_j\right)\mathfrak{M}.$$

Noting that

$$\sum_{j=1}^{s}\gamma_j = \sum_{j=1}^{s}\lg\frac{|E|}{q_j} = (s-1)\lg|E|,$$

we find that the total cost for Step 3, without the precomputation costs, is on average

$$(8(s-1)\lg|E| + 2s)\mathfrak{D}\mathfrak{H} + \left(\frac{1}{2}\sum_{i=1}^{s}q_j + 3(s-1)\lg|E|\right)\mathfrak{I} +$$
$$+\left(\frac{3}{2}(\lg p + 1)\sum_{i=1}^{s}q_j + \frac{1}{4}(3\lg p + 19)(s-1)\lg|E|\right)\mathfrak{M}.$$

Adding the precomputation costs, we finally get the total cost of Step 3

$$\begin{aligned}&(8(s-1/s)\lg|E|+2s)\,\mathfrak{D}\mathfrak{H}+\left(\frac{1}{2}\sum_{i=1}^{s}q_j+(3s+2-\frac{4}{s})\lg|E|\right)\mathfrak{I}+\\&\left(\frac{3}{2}(\lg p+1)\sum_{i=1}^{s}q_j+\left(\frac{1}{4}(3\lg p+19)(s-1)+18-\frac{14}{s}\right)\lg|E|\right)\mathfrak{M}.\end{aligned}\tag{6.8}$$

Step 4: We use the Chinese Remainder Theorem to reconstruct $k \bmod |E|$ from $k \equiv k_j \pmod{q_j}$, $j = 1, \ldots, s$. We compute

$$k = \sum_{j=1}^{s} k_j \cdot \frac{|E|}{q_j} \cdot \hat{q}_j \pmod{|E|},$$

where $\hat{q}_j = \left(\frac{|E|}{q_j}\right)^{-1} \bmod q_j$. This requires $s\mathfrak{I} + 2s\mathfrak{M}$ operations. Note that inversions are computed in $\mathbb{F}_{q_1}, \ldots, \mathbb{F}_{q_s}$.

For computing kP, in affine coordinates, we use the previously precomputed values of 2^iP. So this exponentiation would cost only $(1\mathfrak{I} + 3\mathfrak{M})\frac{1}{2}\lg k$. Taking $k \bmod |E|$ to be $\frac{|E|}{2}$ on average, we find the average cost of Step 4 to be

$$\frac{1}{2}(\lg|E|-1)\mathfrak{I}+\frac{3}{2}(\lg|E|-1)\mathfrak{M}.\tag{6.9}$$

Conclusion. We conclude that the total cost of Algorithm 14 is

$$\begin{aligned}&\left(8\left(s-\frac{1}{s}\right)\lg|E|+\frac{7}{2}\lg p+2s+\frac{7}{8}\right)\mathfrak{D}\mathfrak{H}\\&+\left(\frac{1}{2}\sum_{i=1}^{s}q_j+(3s+\frac{5}{2}-\frac{4}{s})\lg|E|+\frac{1}{2}\right)\mathfrak{I}\\&+\left(\begin{array}{c}\frac{3}{2}(\lg p+1)\sum_{i=1}^{s}q_j+\left(\frac{1}{4}(3\lg p+19)(s-1)+\frac{39}{2}-\frac{14}{s}\right)\lg|E|+\\ \\ +\frac{5}{4}\lg p+\frac{1}{2}\lg b+6\lg d+3\lg 3+\frac{41}{8}\end{array}\right)\mathfrak{M}.\end{aligned}$$

Neglecting small terms and making the approximation[4] $|E| \approx p$ and $b \approx p/2$, the average cost of Algorithm 14 is then found to be

[4] By Hasse's Theorem: $|E| = p+1-t$ where $|t| \leq 2\sqrt{p}$ is the Fröbenius trace, so $|E| = p(1+(1-t)/p) \approx p$, $b \approx p/2$ is the average value of b, and d is small.

$$\left\{\left(8s-\frac{8}{s}+\frac{7}{2}\right)\lg p+2s+\frac{7}{8}\right\}\mathfrak{D}\mathfrak{H}+\left(\frac{1}{2}\sum_{i=1}^{s}q_j+(3s+\frac{5}{2}-\frac{4}{s})\lg p\right)\mathfrak{I}+$$

$$+\left\{\frac{3}{2}(\lg p+1)\sum_{i=1}^{s}q_j+\left(\frac{1}{4}(3\lg p+19)(s-1)+\frac{85}{4}-\frac{14}{s}\right)\lg p\right\}\mathfrak{M}.$$

Note that if we take q_j to be of roughly the same size and fix B to be of a similar size then

$$s\approx\frac{\log|E|}{\log B}\approx\frac{\log p}{\log B}$$

and then

$$\sum_{j=1}^{s}q_j\approx\sum_{j=1}^{s}B=sB\approx\frac{B\log p}{\log B}=\frac{B\lg p}{\lg B}.$$

In practice, the cost of an inversion is at most $10\mathfrak{M}$ for the range of operand-sizes in the standards, see [BSS99, p. 37]. Using this fact we have now established Theorem 6.2, stated on page 125.

6.1.2 Case 2: Random base

In the random base case, the DH-oracle is given a triple (g, g^a, g^b) where g is chosen at random and not necessarily a fixed generator of the cyclic group G. The definition of the DH oracle in this case then becomes:

Definition 6.3 (Random base DH oracle). *A random base* DH-oracle *takes as input three arbitrary elements* $h, h^a, h^b \in G$ *and returns* h^{ab} *in unit time. We write* $\mathcal{DH}(h, h^a, h^b) = h^{ab}$.

Note that if we invoke the DH-oracle with $(g^a, g, g^b) = (g^a, (g^a)^{1/a}, (g^a)^{b/a})$ then we obtain

$$(g^a)^{(1/a)\cdot(b/a)}=g^{b/a}. \tag{6.10}$$

Hence, in such a setup we can use our DH-oracle to perform divisions in a straight forward manner, and thus there will be no need to use non-affine coordinate systems to avoid division. This was first pointed out to me by my colleague Pooya Farshim and then by Fré Vercautern.

By Section 6.4 (page 137), we do not expect major savings using this approach either (a factor of about $2^{3.2}$ at most).

6.2 Implications on the security of the DHP

The implications of this reduction on the security of the DLP was addressed in [MSV04]. We only comment on its implications on the security of the DHP, as it is here where the work done in this chapter matters most.

Let C_{DLP}, C_{DHP} denote the costs of solving the DLP and DHP on an elliptic curve of size p, respectively. By Maurer's reduction, we have $C_{DLP} = N_{\mathfrak{DH}} \cdot C_{DHP} + N_{\mathfrak{M}}$, where $N_{\mathfrak{DH}}, N_{\mathfrak{M}}$ are respectively the number of calls to the DH-oracle and number of multiplications in G. Hence, for $N_{\mathfrak{M}} \ll C_{DLP}$ we get

$$C_{DHP} = \frac{C_{DLP} - N_{\mathfrak{M}}}{N_{\mathfrak{DH}}} \sim \frac{C_{DLP}}{N_{\mathfrak{DH}}}.$$

Since solving the DLP on an elliptic curve E is believed to take at least $\sqrt{|E|}$ steps [BSS99], in general, then setting

$$T_{DH} = \frac{\sqrt{|E|}}{N_{\mathfrak{DH}}},$$

we see that T_{DH} gives us a lower bound on the number of operations required to break the DHP, as long as we have $N_{\mathfrak{M}} \ll C_{DLP}$. Hence, it is the value of T_{DH} that gives the exact security of the DHP, given the best auxiliary elliptic curves that we can find.

The tightness of the security reduction is controlled by two values. The first being the number of field multiplications $N_{\mathfrak{M}}$, and second and most important is the value of T_{DH} for the reason put forth earlier. Tables 6.1 and 6.2 give the logarithms of these key values, namely $\lg N_{\mathfrak{M}}$ and $\lg N_{\mathfrak{DH}}$, for the curves in the SECG standard [SEC00]. They also give $\lg \sqrt{|E|}$, the logarithm of the (believed) generic minimum cost for solving an instance of the DLP on an elliptic curve E. The column headed *adv* gives the number of security bits gained on the previous results from [MSV04]. The last rows of the tables are detached to indicate that the

values are theoretical and that no auxiliary elliptic curves could be generated for them, mainly due to the sheer size of the numbers that needed to be factored.

secp curve	$\lg\sqrt{\lvert E\rvert}$	$\lg N_{\mathfrak{M}}$	$\lg N_{\mathfrak{DH}}$	$\lg T_{DH}$	*adv*
secp112r1	55.9	46.3	11.4	44.4	6.4
secp112r2	54.9	45.6	11.4	43.5	5.5
secp128r1	64.0	51.9	11.6	52.4	6.4
secp128r2	63.0	51.2	11.6	51.4	5.4
secp160k1	80.0	62.9	12.0	68.0	8.0
secp160r1	80.0	62.9	12.0	68.0	6.0
secp160r2	80.0	62.9	12.0	68.0	7.0
secp192k1	96.0	73.8	12.2	83.8	7.8
secp192r1	96.0	73.8	12.2	83.8	6.8
secp224k1	112.0	84.7	12.4	99.6	6.6
secp224r1	112.0	84.7	12.4	99.6	7.6
secp256k1	128.0	95.5	12.6	115.4	7.4
secp256r1	128.0	95.5	12.6	115.4	7.4
secp384r1	192.0	138.8	13.2	178.8	8.8
secp521r1	260.5	184.9	13.7	246.8	-

Table 6.1: Summary of results for curves of large prime characteristic

Now, given our estimates for the number of group operations and DH-oracle calls, we see that the smallest s for which $N_{\mathfrak{M}} \ll \sqrt{|E|}$ is $s = 3$. The reduction cost is then (see Conclusion on page 130 for general s)

$$\left(\frac{149}{6}\lg p + \frac{55}{8}\right)\mathfrak{DH} + \left((\frac{3}{2}\lg p + \frac{13}{2})(3p^{1/3}) + (\frac{3}{2}\lg p + \frac{511}{4})\lg p\right)\mathfrak{M}.$$

As an illustration of the advantage gained over the previous results presented in [MSV04], we consider the security of DHP for secp256r1: The DLP on this curve requires about 2^{128} computational steps, employing the currently known methods. Using our auxiliary elliptic curve, which can be found in Appendix A.1, we deduce that the DHP cannot be solved in less than $2^{115.4}$ computational steps, as opposed to 2^{108} from the previous paper. That is a gain factor of about $2^{7.4}$ over the previously reported value in [MSV04], see Table 6.1.

Since an amount of computation of about $2^{115.3} \approx 5 \cdot 10^{34}$ group operations is infeasible with today's computational power, one can draw the conclusion that a *secure implementation* of a protocol whose security depends on the intractability of

sect curve	$\lg \sqrt{\lvert E \rvert}$	$\lg N_{\mathfrak{M}}$	$\lg N_{\mathfrak{D}_5}$	$\lg T_{DH}$	*adv*
sect113r1	56.0	46.4	11.4	44.6	6.6
sect113r2	56.0	46.4	11.4	44.6	6.6
sect131r1	65.0	52.6	11.7	53.3	6.3
sect131r2	65.0	52.6	11.7	53.3	6.3
sect163k1	81.0	63.5	12.0	69.0	7.0
sect163r1	81.0	63.5	12.0	69.0	7.0
sect163r2	81.0	63.5	12.0	69.0	7.0
sect193r1	96.0	73.8	12.2	83.8	6.8
sect193r2	96.0	73.8	12.2	83.8	6.8
sect233k1	115.5	87.0	12.5	103.0	7.0
sect233r1	116.0	87.4	12.5	103.5	7.5
sect239k1	118.5	89.1	12.5	106.0	8.0
sect283k1	140.5	104.0	12.8	127.7	8.7
sect283r1	141.0	104.3	12.8	128.2	7.2
sect409k1	203.5	146.5	13.3	190.2	8.2
sect409r1	204.0	146.9	13.3	190.7	-
sect571k1	284.5	201.0	13.8	270.7	-
sect571r1	285.0	201.3	13.8	271.2	-

Table 6.2: Summary of results for curves of even characteristic

the DHP on the curve secp256r1 can safely be used, provided the DLP is really of the conjectured complexity.

Note that the SECG standard [SEC00] includes all the curves in the NIST [NIS00] and the most used ones in the ANSI [ANS99] standards, and hence it covers the most commonly used elliptic curves in practice.

6.3 Building the auxiliary elliptic curves

By the argument presented in the previous section, we need to construct elliptic curves whose order is a product of three coprime numbers of roughly the same size. That is $q_i \approx p^{1/3}$. Muzereau *et al.* [MSV04] used the Complex Multiplication (CM) technique to build auxiliary elliptic curves with smooth orders but this does not perform very well as p gets larger, due to the prohibitive precision then needed for the calculations. In our case, it proved to be computationally more efficient to generate random elliptic curves and then test if their sizes are of the required form.

Let us estimate the probability that a number in a large interval centred around p is a product of three co-primes of roughly the same size. This probability is bounded

below by the probability that a number n is a product of exactly three distinct *primes* of size roughly $n^{1/3}$, of which there are about $n^{1/3}/\log n^{1/3} = 3n^{1/3}/\log n$, by the prime number theorem. The number of products that can be made out of these is roughly $(3n^{1/3}/\log n)^3 = 27n/(\log n)^3$, so their proportion is

$$\frac{27n/(\log n)^3}{n} = \frac{27}{\log^3 n}.$$

Hence, a rough lower bound on the probability that we want is $27/\log^3 n$, which is not negligible, and hence we conclude that numbers that are products of three coprimes are not rare and can be found in expected polynomial time.[5]

For most cryptographic groups G from the SECG standard, auxiliary elliptic curve E of the form $y^2 = x^3 - 3x + b$ were successfully generated by finding a suitable value of b. When trying to generate the auxiliary elliptic curves, the main difficulty was to actually factor $|E|$. For large $|G|$, factorisation fails most of the time and another random value of b is tried without any success. This is the main reason for failing to produce the necessary data for the three curves `secp521r1`, `sect571r1` and `sect571k1`. However, two missing auxiliary elliptic curves from [MSV04], namely

---

[5]We should note that the following argument that was presented in [Ben05a, Ben05b], although mathematically correct, is not the probability needed for the analysis of our method.

> Let us estimate the probability that a number in a large interval centred around p is a product of three co-primes of roughly the same size.
>
> Given three randomly chosen (positive) integers, we first want to compute the probability that they are *pairwise* coprime. Let p be prime. The probability that p divides two of these integers but not the third is $3/p^2 \cdot (1 - 1/p)$ and the probability that p divides all of them at once is $1/p^3$. So, the probability that p is not a common divisor of any two of these integers is
>
> $$1 - \frac{3}{p^2}\left(1 - \frac{1}{p}\right) - \frac{1}{p^3} = 1 - \frac{3}{p^2} + \frac{2}{p^3}.$$
>
> Hence, the probability that three randomly chosen integers are pairwise coprime is
>
> $$\prod_{p \text{ prime}} \left(1 - \frac{3}{p^2} + \frac{2}{p^3}\right) = \prod_{p \text{ prime}} \left(1 - \frac{1}{p}\right)^2 \left(1 + \frac{2}{p}\right) \approx 0.2867474.$$
>
> The infinite product is clearly convergent but a closed form of its value could not be obtained by the author. The numerical approximation 0.2867474 was obtained using `PARI`, [BBB+98].
>
> For a large interval (m, n), the product should be taken only for $p \leq m - n$. Now, since $1 - 3/p^2 + 2/p^3$ is positive, strictly increasing approaching 1 from below, we deduce that the above estimate is a *lower bound* to the actual probability we want.

The above analysis estimates the probability that three randomly chosen numbers are relatively coprime, whereas we want to know the probability that a random number factors into a product of three coprime numbers.

`secp224k1` and `sect409r1` were successfully found. While the first seems to have just been forgotten, the second was certainly due to the difficulty of generating the auxiliary elliptic curves using the CM method.

6.3.1 The factoring procedure

To factor a number n into three co-prime numbers of roughly the same size, we first factor n completely then try to write the factorisation in the desired form. If this fails then n does not satisfy the property we want.

Let $n = \prod_{i=1}^{m} p_i^{e_i}$ be the prime factorisation of n. Write $n = \prod_{i=1}^{m} q_i$, where $q_i = p_i^{e_i}$ are coprime. We want to write $n = \hat{q}_1\hat{q}_2\hat{q}_3$ such that $\hat{q}_i$ are coprime and have the same size roughly. Clearly we need $m \geq 3$ to start with, so if $m < 3$ then we abort at this stage.

There are $\binom{m}{3} = m(m-1)(m-2)/6 \sim \frac{1}{6}m^3$ ways of grouping the q_i's into three groups. The search tree comprises $m+1$ levels, where the root node is $(1,1,1)$, and the nodes at each level ℓ are obtained by multiplying one component of the parent node by q_ℓ to get all possible groupings of $\prod_{i\leq\ell} q_i$ (at most 3 children per parent node).

To save on the cost of traversing the tree, we use a depth-first search and adopt an early abort strategy: Note that there is no need to pursue the subtree that has a root $\hat{q}_i \gg n^{1/3}$ for some $\hat{q}_i$, so if any component of a node is significantly greater than $n^{1/3}$ then we stop pursuing the current branch and backtrack. The first 5 levels of the tree are illustrated in Figure 6.1. See Algorithm 16 for the exact details.

Algorithm 16 Factorisations of an integer into three equi-size coprimes.
Input: An integer n, a parameter ϵ defining the interval $[B_\ell, B_u] = [n^{1/3-\epsilon}, n^{1/3+\epsilon}]$.
Output: A set Q of possible factorisations of n into three coprimes in $[B_\ell, B_u]$.

1: Let $n = \prod_{i=1}^{m} p_i^{e_i}$ be the prime factorisation of n
2: **if** $m < 3$ or $p_i^{e_i} > B_u$ for some $i \in \{1, \ldots, m\}$ **then**
3: **return** $\emptyset$
4: **end if**
5: Set $S = (p_1^{e_1}, p_2^{e_2}, \ldots, p_m^{e_m})$.
6: $q \leftarrow (1,1,1), Q \leftarrow \emptyset, depth \leftarrow 1$
7: tocoprimes$(S, q, Q, depth)$
8: **return** Q

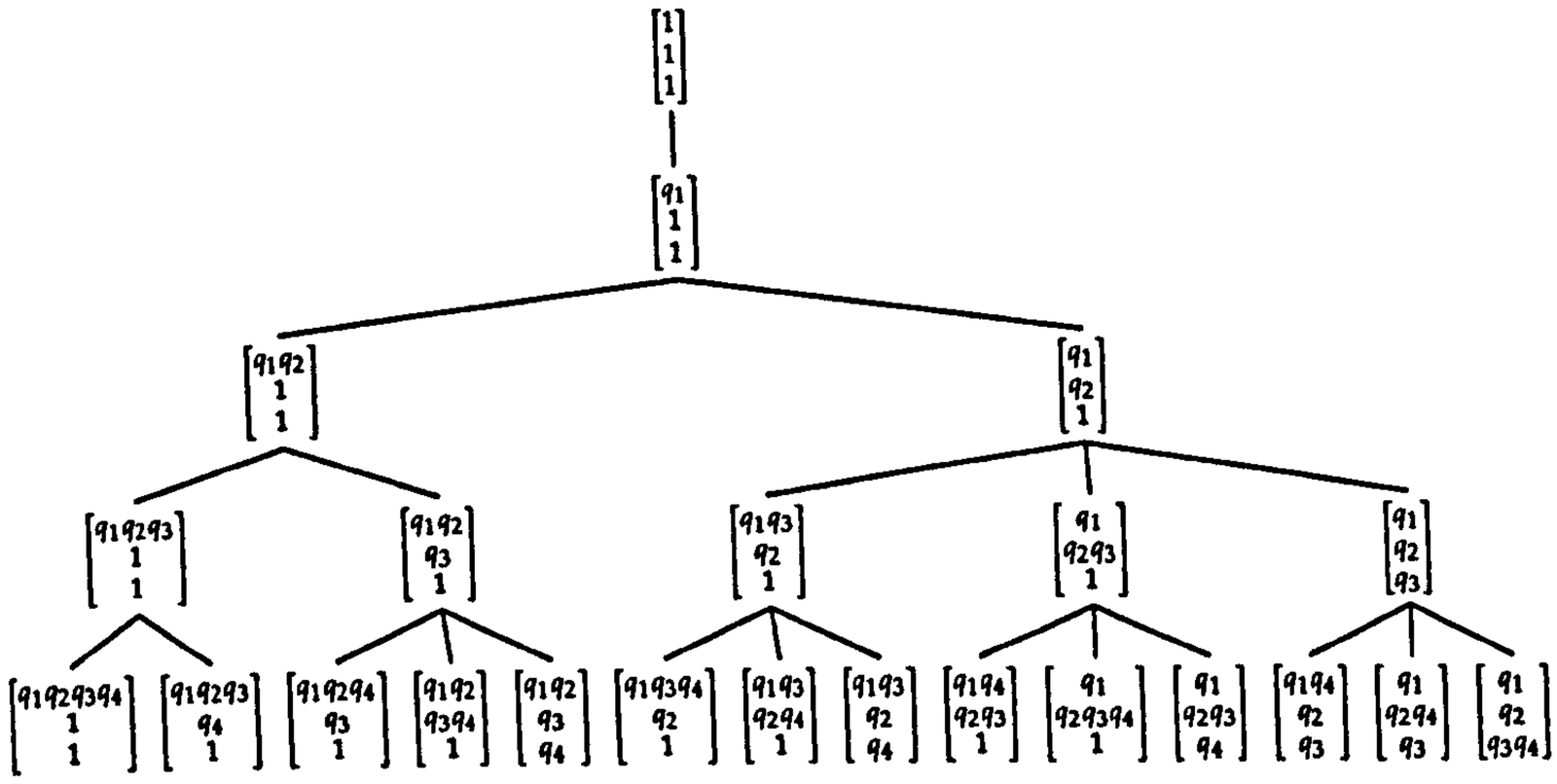


Figure 6.1: The first 5 levels of the factoring into 3 coprimes tree.

Algorithm 17 tocoprimes subroutine
Input: $S = (p_1^{e_1}, p_2^{e_2}, \ldots, p_m^{e_m}), q \in \mathbb{N}^3, depth \in \mathbb{N}, Q \subset \mathbb{N}^3$.

1: $\tilde{q} \leftarrow q$
2: **for** $j = 1, \ldots, \min(3, depth)$ **do**
3: $\quad \tilde{q} \leftarrow q$
4: $\quad \tilde{q}_j \leftarrow \tilde{q}_j S_{depth}$
5: $\quad$ **if** $\tilde{q}_j \leq B_u$ and $depth < m$ **then**
6: $\quad\quad$ tocoprimes$(S, \tilde{q}, depth + 1, Q)$
7: $\quad$ **end if**
8: **end for**
9: **if** $depth = m$ and $\tilde{q}_1, \tilde{q}_2, \tilde{q}_3 \geq B_\ell$ **then**
10: $\quad Q \leftarrow Q \cup \{\tilde{q}\}$
11: **end if**

6.4 Can we do better using Maurer's approach

Here, it is argued that not much improvement can be made using Maurer's reduction, as described in Algorithm 14.

Note that merely computing g^{x^3}, from a random $x \bmod p$, costs at least $2 \times \lg(p/2)$𝔇𝔎 on average. For $s = 3$, we find that the ratio of the estimated DH-oracle calls needed for the reduction to this lower bound is

$$\frac{\frac{149}{6}\lg p + \frac{55}{8}}{2\lg p} \sim \frac{149}{12} \approx 2^{3.6}.$$

Step 2 of the reduction algorithm is not independent from the first so its cost can be reduced further, but the third step does not seem to have any correlation with the previous steps. If we say that Step 3 costs at least one exponentiation, to

compute one of the $(|E|/q_j)Q$, where $q_j \sim |E|^{1/3}$, then the ratio drops to

$$\frac{\frac{149}{6}\lg p + \frac{55}{8}}{(2+2/3)\lg p} \sim \frac{149}{16} \approx 2^{3.2}.$$

If we further assume that $(|E|/q_j)Q$ are all independent for $j = 1,2,3$ then the ratio drops to $(149/6)/(2+3\times 2/3) \approx 2^{2.6}$.

Hence, it turns out that about 3 bits of security is all that can be hoped for above our result!

6.5 Concluding remarks

Assuming the DLP is an exponentially hard problem, we have shown that the Maurer-Wolf reduction with naive search yields a concrete security assurance for the elliptic curves recommended by the current standards, for which we could generate the auxiliary elliptic curves. We have found two new auxiliary elliptic curves, missing from [MSV04], namely `secp224k1` and `sect409r1`. It remains open to find auxiliary elliptic curves for the curves `secp521r1`, `sect571r1` and `sect571k1`. These will have sizes larger than 500 bits, which presents the current factoring algorithms with a big challenge.

Appendix A.1 starting at page 147 lists the auxiliary elliptic curves that we constructed to give almost the tightest possible (Maurer) reduction.

Chapter 7

Conclusion

"وَقِيلَ الْحَمْدُ لِلَّهِ رَبِّ الْعَالَمِين"

— "And it was said (in the end): Praise and thanks be to God the lord of the worlds",

The Quran

7.1 Review of results

In this thesis, we have addressed three instances of cryptographic problems and improved on their current performance using both traditional and new techniques.

In Chapter 4, we allowed the parameter sizes of the RSA cryptosystem to get too large, as suggested by NIST, and proposed new ways to implement the modular arithmetic. We used the wooping error detection technique in a novel way to allow us to achieve better practical performance than would have been possible otherwise.

In Chapter 5, we saw how important is it to take into consideration the exact cost analysis of proposed cryptographic primitives, as asymptotic security results may be misleading and lead to inefficient or insecure systems when instantiated with practical parameter sizes. We tried to remedy the GGH hash function proposal via relaxation of storage requirements and modification of the compression function to break its linearity. This cost us to lose the desirable (asymptotic) provable security property, but has allowed us to make a concrete proposal that we hope will be made secure or at least serve as a starting point for future proposals.

In Chapter 6, we saw a different type of efficiency tightening exercise. This time we addressed efficiency in theoretical arguments of computational reduction between cryptographic hard problems. We used an alternative representation of the elements to decrease both the time and space requirements by a logarithmic factor.

7.2 Open problems and future research

The field of cryptologic research is very young with many open problems, and the topics studied in this thesis are no exception to this rule. Here, we list some of the issues that are directly related to this thesis that we wish to study and hopefully solve.

It is a natural question to ask whether we can improve on the speedups we have already gained with our proposed approaches. The answer was negative for the equivalence between the DLP and DHP chapter where we showed, in Section 6.4, that not much can be saved for the range of parameters we were interested in unless we use a whole different reduction method altogether.

There is however at least one type of DH-oracles that is not covered by our analysis in Chapter 6. Consider the special DH-oracle which, given a group element $h = g^a \in G$ as input, returns $h^x = g^{ax}$ where x is a fixed secret element of the group in question. The problem associated with this type of DH oracles is referred to as *Static Diffie-Hellman Problem* (SDHP), and asks for the recovery of the secret x. A reduction for this oracle type is not yet known despite it being a more realistic model of, for example, a compromised smart card that holds the secret x; and hence some research needs to be directed towards this problem. This is because we do not (currently) know how to use such an oracle to perform arithmetic in the implicit form (on the exponents) and hence we cannot use the Maurer's reduction. Brown and Gallant showed in [BG04] that such an oracle can be used to recover the secret x in less than $\sqrt{p}$ calls, namely $\sqrt[3]{p}$ oracle calls using space $O(\max\{\sqrt{p/d}, \sqrt{d}\})$ where $p = \#G$ is prime and d is a divisor of $p-1$. This running time applies only in the case when $p-1$ has a factor d of order $p^{1/3}$, which is true with significant probability,

and when at least d oracle queries are made. Cheon also describes a similar attack in [Che06] and further extends it to the case where $p + 1$ has a small divisor by exploiting the structure of $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$. We speculate that it might be possible to generalise these attacks to work over higher extensions of $\mathbb{F}_p$ or over elliptic curves but it seems that the restrictive nature of the static DH oracle limits us so much and makes this infeasible [Che06].

One interesting open problem relating to the design of LASH is to find good values for the initial vector IV to circumvent similar attacks to the one on $\mathtt{IV} = 0$. However, since any compression function based solely on lattices will be linear and thus will suffer from sub-exponential attacks using a generalised birthday attack, the chapter on building practical hash functions based on lattice problems seems closed, unless a completely different design strategy is used. The source of difficulty stems from the fact that it seems hard, if not impossible, to avoid these attacks without losing the lattice structure.

The chapter on speeding up RSA arithmetic, however, touches on a fertile subject and has many open questions and possible improvements to be investigated. We will now suggest some possible ways of improving on the (software) solutions given in Chapter 4. One can also explore the possible hardware improvements that can be exploited but this is beyond our scope in this thesis.

7.2.1 Shamir's RSA for paranoids

Shamir suggested in [Sha95] to use "Unbalanced RSA" where the modulus N is a product of a very large prime q and a smaller prime p, e.g. for 5000 bits he suggests using p of size 500 bits and q of size 4500 bits.

The key idea here is that RSA is usually only used for key encapsulation, so we can take p to only be as big as the key-size while we let q as big as it needs to be such that the product $pq = N$ is of the required size. Now let $\kappa = k^e \bmod N$ be the encapsulation of a key $k < p$, then for decapsulation we can see, using the Chinese Remainder Theorem, that we only need to compute $k = \kappa^{d \bmod p-1} \bmod p$ since we already know that $0 < k < p < q$ i.e. there is no need to compute $k = \kappa^{d \bmod q-1} \bmod q$ or any other further computation. For the optimisation of encapsulation, Shamir

proposes using $e = 20$ as then we can compute k^{20} as $(k^{10})^2$ mod N which only costs 10 *integer* multiplications and 1 modular multiplication provided $k < p < q^{1/10}$, and the "wraparound effect is similar to the squaring operation of full size numbers in Rabin's scheme."

This variant of RSA would clearly be of great interest if 15,360-bit RSA is to be used in practice for key encapsulation. Further research needs to be done regarding the security of this proposal and the possible padding schemes (like RSA-OAEP) that need to be used to avoid attacks similar to those presented in [GGOQ98].

7.2.2 Using convolutions to speed up Montgomery reduction

Roughly speaking, convolutions allow us to compute the sum of the upper and lower halves of a product. In this section, we will show how this may save us computation time and then describe how to actually compute these convolutions.

Let us recall the description of the Montgomery reduction steps (see Algorithm 3 on page 41).

1: $u \leftarrow (-m^{-1})z \bmod R$
2: $x \leftarrow (z + um)/R$
3: **If** $x \geq m$ **then**
4: $\quad x \leftarrow x - m.$
5: **end if**
6: Return x

As we have pointed out before, $(z + um)/R$ is simply the upper part of $z + um$ i.e. $(z + um)_u$. The left diagram of Figure 7.1 illustrates the operation $z + um$ graphically.

We will now show a nice modification that will allow us to compute $(z + um)/R$ a bit faster. Note that if we compute $(um)_u + (um)_\ell$ instead of the full um then the lower half of the addition $z + ((um)_u + (um)_\ell)$ is going to be equal to $(um)_u$. Hence, we can read the value of $(um)_u$ straight away and add it to z_u to get an approximation to $(z + um)_u$, which will be off by at most 1 (carry value). This then can either be corrected by wooping or other techniques.

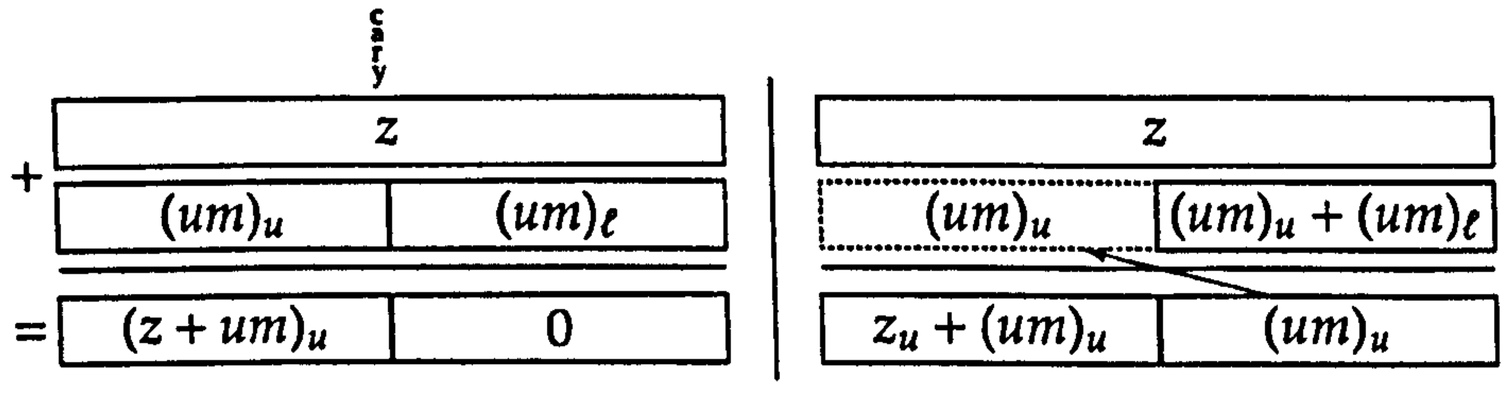
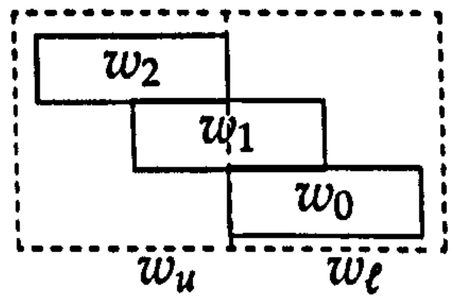


Figure 7.1: Using convolution to compute $(z + um)_u$.

Computation of convolutions

There is a general technique to compute convolutions using FFTs when the number of words is a power of 2, but we will first construct special methods based on the Karatsuba and Toom-3 multiplication algorithms then comment on the possible use of FFT in our case.

Karatsuba-like convolution. Let $f(x) = (a_0 + a_1x)(b_0 + b_1x) = w_2x^2 + w_1x + w_0$.



Note that $f(1) - f(-1) = 2w_1$ and $f(1) + f(-1) = 2(w_0 + w_2)$. So we can construct

$$(uv)_\ell + (uv)_u = (w_0 + w_2) + b^t(w_1)_\ell + (w_1)_u + \textit{carries}.$$

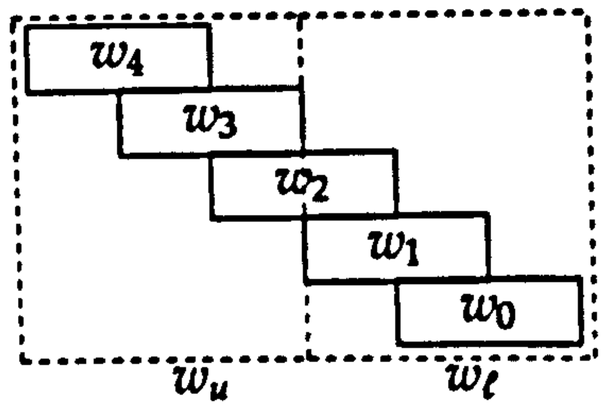
This will only cost two half-size multiplications to compute $f(\pm 1)$, and hence a total of

$$2\mathcal{K}(n/2) + 4n \approx \frac{2}{3}\mathcal{K}(n). \tag{7.1}$$

[Justification: $2(n/2)^{\lg 3} = (2/2^{\lg 3})n^{\lg 3} = (2/3)n^{\lg 3}$.]

Toom3-like convolution. Write $a(x) = a_0 + a_1x + a_2x^2, b(x) = b_0 + b_1x + b_2x^2$ and

$$a(x)b(x) = w_0 + w_1x + w_2x^2 + w_3x^3 + w_4x^4.$$



For $w_u + w_\ell$ we are interested in computing

$$(w_4 + w_1)b^t + (w_3 + w_0) + (w_2)_\ell b^{2t} + (w_2)_u + carries.$$

Let $\zeta \neq 1$ be a root of $x^3 - 1$ (e.g. $\zeta = \exp(2i\pi/3)$). First compute

$$\begin{aligned}
a(\zeta^0)b(\zeta^0) &= w_0 + w_1 + w_2 + w_3 + w_4 &= A \\
a(\zeta^1)b(\zeta^1) &= w_0 + w_1\zeta + w_2\zeta^2 + w_3\zeta^3 + w_4\zeta^4 &= B \\
a(\zeta^2)b(\zeta^2) &= w_0 + w_1\zeta^2 + w_2\zeta^4 + w_3\zeta^6 + w_4\zeta^8 &= C
\end{aligned}$$

i.e.

$$\begin{aligned}
A &= (w_0 + w_3) + (w_1 + w_4) + w_2 \\
B &= (w_0 + w_3) + (w_1 + w_4)\zeta + w_2\zeta^2 \\
C &= (w_0 + w_3) + (w_1 + w_4)\zeta^2 + w_2\zeta
\end{aligned}$$

Solving for $(w_0 + w_3), (w_1 + w_4)$ and w_2, we get

$$\begin{aligned}
w_0 + w_3 &= (2 + \zeta)(B - C\zeta)/3 \\
w_1 + w_4 &= (2 + \zeta)(A - B - (B - C)\zeta)/3 \\
w_2 &= (1 - \zeta)(A - B)/3
\end{aligned}$$

The Karatsuba convolution is very easy to implement but the Toom-3 variant seems harder to implement as it requires us to deal with complex numbers. It would

be interesting to see how one can handle this issue without introducing too much computational overhead.

FFT based multiplication. The authors of [PG05] suggest using *cyclic convolutions* instead of half products and achieve, in [PG06], a complexity of $\Theta(2.5n \log n)$ for a reduction algorithm with the use of *negacyclic convolutions*, but it is unlikely that these will beat our proposal around the 15K bit operands case.

It is argued in [Gar07] that the Schönhage method becomes as efficient as the Karatsuba and Toom-3 methods at about $2^{17} = 131,072$ bits, which is close to the value of the generic FFT multiplication threshold used in the GMP library namely 30 × Toom-3's threshold: 30 × 128 = 3840 words (122,880 bits). These sizes are too high for our purpose. The reader may be interested in having a look at [Zur94] to a see a report on concrete implementation of a wide range of multiplication methods (but run on an old machine).

7.2.3 Cache oblivious Montgomery and Barrett methods

If the cache size is too small then the recursive nature of the used multiplication methods and the large sizes of the operands may cause cache misses and hence slow the computation considerably.

One possible way of circumventing this problem is to use an iterative version of the multiplication algorithms. For a description of an iterative version of the Karatsuba method see [LLH03].

Conclusion

Appendix A

Appendices

A.1 The auxiliary elliptic curve groups

A.1.1 Elliptic curve domain parameters over $\mathbb{F}_p$

secp112r1
$|G| = 4451685225093714776491891542548933$
$b = 2281028298640880380471050241629229$
$|E| = 161721374756 \cdot 170510910317 \cdot 161437658771$
secp112r2
$|G| = 1112921306273428674967732714786891$
$b = 2061835755930385486536405010948 54$
$|E| = 105310592296 \cdot 103373879227 \cdot 102230759539$
secp128r1
$|G| = 340282366762482138443322565580356624661$
$b = 29638221667210512794844809568104407 6642$
$|E| = 7551279841752 \cdot 6513487018025 \cdot 6918394582717$
secp128r2
$|G| = 850705916906205346039557219268136 60579$
$b = 7301954261820617358230137714654813 3543$
$|E| = 4222485329260 \cdot 4376586107537 \cdot 4603369401979$
secp160k1
$|G| = 1461501637330902918203686915170869725397159163571$
$b = 1014269469389219214184903107646149695236127481640$
$|E| = 1113082721280921 5 \cdot 1139497624790683 7 \cdot 1152281106160626 7$
secp160r1
$|G| = 1461501637330902918203687197606826779884643492439$
$b = 1231565154230325865757423073063591837019188457168$
$|E| = 1117488549446764 5 \cdot 1100894918154088 9 \cdot 1187983359875557 9$
secp160r2
$|G| = 1461501637330902918203685083571792140653176136043$
$b = 1987871000780349598609964130362172069236350775 8$
$|E| = 1057372552687927 2 \cdot 1152057259706567 9 \cdot 1199767818043422 7$
secp192k1
$|G| = 6277101735386680763835789423061264271957123915200845512077$
$b = 1094708638413029664629646177364452405008715587623144058105$

$|E| = 163529621162214361 26 \cdot 177054994115072243 87 \cdot$
21679764265977655387

secp192r1

$|G| = 6277101735386680763835789423176059013767194773182842284081$

$b = 7339867319969617520190619107777595180087882698523301357 4$

$|E| = 1729427452043899916 4 \cdot 19491494149529285201 \cdot$
18621372472744345117

secp224k1

$|G| = 26959946667150639794667015087019640346510327083120074 54\backslash$
8994958668279

$b = 24618590432167307909930264143550961204039679464315847760\backslash$
586750945971

$|E| = 25996959705011679445066 \cdot 3344835872642172095654 1 \cdot$
$3100428036195577097238 1$

secp224r1

$|G| = 26959946667150639794667015087019625940457807714424391 72\backslash$
1682722368061

$b = 86181493252759602511614871186111585563413066847517370546 5\backslash$
8821880904

$|E| = 2934361314174457002464 4 \cdot 3179841463232218870759 3 \cdot$
28893487975414890420151

secp256k1

$|G| = 11579208923731619542357098500868790785283756427907490 4\backslash$
$3826051631415181614943 37$

$b = 58603723116421395918689089913861383686268511262358322 04\backslash$
$688066666346673735409 9$

$|E| = 47494383239999767419320745 \cdot 45175228939925617688211569 \cdot$
$53967993991985944506666061$

secp256r1

$|G| = 11579208921035624876269744694940757352999695522413576 0\backslash$
$3424222590610685120443 69$

$b = 47655894101463316762236526132016393253057270840001423 83\backslash$
9782911257030924437529

$|E| = 50851524730203743853228640 \cdot 55497037692343386526156881 \cdot$
$41030339309908399787973083$

secp384r1

$|G| = 39402006196394479212279040100143613805079739270465446 6\backslash$
$6794690527962765939911326356939895630815229491355443365394 2643$

$b = 89890103691693584367418476819795701055812436905742082 63\backslash$
$2695560596504661582700569954858820254069479866825873678896 24$

$|E| = 33986987048189154740054658522517921329 0 \cdot$
$34957975980158220309922293105381374555 3 \cdot$
$33163425973966331908531830503110509205 9$

secp521r1

Not available due to hardness of factoring.

A.1.2 Elliptic curve domain parameters over $\mathbb{F}_{2^m}$

sect113r1

$|G| = 5192296858534827689835882578830703$
$b = 98763709954301375702954581001609 8$
$|E| = 178524038025 \cdot 170996556499 \cdot 170088694619$
sect113r2
$|G| = 5192296858534827702972497909952403$
$b = 45837693630171016082451877084589 01$
$|E| = 173146840968 \cdot 166401825973 \cdot 180213306239$
sect131r1
$|G| = 136112946768375385389393275685365560653$
$b = 125832860520930687507071669649567519 6119$
$|E| = 11939631029912 \cdot 10689621208893 \cdot 10664640354101$
sect131r2
$|G| = 136112946768375385387953504341281286 7983$
$b = 35823234234411939205840423080645359 4114$
$|E| = 11466564749342 \cdot 10619089660293 \cdot 11178378760169$
sect163k1
$|G| = 5846006549323611672814741753598448348329118574063$
$b = 17767337697332384777035473627178295668998324 8537$
$|E| = 1824780453881666 1 \cdot 1943646869894155 1 \cdot 1648281233285216 9$
sect163r1
$|G| = 5846006549323611672814738465098798981304420411291$
$b = 15874048673063598988848193391540827816535852093 24$
$|E| = 1786992089997791 2 \cdot 1755136344494492 3 \cdot 1863913732179538 1$
sect163r2
$|G| = 5846006549323611672814742442876390689256843201587$
$b = 29562833239804228892914784773703209533557315769 40$
$|E| = 1820071960355955 9 \cdot 1756808627444010 1 \cdot 1828295048008093 1$
sect193r1
$|G| = 6277101735386680763835789423269548053691575186051040197193$
$b = 35338895987916163832451188982915353767627436600288649159$
$|E| = 16547960255111188472 \cdot 19478515037898861263 \cdot$
19474165359321867611
sect193r2
$|G| = 6277101735386680763835789423314955362437298222798401438 29$
$b = 44175595756811211606663340113336051184739649262973176442 9$
$|E| = 19387762096509288342 \cdot 18577800791543661067 \cdot$
17427583967788534019
sect233k1
$|G| = 3450873173395281893717377931138512760570940988862252126\backslash$
328087024741343
$b = 2512214920549173559513768839048670735137036898029798853 8\backslash$
30832766551245
$|E| = 1554030093442781185542 32 \cdot 1533857407177146667391 25 \cdot$
$14477200391328782477823 1$
sect233r1
$|G| = 6901746346790563787434755862277025555839812737345013555\backslash$
379383634485463
$b = 7040938164755706341740819287052251842563468263172882818 2\backslash$
1773151878529

$|E| = 20679961703033668255541 6 \cdot 19518549023892523058088 9 \cdot$
$17098646513459315515294 9$

sect239k1

$|G| = 22085588309729804119791218759286481494821656132170984 88\backslash$
$8748021921536221 3$

$b = 27650235244228507853355450435057293014412082341226059038\backslash$
361077531582683

$|E| = 54381492548936524083766 8 \cdot 61057636259911441694809 7 \cdot$
$66514734518374326199148 5$

sect283k1

$|G| = 38853377844514581418389238136470378132848117337930613 24\backslash$
$29587499752981582970442260387 3$

$b = 28183552298654367145273437136771989603707301993060462481\backslash$
$46777682199067799961811453900$

$|E| = 16292450803352497273678817784 \cdot$
$152013619523505578126840970 49 \cdot 15687721231974421411325545219$

sect283r1

$|G| = 77706755689029162836778476272940756265696259243769048 89\backslash$
$10919652677004427778737869287 1$

$b = 71767445486180876851805109646321526052188997926851304655\backslash$
$95965436250552932458637035413$

$|E| = 16932408152570400028840713015 \cdot$
$19857620455536755941661666843 \cdot 23110686327095779427460999989$

sect409k1

$|G| = 33052798439512429947595765401638551991420234148214060 96\backslash$
$42324395022880711289249191050673258457777458014096366590617 73\backslash$
1358671

$b = 13877074019970923581077302466204224976964264102344770827\backslash$
$43704801735884537140792236509289413698528330836985031075479 69\backslash$
459853

$|E| = 54923628603232455334113678631129360414184 \cdot$
$62030988940606152064529997029577410596573 \cdot$
$97015317302467505937973376689033052671801$

sect409r1

$|G| = 66105596879024859895191530803277103982840468296428121 92\backslash$
$84648798304157774827374805208143723762179110965979867288366 56\backslash$
7526771

$b = 13817711446362728360145301111436486530925505507402770142\backslash$
$25867302825624633843006405426647064207268626654704613434090 3\backslash$
3831354

$|E| = 87268656040437200019781889318456334448900 \cdot$
$81063003278915230074335552542219354685229 \cdot$
$93445254974986510684197220630040488205129$

sect571k1

Not available due to hardness of factoring.

sect571r1

Not available due to hardness of factoring.

A.2 Trace of a single execution of LASH-160

To help with finding bugs in implementations of LASH, we give a trace of the internal variables when hashing the three-byte ASCII string "abc".

```
First compression function iteration.

   r:        00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
             00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00

   s:        61 62 63 80  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
             00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00

Init t:      61 62 63 80  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
             00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
(i=321)      ee 72 5e a4  bf bb f6 55  1f a1 48 1c  ef 02 55 1b  16 75 30 2d
             3f eb 68 aa  c6 e3 1a 2d  eb 14 2f d0  e5 6a 2c c1  ff 07 77 3c
(i=322)      51 ff 6e 9f  e3 7a b1 4b  74 c0 e9 64  0b f1 57 70  31 8b a5 5d
             6c 2a 53 12  70 a9 fd 47  18 ff 43 ff  b5 4f 96 ed  c0 06 7e b3
(i=327)      36 0b 20 47  ec dd 3e 5b  6f e4 a8 1f  01 46 76 11  79 a7 94 5f
             c1 45 69 87  a0 d6 3c 32  80 a9 09 e2  cf 7c 81 01  ef d6 63 1d
(i=329)      da 32 05 53  9e 85 47 be  fc f4 a3 43  c0 01 6c 66  98 48 dc 7b
             b0 47 be a2  b6 4b 6c 5f  bf 94 71 8c  95 5f 9b 2e  da ea 92 ed
(i=330)      9a d6 2c 38  aa 37 ef c7  5f 81 b3 3e  e4 c0 27 5c  ed 67 7d c3
             cc 36 c0 f7  d1 61 e1 8f  ec d3 5c f4  3f 25 7e 48  07 d5 a6 1c
(i=334)      28 53 00 c1  6a db 16 ac  6b 33 5b 47  47 4d 37 57  11 26 38 b9
             21 55 61 3f  ed 50 e3 e4  07 e9 d1 24  6c 64 69 b0  b1 9b 89 36
(i=337)      f1 cc ba 4f  e7 af 9f 6c  0f 5a 40 53  f9 f5 40 ba  9e 36 33 dd
             e0 10 57 94  0c f1 2b 00  f6 eb 26 3f  82 d9 99 dd  f0 86 f1 e0
(i=338)      c0 95 33 09  75 2c 73 f5  cf fe 67 38  05 a7 e8 c3  01 c3 43 d8
             04 cf 12 8a  61 10 cc 48  12 da 28 94  9d ef 0e 0d  1d c5 dc 48
(i=342)      c1 d5 aa fa  44 f5 ec af  5d 7b 3b c1  c5 4b 0f a8  0d 75 eb e1
             67 5c 22 85  85 cf 87 3e  67 f9 c9 dc  b9 de 10 62  38 db 51 78
(i=343)      cd d6 ea 71  35 c4 b5 28  17 09 b8 95  4e 0b b3 cf  f2 81 9d 89
             70 bf af 95  80 f3 46 f9  5d 4e e8 7d  01 fa ff 64  8d f6 67 ed
(i=344)      49 e2 eb b1  ac b5 84 f1  90 c3 46 12  22 94 73 73  19 66 a9 3b
             18 c8 12 22  90 ee 6a b8  18 44 3d 9c  a2 42 1b 53  8f 4b 82 03

Final compression function iteration.

   r:        49 e2 eb b1  ac b5 84 f1  90 c3 46 12  22 94 73 73  19 66 a9 3b
             18 c8 12 22  90 ee 6a b8  18 44 3d 9c  a2 42 1b 53  8f 4b 82 03

   s:        18 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
             00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00

Init t:      51 e2 eb b1  ac b5 84 f1  90 c3 46 12  22 94 73 73  19 66 a9 3b
             18 c8 12 22  90 ee 6a b8  18 44 3d 9c  a2 42 1b 53  8f 4b 82 03
(i=1)        a6 13 4f 38  99 14 9e 7c  66 66 08 cb  08 f6 d1 06  2b d8 b4 4a
             27 62 0f e5  62 db 26 d6  17 5a 3e 32  95 d2 c6 42  67 df 3d 2c
(i=4)        74 8d 4c 8d  ca 78 25 69  c5 80 93 a1  ab b8 8a ec  8d 36 47 5c
             99 6d 1e f4  fc d8 e9 a8  04 16 5c 31  ab d3 5c 35  f7 8a 2c 04
(i=7)        77 e3 84 5b  44 75 7a 9a  29 07 80 00  c5 43 60 8f  4f ef 2d be
             f7 00 30 66  07 e7 f8 42  01 d9 2e 1e  67 f1 5b 4b  f8 20 1f 94
(i=8)        ec e6 da 93  12 ef 77 ef  5a 6b 07 ed  24 5d eb 65  f2 b1 e6 a4
             59 5e c3 78  79 f2 07 51  9b d6 f1 f0  54 ad 79 4a  0e 21 b5 87
(i=9)        57 5b dd e9  4a bd f1 ec  af 9c 6b 74  11 bc 05 f0  c8 54 a8 5d
             3f c0 21 0b  8b 64 12 60  aa 70 ee b3  26 9a 35 68  0d 37 b6 1d
(i=10)       bc c6 52 ec  a0 f5 bf 66  ac f1 9c d8  98 a9 64 0a  53 2a 4b 1f
             f8 a6 83 69  1e 76 84 6b  b9 7f 88 b0  e9 6c 22 24  2b 36 cc 1e
(i=14)       69 52 eb b8  05 60 34 69  02 29 6a 52  95 fe 95 6e  da 17 aa 39
             83 7c 26 2b  d7 5c e6 c9  4c 91 fa bb  f8 7b bc 21  ee 08 b9 da
(i=16)       ec c1 98 44  9e 2c 99 d4  77 2c c0 8a  63 78 92 c3  0b 7b 31 26
             e2 96 b1 01  7a 1e 9f af  ae ef 8d cd  6a 86 cb 30  88 05 7c ac
(i=17)       01 44 07 f1  2a c5 65 39  e2 a1 c3 e0  9b 46 0c c0  60 ac 95 ad
             cf f5 cb 8c  50 c1 61 68  94 51 eb 60  7c f8 d6 3f  97 9f 79 6f
(i=18)       9a 59 8a 60  d7 51 fe 05  47 0c 38 e3  f1 7e da 3a  5d 01 c6 11
             56 e2 2a a6  db 97 04 2a  4d 37 4d be  0f 0a 48 4a  a6 ae 13 6c
```

```
(i=20)   18 b2 23 75  5a c0 ab 91  e0 d8 9d 4e  66 81 30 72  2b 7b c3 66
         87 46 b1 93  3a b1 8f 00  f0 f9 06 a4  71 68 db 5c  18 b9 22 7b
(i=22)   27 0f a1 ce  f3 d5 2e 00  8d 64 36 1a  cb ec a5 75  81 b3 91 e0
         84 9b e2 f7  c1 9e ee 1a  7b cf a9 66  2a 4e 3d ba  ab cb 94 86
(i=23)   75 1e fe 4c  4c 6e 43 83  fc 11 c2 b3  97 51 10 ea  84 09 c9 ae
         fe 98 37 28  25 25 db 79  95 5a 7f 09  ec 07 23 1c  09 5e a6 f8
(i=24)   ea 6c 0d a9  ca c7 dc 98  7f 80 6f 3f  30 1d 75 55  f9 0c 1f e6
         cc 12 34 7d  56 89 62 66  f4 74 0a df  8f c9 dc 02  6b bc 39 0a
(i=26)   a8 03 82 f7  d9 24 5a f1  18 95 f2 ae  dd a9 0e 21  5e 77 94 e9
         22 4a 02 f7  53 de 93 ca  7b 61 69 f9  1a 9f 7f c4  24 a2 9b 68
(i=27)   d6 c1 19 6c  27 33 b7 6f  71 2e 07 31  4c 56 9a ba  2a dc ff 5e
         25 a0 3a c5  cd db e8 fb  df e8 56 58  34 2a 55 67  e6 5b 81 ca
(i=31)   d7 71 7d 8c  55 f1 4e e4  bf 3d 64 af  a5 ef af 3d  99 89 8b f7
         f1 05 a5 3a  d0 31 20 c9  59 e5 ab 89  98 b1 42 c6  00 e6 57 6d
(i=32)   2b 72 2d f0  75 1f 0c 7b  34 8b 73 0c  23 48 48 52  1c f8 38 83
         8a d1 0a a5  45 34 76 01  27 5f a8 de  c9 15 c9 b3  5f 00 e2 43
(i=34)   1f 15 81 f1  25 83 2c a9  f2 22 e8 5a  32 a5 c6 ab  b5 0d bb f2
         37 5d a3 71  aa 9f eb 04  7d 97 76 58  c6 6a fa 17  e6 ed 41 5d
(i=36)   46 c5 75 94  79 84 dc 0d  12 50 a6 f1  a7 f3 d5 08  33 66 54 07
         ba cc 50 fd  43 6b 50 6f  f2 9a cc 90  94 e4 f7 6c  17 51 c8 4a
(i=37)   68 ec 25 88  1c d8 dd bd  76 70 d4 af  3e 68 23 17  90 e4 ad a0
         cf 4f bf aa  cf 04 1c d4  5d 0f cf e6  cc b2 71 69  6c 82 2c d1
(i=40)   75 cc 53 aa  43 88 d1 60  ca 71 84 13  5e 96 e1 ae  05 32 bc fd
         4d a8 58 bf  52 73 c9 60  f6 db 34 51  41 b5 c7 a1  3a fc 29 26
(i=42)   9e d1 60 8a  71 aa f8 10  be 14 d8 14  0e fa 01 dc  c3 c9 31 4b
         5c 05 d6 18  eb 88 4c cf  a3 67 cd 1d  a6 20 3c a4  90 34 f7 a0
(i=43)   44 fa 65 97  51 d8 1a 37  6e 08 7b 68  0f aa 65 fc  f1 87 c8 c0
         aa 14 33 96  44 21 61 52  12 14 59 b6  72 85 a7 19  93 8a 2f 6e
(i=45)   1e b4 0b c0  56 e5 fa 65  90 2f 2b 5c  b2 fe 66 ac  55 a7 f6 7e
         41 89 81 a5  a1 9f ba eb  27 97 c8 63  fe 1e 73 7e  fe ff 32 c4
(i=47)   60 ea e5 7a  fc 0e ff 72  70 5d 4d 83  62 f2 09 00  56 57 5a 9e
         6f 47 18 1a  ef ae 17 69  80 30 dd e6  6d cb ff 17  ca 64 9d 39
(i=48)   6e 2c 1b 54  b6 b4 28 77  7d 3d 7b a5  89 a2 fd a3  aa 58 0a 02
         8f 75 d6 b1  64 fc 26 c6  fe 89 76 fb  f0 3a ac a3  63 30 02 a4
(i=53)   94 81 28 51  a5 c2 6a ad  57 f7 21 ce  8e af dd d1  cc 7f ba f6
         32 c9 d7 61  c8 1c 54 84  95 fe c4 0a  4d b8 05 3c  78 b3 71 51
(i=56)   37 92 80 77  fa cf 67 9c  65 39 57 a8  48 55 06 d6  d9 5f e8 18
         59 79 cb 04  1c 1d 04 e8  b5 2c 82 a1  c2 06 14 99  f6 0c 0a 66
(i=57)   ff 35 91 cf  20 24 74 99  54 47 99 de  22 0f ac ff  de 6c c8 46
         7b a0 7b f8  bf 71 05 98  19 4c b0 5f  59 7b 62 a8  53 8a 63 ff
(i=58)   78 fd 34 e0  78 4a c9 a6  51 36 a7 20  58 e9 66 a5  07 71 d5 26
         a9 c2 a2 a8  b3 14 59 99  c9 b0 d0 8d  17 12 d7 f6  62 e7 e1 58
(i=59)   7a 76 fc 83  89 a2 ef fb  5e 33 96 2e  9a 1f 40 5f  ad 9a da 33
         89 f0 c4 cf  63 08 fc ed  ca 60 34 ad  45 d0 6e 6b  b0 f6 3e d6
(i=63)   fb ad 70 41  8b 1b b7 9e  6f 8b bc 83  a7 1c 2f 6d  ef d0 b4 ed
         2f 19 c9 dc  43 36 1e 14  7a 54 d7 01  46 80 d2 8b  de b4 d5 4b
(i=64)   b1 2e a7 b5  49 1d 30 66  12 9c 14 a9  fc 29 2c 5c  fd 12 ea c7
         e9 bf f2 e1  50 16 4c 36  a1 04 cb a4  9a 81 82 ef  fe e2 93 e2
(i=67)   2a 21 28 6b  ca 54 a4 24  14 15 dc 4c  0d 81 52 b1  0a 0f d9 d5
         2b f5 cc 9b  f6 3f 51 43  81 32 ed cb  4a 75 25 43  ff 92 f7 02
(i=72)   f0 1d a5 79  74 cd 97 a5  ca 96 13 c0  cb 83 cb 79  ad 20 31 fb
         80 02 c9 8a  04 81 87 1d  3b d8 16 d0  57 55 53 65  26 42 eb a5
(i=73)   82 e3 a1 f6  82 77 10 98  4b 4c 94 f7  3f 41 cd f2  75 c3 42 53
         a6 57 d6 87  f3 8f c9 53  15 92 bc f9  5c 62 33 93  48 69 9b 99
(i=78)   09 18 6d c3  57 09 d6 94  c8 5a 3e 70  32 c2 83 73  ac 37 00 55
         1f 1f 79 98  4b b5 1e 60  12 81 ca 3b  92 3c ed 39  71 6e a8 79
(i=79)   4e 9f a2 8f  24 de 68 5a  c4 d7 4c 1a  ab b5 04 29  2d 6e 74 13
         21 98 41 3b  5c 0d 44 b5  1f 7e b9 49  d4 72 c7 f3  17 97 ad 86
(i=81)   0d ed e7 16  59 aa 35 2f  56 9d 48 97  b9 5f 7d 1c  ae 24 f5 4a
         95 56 43 b4  24 b0 55 0d  45 d3 c6 46  c3 80 09 29  f1 51 53 af
(i=85)   44 b7 98 97  18 f8 7a b6  8b 69 15 6c  4b 25 79 99  bc ce 6e 3d
         16 0c c4 eb  98 6e 57 86  0d 76 d7 9e  e9 d5 16 26  e0 5f 95 e5
(i=86)   20 ee 62 48  99 b7 c8 fb  12 9e e1 39  20 b7 3f 95  39 dc 18 b6
         09 8d 7a 6c  cf e2 15 88  86 3e 7a af  41 fb 6b 33  dd 4e a3 27
(i=91)   5e c8 ab 7a  ff 93 ff c5  c3 1f a0 87  65 3e 74 61  06 b1 aa 7c
         05 0a 88 16  48 d5 96 3e  07 75 ee 6d  43 74 33 d6  ee a6 c9 7c
(i=94)   af 22 df b8  d9 dc 31 2b  9f 56 6a 38  e6 fd c2 a6  8d e6 76 49
         da 9c 4e 12  c5 e3 40 b7  fa f6 a4 ee  7a e8 f1 d8  67 6e 6c 8d
(i=98)   a8 61 bb d0  2a 36 65 69  79 9f 9c 9e  c2 34 8c 57  0e a5 c4 8e
         61 d1 1a df  9a 75 06 b3  77 04 4e 67  6d 69 a7 59  9e e2 2a 8f
```

```
(i=102)  cd 29 da c3  23 75 41 81  ca f9 d0 dc  9c 7d be bd  ea dc 8e 3f
         e2 90 68 24  21 aa d2 80  4c 96 14 63  ea 77 51 d2  91 63 e0 10
(i=104)  bc 18 ff 8b  42 68 3a c0  a6 11 21 36  d0 bb 98 06  1c 42 6a 76
         ac 41 e9 e3  6f ef 59 b5  18 63 e9 f5  b0 73 ce e0  3b dc d3 91
(i=107)  76 84 fc 7a  31 8d 02 df  99 0a 60 12  e8 0c f2 3a  5a 1c b3 a8
         12 1d 20 ad  20 70 18 03  5d ea 1e c1  7d 48 60 a6  37 59 e1 3b
(i=109)  0c 2c b6 e6  2e 7c f1 04  61 29 53 0b  27 e8 0a 8b  b4 50 f1 82
         5b 4f 86 89  57 3a c9 84  1c 38 63 48  b2 14 2d 7b  c9 1f dd b8
(i=113)  60 a2 a7 3c  c4 24 ab 70  5e 18 42 30  ef 07 fd 84  f3 2c 09 d3
         b5 83 c4 63  a0 6c 2f 60  53 02 14 c9  71 62 72 02  fe eb aa 8d
(i=114)  99 f6 1d 2d  1a ba 53 2a  ca 15 31 1f  14 cf 1c 77  ec 6b e5 eb
         06 dd f8 a1  7a b5 61 c6  2f 39 de 7a  f2 21 c0 47  85 20 76 5a
(i=115)  f9 2f 71 a3  0b 10 e9 d2  84 81 2e 0e  03 f4 e4 96  df 64 24 c7
         1e 2e 52 d5  b8 8f aa f8  95 15 15 44  a3 a2 7f 95  ca a7 ab 26
(i=118)  3c 66 3c 03  44 64 5f c3  da 17 d6 c8  6f f1 d3 85  04 2c 43 ba
         17 6d 2e ed  09 e9 de 36  6f 5e 47 aa  7f d9 49 46  4b 66 f9 6b
(i=119)  77 a9 73 ce  a4 9d b3 39  cb 6d 6c 70  29 5d d0 74  f3 51 0b d9
         0a 66 6d c9  21 3a 38 6a  ad 38 90 dc  e5 b5 80 10  fc e7 b8 b9
(i=121)  b8 63 ae 11  db 68 13 72  1f e3 5d c6  bf 05 8a e0  f0 40 fa fe
         d2 85 60 c2  60 16 50 bb  07 6c ce b6  2e e7 e6 ec  33 b1 69 3a
(i=122)  34 a4 68 4c  1e 9f de d2  58 37 d3 b7  15 9b 32 9a  5c 3d e9 ed
         f7 4d 7f b5  59 55 2c d3  58 c6 02 f4  08 30 18 52  0f e8 33 eb
(i=123)  e7 20 a9 06  59 e2 15 9d  b8 70 27 2d  06 f1 c8 42  16 a9 e6 dc
         e6 72 47 d4  4c 4e 6b af  70 17 5c 28  46 0a 61 84  75 c4 6a b5
(i=126)  00 d3 94 b9  d5 23 cf d8  fb a7 f2 8d  3f 45 3e 33  6c 3f 8e 96
         52 6f 36 c3  71 16 8a a2  69 56 38 40  97 64 95 c2  4f 0d 9c 1b
(i=127)  8b ec 47 a4  88 9f 10 92  36 ea 29 58  9f 7e 92 a9  5d 95 24 3e
         0c db 33 b2  60 3b 52 c1  5c 4f 77 1c  af b5 ef f6  8d e7 e5 4d
(i=131)  c8 3b b1 52  13 b8 c3 7d  e9 66 6a 12  da c1 c9 74  bd ce 78 b4
         fd 31 c9 5a  1a a7 4f b0  4b 74 3f 3b  a2 ae 2e d2  a5 38 3f 81
(i=132)  91 78 00 bc  c1 43 dc 30  d4 19 e6 53  94 fc 0c ab  88 2e b1 08
         73 22 1f f0  c2 61 bb ad  3a 63 64 03  c1 a1 27 11  81 50 90 db
(i=135)  e3 bb 60 85  fe 92 46 de  5f 32 99 3e  47 78 4d 65  c3 71 e8 d3
         d3 5b 73 66  b3 b7 51 55  f4 cf 61 f2  b0 c6 ef 30  74 49 cf b7
(i=137)  17 36 b2 c8  5e 5b 83 2d  c9 e0 24 57  fa 63 00 e1  04 2b 23 16
         0a 26 d3 9f  07 2d 42 ab  8a 77 1b 5e  ad b5 de 55  3c 68 c2 b0
(i=138)  ea 6a 2d 1a  a1 bb 4c 6a  18 4a d2 e2  13 16 eb 94  80 6c dd 51
         4d 5d 9e ff  40 81 b8 9c  e0 0d c3 18  19 b2 cd 44  61 30 e1 a3
(i=141)  e0 d8 78 ed  d5 36 9e ad  78 13 0f 31  7d c4 76 ad  33 57 90 cd
         8e 17 d9 42  77 4c 18 d5  34 83 b4 6e  af 5a 87 b0  5e 1f d0 c8
(i=142)  3d ce e6 38  a8 6a 19 ff  bb 73 d8 6e  cc 2e 24 38  4c 0a 7b 80
         0a 58 93 7d  ba 83 e3 35  6d d7 2a 5f  05 f0 2f 6a  ca 1c bf b7
(i=144)  e8 e6 43 2e  16 b5 ec 33  36 c5 1b ce  95 6b 73 a2  fa 95 94 33
         f5 0b 0f be  74 be 26 6c  38 37 63 b3  7b e1 85 00  72 d6 2b b4
(i=146)  be c7 ee 46  73 ab 5a 7e  09 f9 96 20  d8 cb 3c df  49 ff 42 be
         0e be fa 71  f0 ff e0 a7  7b 6e 2e 13  b4 35 fb f1  c8 6c d3 6e
(i=148)  95 2a c4 27  1e c3 b7 74  77 44 69 54  53 1d 7f 3f  12 3c 91 28
         bc 49 13 24  db b2 5c e8  35 a9 71 4a  7f 95 34 45  3e 5d 29 04
(i=151)  d4 1d 0f fe  81 99 98 1f  8f a1 5f c2  9e f0 b3 ba  64 7f f1 f1
         f9 98 7d d2  66 cb 0f d3  e8 25 b2 04  ba d8 6b 10  9e 96 7d 7a
(i=154)  15 65 f1 3d  74 e4 6f 82  65 82 0a da  fb e6 21 05  37 b3 6c 43
         3c f8 46 0f  b5 35 bd 5e  01 d8 9d b7  36 19 25 4b  e1 cd 48 da
(i=155)  c5 a6 39 1f  b3 d7 ba 59  c8 58 eb 85  13 43 17 73  82 86 a0 be
         8e 3b a6 d8  f2 84 27 0c  8c f1 50 a2  e9 95 66 05  1c 10 7f a5
(i=156)  8c 56 7a 67  95 16 ad a4  9f bb c1 66  be 5b 74 69  f0 d1 73 f2
         09 8d e9 38  bb c1 76 76  3a 7c 69 55  d4 48 e2 46  d6 4b c2 dc
(i=158)  c4 bc 41 17  d6 5e 8f e3  92 06 98 c9  94 3c 1f 81  4d c7 e1 3d
         dc c1 64 8a  fe 21 3f b3  89 e6 17 e0  ed fb cd f9  52 8c 7c 17
(i=159)  70 f4 a7 de  86 9f d7 c5  d1 f9 e3 a0  f7 12 00 2c  65 24 d7 ab
         27 94 98 05  50 64 9f 7c  c6 35 81 8e  78 14 80 e4  05 08 bd d1
(i=163)  1f 5f 8f 4f  32 d7 3d 8c  81 3a 2b 82  36 05 4b 03  c8 fa b8 56
         3f f1 8e 73  9b 37 d3 f7  18 78 e1 57  b5 63 ea 92  90 21 70 bc
(i=164)  79 0e fa 37  a3 83 75 f2  48 ea 6c ca  18 44 3e 4e  9f 5d 8e 37
         ea 09 eb 69  09 82 a6 2b  93 ca 24 b7  7e a0 39 fc  3e ac 89 6f
(i=168)  19 f6 3c 57  fd 32 e0 da  b9 96 a4 30  df f4 7f 96  81 9c 81 82
         c1 6c c1 4a  b4 9a 03 21  01 15 f7 eb  f9 f2 7c 5c  07 e9 d8 d9
(i=169)  66 96 24 99  1d 8c 8f 45  a1 07 50 68  45 bb 2f d7  c9 7e c0 75
         0c 43 24 20  95 45 1b 7e  f7 83 42 be  2d 6d ce 9f  67 b2 15 28
(i=172)  c8 85 dc e6  bd 74 d1 65  fb b6 bb 50  b6 67 67 3d  90 2e 01 bd
         ee 82 17 6b  6c a8 f1 5f  a2 9b 9f b4  9b b8 a1 d3  e2 04 58 88
```

```
(i=179)   75 90 b1 7f  b1 47 23 c7  ea 6e 08 f0  9e a9 87 97  3f 99 e9 2e
          9a ba 7d 32  1c e9 39 41  e1 8e ea 8b  fe 8e 82 7e  fa 61 4e f6
(i=182)   64 d8 06 2c  bc 1c bc bb  bd c0 6a df  56 f6 27 7f  81 b9 43 dd
          05 a2 ee de  54 4f 00 f1  22 d6 cc ca  f1 d9 59 e1  d0 42 f9 0e
(i=186)   11 38 51 d0  ab 64 11 68  c8 95 03 d3  29 48 89 6e  39 06 e3 c5
          47 c2 48 8d  bf 37 71 9d  5a 3c 93 7a  32 21 3b 20  c3 8d d0 71
(i=190)   76 e1 77 de  58 c4 5c 0c  b7 dd 58 80  34 1d 22 62  0c 58 45 b4
          ff 0f e8 75  01 57 cb 4c  c5 24 04 26  6a 87 02 d0  04 d5 b2 b0
(i=192)   da 04 dc 87  7e d2 09 6c  02 81 47 c8  89 ca 2d 37  a5 4c 18 06
          61 fe a0 c2  a1 3f 0d 6c  1f d3 6f 0e  db 33 3a 36  cb 85 f3 f8
(i=195)   3f 32 c9 eb  a1 37 b2 92  10 2e a7 13  2d b9 75 8c  52 57 ed 9f
          55 d1 f2 24  90 f7 5a 0c  07 15 8f 68  8a 9e 22 a7  77 bd 59 bf
(i=200)   f8 7a f0 ef  b5 9c e0 7f  74 51 0c bc  53 c7 22 ec  9d fb dc e7
          aa 7e fd f9  29 eb 2d 5e  69 04 47 b5  2a 86 64 c7  d1 6c c4 a7
(i=201)   fd 33 38 16  b9 b0 45 ad  61 b5 2f 21  fc ed 30 99  fd 46 80 d6
          f2 d3 aa 04  fe 84 21 31  bb 66 36 6d  77 26 4c 09  f1 c6 73 12
(i=202)   c2 38 f1 5e  e0 b4 59 12  8f a2 93 44  61 96 56 a7  aa a6 cb 7a
          e1 1b ff b1  09 59 ba 25  8e b8 98 5c  2f 73 ec f1  33 e6 cd c1
(i=204)   a9 a4 b6 63  99 fc 80 16  a3 07 c1 31  c5 b9 bb 50  d0 b4 78 da
          2c bf ee f9  5e 06 c5 fa  27 ac 6b ae  91 62 a4 3e  d3 ce 0f e1
(i=205)   df 8b 22 28  9e b5 c8 3d  a7 1b 26 5f  b2 1d de b5  79 da 86 87
          8c 0a 92 e8  a6 5b 72 05  fc 45 5f 81  e3 c4 93 f6  20 6e f7 23
(i=206)   97 c1 09 94  63 ba 81 85  ce 1f 3a c4  e0 0a 42 d8  de 83 ac 95
          39 6a dd 8c  95 a3 c7 b2  07 1a f8 75  b6 16 f5 e5  d8 bb 97 0b
(i=209)   25 90 e2 4c  99 a1 ed 4a  d3 d8 82 eb  e4 1e a7 06  cb e7 cf fa
          e2 90 eb 39  f5 ee 6b a1  4f 6f a5 80  8b af e9 b8  2a 1d 86 c3
(i=210)   08 1e b1 25  51 d7 d4 b6  98 dd 3b 33  0b 22 bb 6b  f9 d4 33 1d
          47 39 11 47  a2 4e b6 45  3e b7 fa 2d  96 84 82 ac  fd 6f e8 b2
(i=212)   79 dc 94 b3  20 b0 8c ec  7f 49 00 38  c4 6a e2 6f  0d 39 61 0a
          ab 5c 76 f0  c8 5c 63 a5  89 5b e9 75  eb 31 8d 81  96 63 bb 04
(i=214)   7e 1d 05 71  03 3e 5b c5  37 7f e7 a4  89 6f 9b b7  34 3d 75 6f
          d9 49 da 13  2d 05 89 b3  36 bb 34 19  da 79 e2 2e  a1 38 54 f8
(i=216)   6a 25 0a b2  74 fc 3e 53  06 58 9f da  70 db 60 bc  ed 85 9c 73
          ed ae 08 00  91 28 ee 5c  5c c9 e1 79  25 1d d1 76  f6 e5 5f cd
(i=218)   97 f8 f6 ba  79 3d af 11  e9 e6 6e b3  28 11 47 28  b2 8a 55 bb
          14 b2 1c 65  bf 15 52 7f  c1 72 07 87  d2 7d 1c 1a  e5 2d b4 7a
(i=219)   8f 25 c9 a6  81 42 f0 82  a7 c9 fc 82  01 c9 7d 0f  1e 4f 5a 74
          5c d9 20 79  24 43 3f e3  e4 d7 b0 ad  e0 2a 7c 65  89 1c fc cf
(i=220)   65 1d f6 79  6d 4a f5 c3  18 87 df 10  d0 a2 35 45  05 bb 1f 79
          15 21 47 7d  38 a8 6d d0  48 fa 15 56  06 38 29 c5  d4 c0 eb 17
(i=227)   7a cd de da  84 cb 3c 99  10 b4 b2 fc  d8 a7 76 b6  c3 9e ad 48
          ee d9 7d 64  a4 6d 72 89  90 21 19 6a  6b 66 16 29  f7 25 94 3d
(i=228)   f0 e2 8e c2  e5 e2 bd e0  e6 ac df cf  c4 af 7b f7  34 5c 90 d6
          bd b2 35 9a  8b d9 37 8e  49 69 40 6e  7f cb 44 16  5b 48 f9 e6
(i=233)   af 82 b9 d9  ca 58 d2 90  ce 0d f6 50  0b 85 73 24  07 48 98 db
          fe 23 f3 7d  19 a8 10 46  7f 50 ac 33  84 84 8c 3d  5f 5c 5e 14
(i=237)   a3 89 47 6b  89 f8 fd a7  b3 83 0b 00  f3 e6 8a a5  4e 1e 90 08
          d1 0f fb 82  5a 19 ce 29  0d 1f 85 eb  ba 6b f8 02  64 15 a6 3b
(i=242)   9e 4e d8 9e  96 ec 04 35  45 42 ab 2b  0a cb 00 ba  fe 06 f1 1f
          52 56 d1 7a  87 ec ba 31  12 60 f6 a9  9d f9 c7 db  1c 4b 8d a7
(i=243)   4b 49 9d 2f  c9 f9 f8 3c  d3 d4 6a cb  35 e2 e5 30  13 b6 d9 80
          69 d7 18 50  7f 19 8d 1d  1a 65 37 1a  5b dc 55 aa  f5 03 c3 8e
(i=244)   20 f6 98 f4  5a 2c 05 30  da 62 fc 8a  d5 0d fc 15  89 cb 89 68
          ca ee 99 97  55 11 ba f0  06 6d 3c 5b  cc 9a 38 38  c4 dc 7b c4
(i=245)   9d cb 45 ef  1f bd 38 3d  ce 69 8a 1c  94 ad 27 2c  6e 41 9e 18
          b2 4f b0 18  9c e7 b2 1d  d9 59 44 60  0d 0b f6 1b  52 ab 54 7c
(i=247)   77 11 c2 c4  cc b8 fd ce  01 76 7e 23  22 3f e6 cc  99 58 83 8e
          c7 ff 98 79  b3 68 f9 f3  d1 86 17 4c  15 10 37 8c  10 8e e2 4b
(i=248)   98 eb 08 41  a1 65 f8 93  92 a9 8b 17  29 cd 78 8b  39 83 9a 73
          3d 14 48 61  14 7f 7a 3a  a7 7e 44 1f  01 18 3c cd  81 4c c5 d9
(i=251)   18 c8 d6 62  7b ab 75 68  3f a4 50 a8  5c da 6c 92  c7 15 59 13
          68 2b 2d d7  29 2f 62 9b  be ff 8b f5  f9 45 0f b9  89 51 06 4a
(i=252)   2b 48 b3 30  9c 85 bb e5  14 51 4b 6d  ed 0d 79 86  ce a3 eb d2
          08 56 44 bc  9f 44 12 83  1f 16 0c 3c  cf 3d 3c 8c  75 59 0b 8b
(i=253)   1f 5b 33 0d  6a a6 95 2b  91 26 f8 68  b2 9e ac 93  c2 aa 79 64
          c7 f6 6f d3  84 ba 27 33  07 77 23 bd  16 13 34 b9  48 45 13 90
(i=256)   6f 93 59 01  7d 26 72 f9  b2 00 3e e5  87 4b a7 58  53 dd 86 58
          ce 84 01 92  24 e5 3e 18  7d 8c d3 a5  77 2a b5 00  1e 3d 40 63
(i=258)   87 34 a9 39  a3 1a 85 79  8f ce 5f bf  cd c8 7c 05  4e a2 17 8b
          db 78 08 20  b6 a4 de 43  94 71 49 ba  27 12 16 17  9f 84 16 5b
```

```
(i=262)  6f bb d4 ac  bb bb d5 b1  b5 c2 72 3f  aa 96 9d df  94 1f ec 38
         d6 3d 99 53  c3 98 e5 d1  26 30 e9 e5  3e f7 8c 2c  4f 6c 77 72
(i=265)  4a f0 14 94  42 e6 48 c9  56 12 aa 65  9e a9 1d bc  62 40 c6 7e
         53 12 46 4e  88 29 18 de  1a 37 77 77  fd 97 b7 43  34 e2 8c 22
(i=270)  b8 ea 54 f0  d6 c1 7d 09  3e 99 d5 d8  b6 4a 6d f4  88 34 d9 fe
         30 e0 67 28  ce a6 ed 8b  15 fc 08 aa  0a 8b be d1  c6 a1 2c 4d
(i=275)  5a df 3e 09  ce 2f 77 49  9a 2d b0 0d  f6 32 f4 1f  fb 4c 7a 4e
         68 06 5b 3b  4e 83 bb ac  ef 42 85 7f  b7 86 83 62  f9 ae 20 54
(i=276)  d7 81 33 f3  e7 27 e5 43  da 89 44 e8  2b 72 dc a6  26 bf 92 ef
         b8 3e 81 2f  61 03 98 7a  10 1c cb fc  8c 33 7e 27  8a e1 2d 48
(i=278)  06 f9 b0 95  dc 11 fe 3b  48 83 84 44  bf 4d 11 e6  0e 46 bd 62
         d0 df d1 67  87 f7 ab fa  ed ea ec d6  d2 b0 53 d4  85 a6 be 7b
(i=279)  f4 28 28 12  7e 06 e8 54  40 f1 7e 84  1b e1 ec 1b  4e 2e 44 8d
         43 f7 72 b7  bf 1d 9f 0d  6d c7 ba f7  ac f6 d0 a9  32 a1 83 0c
(i=281)  ec d7 16 41  f6 83 8a 49  2a 0a 76 f2  15 21 48 af  29 63 84 75
         ca 22 e5 cf  60 6d d7 33  61 da 3a d4  7a 17 aa ef  af 76 30 07
(i=283)  63 13 0e f0  e4 b2 02 c6  cc ff 60 0b  0d 8f 42 ef  85 f7 5f aa
         0a 0a 6c fa  d3 85 78 83  99 00 2e e7  fa f4 78 10  89 bc ad dc
(i=286)  24 12 15 67  20 aa b1 b4  fb 77 dd ad  02 79 5b e7  f3 f1 9f 06
         9e e5 a1 3a  bb 0c a3 f6  b1 a1 7e 1f  20 e8 8b 90  66 8a ce b6
(i=287)  50 d3 14 6e  97 e6 a9 63  e9 a6 55 2a  a4 6e 45 00  eb 5f 99 46
         fa 79 7c 6f  fb f4 2a 21  24 b9 1f 6f  58 0e 7f a3  e6 67 9c d7
(i=288)  ba ff d5 6d  9e 5d e5 5b  98 94 84 a2  21 10 3a ea  04 57 07 40
         3a d5 10 4a  30 34 12 a8  4f 2c 37 10  a8 46 a5 97  f9 e7 79 a5
(i=292)  ce 2e a5 52  08 89 a6 5a  9f 0b c0 9a  d0 fe 69 62  81 f9 fc 2a
         53 cd 7e 44  70 90 a6 83  84 6c 1f 97  d3 b9 bd 38  49 1f 9f 99
(i=293)  b9 42 d4 22  ed f3 d2 1b  9e 12 37 d6  c8 ad 57 91  f9 76 9e 1f
         3d e6 76 b2  6a d0 02 17  5f a1 5f 7f  5a e4 30 50  ea 6f d7 bf
(i=294)  e6 2d e8 51  bd d8 3c 47  5f 11 3e 4d  04 a5 06 7f  28 ee 1b c1
         32 d0 8f aa  d8 ca 42 73  f3 7c 94 bf  42 6b 5b c3  02 10 27 f7
(i=295)  00 5a d3 65  ec a8 21 b1  8b d2 3d 54  7b e1 fe 2e  16 1d 93 3e
         d4 c5 79 c3  d0 38 3c b3  4f 10 6f f4  82 53 e2 ee  75 28 c8 47
(i=297)  c6 3d ed 92  d7 bc 50 81  70 3c 69 15  7a e8 75 6a  0e cc 81 6d
         4c 42 1b b8  ba 51 34 21  49 50 cb 88  5d 88 22 d6  fc 53 3b 5f
(i=300)  b1 a5 97 58  ba d6 7d 6c  84 6b 39 fa  e4 14 36 69  15 43 bd 65
         fb 30 4a 30  37 f3 29 0b  62 48 39 82  9d e4 b6 b1  31 93 23 e6
(i=302)  de e4 82 c0  64 9c 60 86  b1 56 4d 29  b4 f9 a0 95  d6 42 c4 dc
         37 28 f9 1e  66 6b a6 ad  57 32 52 7a  0b de f6 0d  c5 6e 58 26
(i=303)  0e 11 c1 ab  cc 46 26 69  cb 83 38 3d  e3 c9 85 ff  02 03 c3 e3
         ae 64 f1 cd  54 9a 1e 2a  f9 27 3c 93  03 4c f0 4d  21 02 33 5b
(i=304)  83 41 ee ea  b7 ae d0 2f  ae 9d 65 28  f7 f8 55 e4  6c 2f 84 e2
         b5 db 2d c5  03 88 4d a2  76 c9 31 7d  1c 44 5e 47  61 5e c7 36
(i=310)  9f 30 f0 3f  d2 c4 45 5f  db dc 50 90  a1 be 38 fe  99 1a 98 11
         85 c0 97 f1  c4 87 54 19  b2 c1 e0 6b  4b bc db e9  56 48 e0 2e
(i=318)  c3 ef ab 35  27 e3 e6 a7  f7 cb 52 e5  bc d4 ad 2e  c6 59 83 79
         2f 86 7a 0b  f1 72 68 48  82 a6 4a 97  0c bb e2 60  92 40 8f 1c
(i=319)  be 13 6a f0  1d 38 05 48  3f e7 41 e7  11 ef c3 a3  f6 86 c2 64
         97 30 40 ee  0b 9f 53 5c  b1 76 2f 01  38 7c e1 67  09 7c 87 cb
(i=323)  c7 76 f7 00  18 5c c4 03  35 3c 60 88  59 0b b2 a5  4b a1 d8 d9
         c7 5d 7f d9  73 49 19 3f  cb a3 1a 15  67 4c c6 d1  35 3d 86 d2
(i=324)  6f 7f 5a 8d  28 57 e8 c2  f0 32 b5 a7  fa 53 ce 94  4d f6 f3 ef
         3c 8d ac 18  5e b1 c3 05  ae bd 47 00  7b 7b 96 b6  9f 69 47 d1
```

Final hash function result:

```
Hash:    67 58 25 ec  f3 ba f5 c9  4f fe 38 a1  5b c0 ab 40  77 9b 96 4d
```

Appendices

Bibliography

[Ajt96] M. Ajtai. Generating hard instances of lattice problems. In *28th ACM Symposium on Theory of Computing*, pages 99–108, 1996.

[Al-92] I.A. Al-Kadi. Origins of cryptology: The arab contributions. *Cryptologia*, 16(2):97–126, 1992.

[ANS99] ANSI. X9.62 – public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA). Technical report, ANSI ——, 1999.

[ARS78] L. Adleman, R.L. Rivest, and A. Shamir. A method for obtaining digital signature and public-key cryptosystems. *Communication of the ACM*, 21(2):120–126, 1978.

[Bab86] L. Babai. On lovàsz lattice reduction and the nearest lattice point problem. *Combinatoria*, 6:1–13, 1986.

[BBB+98] C. Batut, K. Belabas, D. Benardi, H. Cohen, and M. Olivier. *User's Guide to PARI-GP*. By anonymous ftp from `ftp://megrez.math.u-bordeaux.fr/pub/pari`, 1998. See `http://pari.home.ml.org`.

[BCJ+05] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet, and W. Jalby. Collisions of SHA-0 and reduced SHA-1. In *Springer-Verlag*, LNCS 3494, pages 36–57, 2005.

[Ben05a] K. Bentahar. The equivalence between the DHP and DLP for elliptic curves used in practical applications, revisited. In N.P. Smart, editor, *Cryptography and Coding*, LNCS 3796, pages 376–391. Springer-Verlag, December 2005.

[Ben05b] K. Bentahar. The equivalence between the DHP and DLP for elliptic curves used in practical applications, revisited. Cryptology ePrint Archive, Report 2005/307, 2005.

[BG04] D.R.L. Brown and R.P. Gallant. The static diffie-hellman problem. Cryptology ePrint Archive, Report 2004/306, 2004.

[BPS+06] K. Bentahar, D. Page, M.J.O. Saarinen, J.H. Silverman, and N. Smart. LASH. In *2nd NIST Cryptographic Hash Workshop*, 2006.

[BR94] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology — EUROCRYPT '94*, LNCS 950, pages 92–111. Springer-Verlag, 1994.

[BRS02] J. Black, P. Rogaway, and T. Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In *Advances in Cryptology — CRYPTO 2002*, LNCS 2442, pages 320–335. Springer-Verlag, 2002.

[BS07] K. Bentahar and N.P. Smart. Efficient 15,360-bit RSA using woop-optimised Montgomery arithmetic. In S.D. Galbraith, editor, *Cryptography and Coding*, LNCS 4887, pages 346–363. Springer-Verlag, December 2007.

[BSS99] I.F. Blake, G. Seroussi, and N.P. Smart. *Elliptic curves in cryptography*, volume 265 of *LMS Lecture Note Series*. Cambridge University Press, 1999.

[BSS04] I.F. Blake, G. Seroussi, and N.P. Smart. *Advances in Elliptic Curve Cryptography*, volume 317 of *LMS Lecture Note Series*. Cambridge University Press, 2004.

[BV98] D. Boneh and R. Venkatesan. Breaking RSA may not be equivalent to factoring. In *Advances in Cryptology — EUROCRYPT 1998*. Springer-Verlag, 1998.

[CDMP05] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle–damgård revisted: How to construct a hash function. In *Advances in Cryptology — CRYPTO 2005*, LNCS 3621, pages 430–448. Springer-Verlag, 2005.

[CFA+06] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press, 2006.

[Che06] J.H. Cheon. Security analysis of the strong Diffie-Hellman problem. In *Advances in Cryptology — EUROCRYPT 2006*, LNCS 4004, pages 1–11. Springer, 2006.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, second edition, 2001.

[CLS05] S. Contini, A.K. Lenstra, and R. Steinfeld. VSH, an efficient and provable collision resistant hash function. APR e-print 2005/193, 2005.

[CMP+07] S. Contini, K. Matusiewicz, J. Pieprzyk, R. Steinfeld, J. Guo, S. Ling, and H. Wang. Cryptanalysis of LASH. Cryptology ePrint Archive, Report 2007/430, 2007.

[Coh93] H. Cohen. *A Course In Computational Algebraic Number Theory*. GTM 138. Springer-Verlag, 1993.

[Coh05] H. Cohen. Analysis of the sliding window powering algorithm. *Journal of Cryptology*, 18(1):63–76, 2005.

[Coo06] S. Cook. The P versus NP problem. *The Millennium Prize Problems*, 2006.

[CS03] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal of Computing*, 33:167–226, August 2003.

[Dam88] I.B. Damgård. Collision free hash functions and public key signature schemes. In *Advances in Cryptology — EUROCRYPT 1987*, LNCS 304, pages 203–216. Springer-Verlag, 1988.

[Den] T. St Denis. `LibTomCrypt`: A portable ISO C cryptographic toolkit. `http://libtomcrypt.org/`.

[Den06] A.W. Dent. The hardness of the DHK problem in the generic group model. Cryptology ePrint Archive, Report 2006/156, 2006.

[DH76] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976. `citeseer.ist.psu.edu/diffie76new.html`.

[DR02] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag, 2002.

[Dwo97] C. Dwork. Positive applications of lattices to cryptography. In *22nd International Symposium on Mathematical Foundations of Computer Science*, LNCS 1295, pages 44–51. Springer-Verlag, 1997.

[FOPS01] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. In *Advances in Cryptology — CRYPTO 2001*, LNCS 2139, pages 260–274. Springer-Verlag, 2001.

[FOPS04] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. *Journal of Cryptology*, 17(2):81–104, 2004.

[FS03] N. Ferguson and B. Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[Gar07] Luis Carlos Coronado García. Can Schönhage multiplication speed up the RSA decryption or encryption? *MoraviaCrypt*, 2007. Preprint available from: `http://www.cdc.informatik.tu-darmstadt.de/mitarbeiter/coronado.html`.

[GAST05] J. Großschädl, R.M. Avanzi, E. Savaş, and S. Tillich. Energy-efficient software implementation of long integer modular arithmetic. In J.R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 75–90. Springer-Verlag, 2005.

[Gau] D. Gaudet. SHA1 and SHA256 using SSE2. `http://www.arctic.org/~dean/crypto/`.

[Gay] O. Gay. SHA-224, SHA-256, SHA-384 and SHA-512. `http://www.ouah.org/ogay/sha2/`.

[GGH96] O. Goldreich, S. Goldwasser, and S. Halevi. Collision-free hashing from lattice problems. In *Electronic Colloquium on Computational Complexity TR96-042*, 1996.

[GGOQ98] H. Gilbert, D. Gupta, A. Odlyzko, and J.J. Quisquater. Attacks on Shamir's 'RSA for paranoids'. *Information Processing Letters*, 68(4):197–199, 1998.

[Gol04a] O. Goldreich. *Foundations of Cryptography, Volume I*, volume 1. Cambridge University Press, 2004.

[Gol04b] O. Goldreich. *Foundations of Cryptography, Volume II Basic Applications*, volume 2. Cambridge University Press, 2004.

[Gor98] D.M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, 1998.

[Gra07a] T. Granlund. GNU multiple precision arithmetic library 4.1.2. `http://swox.com/gmp`, 2007.

[Gra07b] T. Granlund. Instruction latencies and through put for amd and intel x86 processors. `http://swox.com/doc/x86-timing.pdf`, 2007.

[Har05] L. Hars. Fast truncated multiplication for cryptographic applications. In J.R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2005: 7th International Workshop, Edinburgh, UK,*, LNCS 3659, pages 211–225. Springer-Verlag, August 2005.

[Har07] L. Hars. Applications of fast truncated multiplication in cryptography. *EURASIP Journal on Embedded Systems*, 2007:Article ID 61721, 9 pages, 2007. doi:10.1155/2007/61721.

[HMV03] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to elliptic curve cryptography*. Springer-Verlag, 2003.

[HPS98] J. Hoffstein, J. Pipher, and J.H. Silverman. NTRU: A new high speed public key cryptosystem. In *Algorithmic Number Theory — ANTS III*, LNCS 1423, pages 267–288. Springer-Verlag, 1998.

[ISO96] ISO/IEC 10118-4. Information technology – security techniques – hash-functions – part 4: Hash-functions using modular arithmetic. Draft, 1996.

[JK03] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA cryptography specification version 2.1. Technical report, RFC 3447, February 2003.

[Jou04] A. Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *Advances in Cryptology — CRYPTO 2004*, LNCS 3152, pages 306–316. Springer-Verlag, 2004.

[Kah67] D. Kahn. *The code breakers: the story of secret writing*. New York MacMillan, 1967.

[KAK96] C.K. Koç, T. Acar, and B.S. Kaliski, Jr. Analyzing and comparing Montgomery multiplication algorithms — assessing five algorithms that speed up modular exponentiation, the most popular method of encrypting and signing digital data. *IEEE Micro*, 16(3):26–33, 1996.

[Ker83] A. Kerkhoff. La cryptographie militaire — (2 papers). *Journal des sciences militaires*, IX:5–38, January 1883.

[Knu98] D.E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley Longman, Reading, Massachusetts, third edition, 1998.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[Koc94] C.K. Koc. High-speed RSA implementation. Technical Report TR201, RSA Laboratories, 1994.

[KS05] J. Kelsey and B. Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In *Advances in Cryptology — EUROCRYPT 2005*, LNCS 3495, pages 474–490. Springer-Verlag, 2005.

[LJ87] H.W. Lenstra Jr. Factoring integers with elliptic curves. *The Annals of Mathematics*, 126(3):649–673, 1987.

[LLH03] C.-L. Lei, C.-B. Liu, and C.-H. Huang. Design and implementation of long-digit karatsuba's multiplication algorithm using tensor product formulation. *The Ninth Workshop on Compiler Techniques for High-Performance Computing*, pages 23–30, 2003.

[LMPR06] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. Provably secure FFT hashing. *NIST 2nd Cryptographic Hash Function Workshop*, 2006.

[Luc04] S. Lucks. Design principles for iterated hash functions. Cryptology ePrint Archive, 2004/253, 2004.

[Mao04] W. Mao. *Modern Cryptography: Theory and Practice*. Prentice Hall, 2004.

[Mau94] U.M. Maurer. Towards the equivalence of breaking the diffie-hellman protocol and computing discrete logarithms. In *Advances in Cryptology — CRYPTO 1994*, LNCS 839, pages 271–281, 1994.

[Mer90] R.C. Merkle. A fast software one-way hash function. *Journal of Cryptology*, 3:43–58, 1990.

[MG02] D. Micciancio and S. Goldwasser. *Complexity of Lattice Problems: a cryptographic perspective*, volume 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, Massachusetts, March 2002.

[Mic01] D. Micciancio. The shortest vector problem is NP-hard to approximate to within some constant. *SIAM Journal on Computing*, 30(6):2008–2035, March 2001. Preliminary version in FOCS 1998.

[Mil86] V.S. Miller. Use of elliptic curves in cryptography. *Advances in cryptology — CRYPTO 85*, pages 417–426, 1986.

[MMA97] M. Mrayati, Y. Meer Alam, and H. Al-Tayyan. *Origins of Arab Cryptography and Cryptanalysis*. Arab Academy of Damascus, 1st vol. 1987, 2nd vol. 1997.

[MOI90] S. Miyaguchi, K. Ohta, and M. Iwata. 128-bit hash function (N-hash). *NTT Review*, 2(6):128–132, 1990.

[Mon85] P.L. Montgomery. Modular multiplication without trail division. *Mathematics of Computation*, 44(170), 1985.

[MOV97] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook Of Applied Cryptography*. CRC Press, 1997.

[MS01] A. May and J.H. Silverman. Dimension reduction methods for convolution modular lattices. In *Cryptography and Lattices Conference — CaLC 2001*, LNCS 2146, pages 110–125. Springer-Verlag, 2001.

[MSV04] A. Muzereau, N.P. Smart, and F. Vercauteren. The equivalence between the DHP and DLP for elliptic curves used in practical applications. *LMS J. Comput. Math.*, 7:50–72, 2004.

[Mul97] T. Mulders. On computing short products. Technical Report 276, Dept of CS, ETH Zurich, November 1997. `ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/2xx/276.pdf`.

[MW96a] U.M. Maurer and S. Wolf. Diffie-hellman oracles. In *Advances in Cryptology — CRYPTO 1996*, LNCS 1109, 1996.

[MW96b] U.M. Maurer and S. Wolf. On the difficulty of breaking the DH protocol. Technical report, Department of Computer Science, ETH Zurich, 1996.

[MW99] U.M. Maurer and S. Wolf. The relationship between breaking the diffie-hellman protocol and computing discrete logarithms. *SIAM Journal on Computing*, 28:1689–1721, 1999.

[MW00] U.M. Maurer and S. Wolf. The diffie-hellman protocol. In *Designs, Codes, and Cryptography*, volume 19, pages 147–171, 2000.

[Nat06] National Institute of Standards and Technology (NIST). Recommendation for key management - part 1: General. Technical Report NIST Special Publication 800-57, National Institute of Standards and Technology, May 2006. `http://csrc.nist.gov/publications/nistpubs/800-57/SP800-57-Part1.pdf`.

[Nec94] V.I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55:165–172, 1994.

[NIS00] NIST. FIPS 186.2 digital signature standard (DSS). Technical report, NIST, 2000.

[OK63] Y. Ofman and A. Karatsuba. Multiplication of multidigit numbers on automata. *Soviet Physics - Doklady*, 7:595–596, 1963.

[OW99] P.C. van Oorschot and M. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12:1–28, 1999.

[Pag07] D. Page. Embedded implementation of LASH. Technical Report CSTR-07-003, University of Bristol, 2007.

BIBLIOGRAPHY

[Pap94] C.H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[PG05] D.S. Phatak and T. Goff. Fast modular reduction for large wordlengths via one linear and one cyclic convolution. In *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium*, pages 179–186, June 2005.

[PG06] D.S. Phatak and T. Goff. Low complexity algorithms for fast modular reduction: New results and a unified framework. Technical report, Computer Science and Electrical Engineering Department. University of Maryland, Baltimore County, Baltimore, MD 21250, 2006.

[Pin97] R. Pinch. *Mathematics for Cryptography*. Lecture notes for the University of Cambridge. University of Cambridge, 1997.

[Pre93] B. Preneel. Analysis and design of cryptographic hash functions. PhD Thesis, KU Leuven, 1993.

[RSA77] R.L. Rivest, A. Shamir, and L.M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, MIT, 1977.

[Sch85] R. Schoof. Elliptic curves over finite fields and the computation of square roots mod$p$. *Math. Comp.*, 44:483–494, 1985.

[Sch96] B. Schneir. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2 edition, 1996.

[Sco96] M.P. Scott. Comparison of methods for modular exponentiation on 32-bit intel 80x86 processors. Draft available for download from `ftp://ftp.computing.dcu.ie/pub/crypto/timings.ps`, 1996.

[SE91] C.P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Fundamentals of Computation Theory: 8th International Conference, FCT'91, Gosen, Germany, September 9-13, 1991: Proceedings*, 1991.

[SEC00] SECG. SEC2: Recommended elliptic curve domain parameters. Technical report, SECG, 2000. `http://www.secg.org`.

[Sha48] C.E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 1948.

[Sha95] A. Shamir. RSA for paranoids. *CryptoBytes*, 1(3):1–4, 1995.

[Sha01] C.E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, 2001.

[Sho97] V. Shoup. Lower bounds for discrete logarithms and related problems. In *Advances in Cryptology - EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266. Springer-Verlag, 1997.

[Sho06] V. Shoup. NTL: A library for doing number theory, 2006.

[Sil86] J.H. Silverman. *The Arithmetic of Elliptic Curves*. Springer, 1986.

[Sil92] J.H. Silverman. *Rational Points on Elliptic Curves*. Springer, 1992.

[Sip05] M. Sipser. *Introduction to the Theory of Computation*. International Thompson Publishing Inc., second edition, 2005.

[Sma01] N. Smart. The exact security of ECIES in the generic group model. *Cryptography and Coding*, 2001.

[Sma02] N.P. Smart. *Cryptography: An Introduction*. McGraw-Hill Education, 2002. Second edition is freely available online `http://www.cs.bris.ac.uk/~nigel/Crypto_Book/`.

[SS81] R. Schroeppel and A. Shamir. A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM Journal on Computing*, 10(3):456–464, 1981.

[Sti06] D.R. Stinson. *Cryptography, Theory and Practice*. Discrete Mathematics and its Applications. Chapman & Hall/CRC, 3 edition, 2006.

[Tes98] E.E. Teske. *Speeding Up Pollard's Rho Method for Computing Discrete Logarithms*. Springer, 1998.

[Tes01] E.E. Teske. On random walks for Pollard's Rho method. *Mathematics of Computation*, 70(234):809–825, 2001.

[Wag02] D. Wagner. A generalized birthday problem. In *Advances in Cryptology — CRYPTO 2002*, LNCS 2442, pages 288–303. Springer-Verlag, 2002.

[Wat69] W.C. Waterhouse. Abelian varieties over finite fields. *Ann. Sci. École Norm. Sup.*, 2:521–560, 1969.

[WLF+05] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the hash functions MD4 and RIPEMD. In *Advances in Cryptology — EUROCRYPT 2005*, LNCS 3494, pages 1–18. Springer-Verlag, 2005.

[WY05] X. Wang and H. Yu. How to break MD5 and other hash functions. In *Advances in Cryptology — EUROCRYPT 2005*, LNCS 3494, pages 19–35. Springer-Verlag, 2005.

[WYY05] X. Wang, H. Yu, and Y.L. Yin. Efficient collision search attacks on SHA-0. In *Advances in Cryptology — CRYPTO 2005*. Springer-Verlag, 2005.

[XW05] H. Yu X. Wang, Y. Yin. Finding collisions in the full SHA-1. In *Advances in Cryptology — CRYPTO 2005*. Springer-Verlag, 2005.

[Zur94] D. Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, August 1994.

Index

Epilogue

I end with the following verses from the ending of a long "scientific" poem written by اَلشَّاطِبِي *aš-šāṭibī* on The Quran (حِرْزُ الأَمَانِي وَوَجْهُ التَّهَانِي فِي القِرَاءَاتِ السَّبْع, 1173 verses). These verses include thanking God for easing the completion of an elegant work, praying for forgiveness for any mistakes or errors, and sending prayers upon the prophet Muhammad and his companions. I then finish with two prayers, that the prophet Muhammad – may peace and blessings be upon him – taught us, exalting God (Allah) and asking Him for forgiveness and mercy and bearing witness that there is no deity (worthy of worship) but Him – May Allah accept.

وَقَدْ وَفَّقَ اللَّهُ الْكَرِيمُ بِمَنِّهِ لِإِكْمَالِهَا حَسْنَاءَ مَيْمُونَةَ الْجِلَا
وَأَبْيَاتُهَا أَلْفٌ تَزِيدُ ثَلَاثَةً وَمَعْ مِائَةٍ سَبْعِينَ زُهْرًا وَكُمَّلَا
وَقَدْ كُسِيَتْ مِنْهَا الْمَعَانِي عِنَايَةً كَمَا عَرِيَتْ عَنْ كُلِّ عَوْرَاءَ مِفْصَلَا
وَتَمَّتْ بِحَمْدِ اللَّهِ فِي الْخَلْقِ سَهْلَةً مُنَزَّهَةً عَنْ مَنْطِقِ الْهُجْرِ مِقْوَلَا
وَلَكِنَّهَا تَبْغِي مِنَ النَّاسِ كُفْؤَهَا أَخَا ثِقَةٍ يَعْفُو وَيُغْضِي تَجَمُّلَا
وَلَيْسَ لَهَا إِلَّا ذُنُوبُ وَلِيِّهَا فَيَا طَيِّبَ الْأَنْفَاسِ أَحْسِنْ تَأَوُّلَا
وَقُلْ رَحِمَ الرَّحْمَنُ حَيًّا وَمَيِّتًا فَتًى كَانَ لِلْإِنْصَافِ وَالْحِلْمِ مَعْقِلَا
عَسَى اللَّهُ يُدْنِي سَعْيَهُ بِجَوَازِهِ وَإِنْ كَانَ زَيْفًا غَيْرَ خَافٍ مُزَلَّلَا
فَيَا خَيْرَ غَفَّارٍ وَيَا خَيْرَ رَاحِمٍ وَيَا خَيْرَ مَأْمُولٍ جَدًا وَتَفَضُّلَا
أَقِلْ عَثْرَتِي وَانْفَعْ بِهَا وَبِقَصْدِهَا حَنَانَيْكَ يَا اللَّهُ يَا رَافِعَ الْعُلَا
وَآخِرُ دَعْوَانَا بِتَوْفِيقِ رَبِّنَا أَنِ الْحَمْدُ لِلَّهِ الَّذِي وَحْدَهُ عَلَا
وَبَعْدُ صَلَاةُ اللَّهِ ثُمَّ سَلَامُهُ عَلَى سَيِّدِ الْخَلْقِ الرِّضَا مُتَنَخِّلَا
مُحَمَّدٍ الْمُخْتَارِ لِلْمَجْدِ كَعْبَةً صَلَاةً تُبَارِي الرِّيحَ مِسْكًا وَمَنْدَلَا
وَتُبْدِي عَلَى أَصْحَابِهِ نَفَحَاتِهَا بِغَيْرِ تَنَاهٍ زَرْنَبًا وَقَرَنْفُلَا

اللَّهُمَّ إِنِّي ظَلَمْتُ نَفْسِي ظُلْمًا كَثِيرًا وَلَا يَغْفِرُ الذُّنُوبَ إِلَّا أَنْتَ فَاغْفِرْ لِي مَغْفِرَةً مِنْ عِنْدِكَ وَارْحَمْنِي إِنَّكَ أَنْتَ الْغَفُورُ الرَّحِيم.

سُبْحَانَكَ اللَّهُمَّ وَبِحَمْدِك، أَشْهَدُ أَنْ لَا إِلَهَ إِلَّا أَنْت، أَسْتَغْفِرُكَ وَأَتُوبُ إِلَيْك.