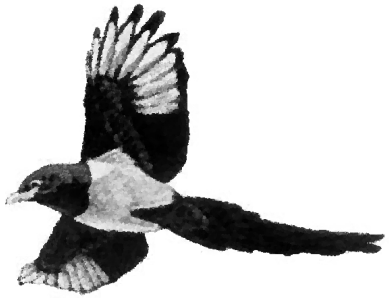
Speeding up Subgroup Cryptosystems



Martijn Stam

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Stam, Martijn

Speeding up Subgroup Cryptosystems / by Martijn Stam.
Eindhoven: Technische Universiteit Eindhoven, 2003
Proefschrift. – ISBN 90-386-0692-3
NUR 918
Subject headings: cryptology / finite fields / elliptic curves / recurrences
2000 Mathematics Subject Classification: 94A60, 11T71, 14H52, 94A55

Cover: Paul Verspaget

Printed by Universiteitsdrukkerij Technische Universiteit Eindhoven

Speeding up Subgroup Cryptosystems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. R.A. van Santen, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op woensdag 4 juni 2003 om 16.00 uur

door

Martijn Stam

geboren te Tilburg

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. A.K. Lenstra
en
prof.dr.ir. H.C.A van Tilborg

De totstandkoming van dit proefschrift is mede mogelijk gemaakt door de STW.

Opgedragen aan mijn lieve oma

Geertje den Ouden-Heijkoop

Preface

The title of this thesis contains the word cryptosystem. Originally cryptosystems were used to make messages unintelligible for prying eyes; only the intended receiver could make up what the message said. In the very early beginnings this could be done by heart, then by hand using pencil and paper and later using tables and sophisticated mechanical devices. Today computers are used most of the times to implement cryptosystems that can provide confidentiality but also authenticity and other services. Authenticity is comparable to signing a letter by hand; the receiver wants to be sure that the message does originate from whom he thinks it does. A signature provides legal proof it does. Not surprisingly, this type of cryptosystem is commonly referred to as a digital signature, even though it bears no structural resemblance to handwritten signatures.

Subgroup cryptosystems are cryptosystems based on a specific number theoretic assumption concerning the discrete logarithm problem. Ignoring the specifics for now, the main point is that an honest user of such a cryptosystem has to perform a task called exponentiation to, say, sign a message or decrypt what is called a ciphertext. Such an exponentiation can be done in a reasonable amount of time. An adversary who wants to break the system —by forging a signature or reading a message not intended for him— has to compute a discrete logarithm. This is assumed to be very hard, probably taking him the rest of the millennium.

Speeding up a subgroup cryptosystem is aimed at reducing the time a user needs to wait, for instance to encrypt a message or verify a signature. This requires fast exponentiation routines, either by performing less group operations or by reducing the costs of the group operations.

Chapters 2 and 3 concentrate on the first possibility: reducing the number of group operations. Chapter 2 gives an overview of several exponentiation routines and some relevant results in the theory of addition chains. Chapter 3 extends the notion of addition chain to second and third order. After reviewing several binary algorithms and Montgomery's Euclidean algorithm for second degree, a third degree adaptation of the latter is presented, based on joint work with Arjen K. Lenstra [**180**]. Proposition 3.34 also stems from this work.

The approach of reducing the costs of the group operation is taken in Chapters 4 and 5. In Chapter 4 several subgroups of finite field extension are discussed. Sections 4.3 and 4.5 show that in special subgroups of second respectively sixth degree extensions squaring is cheaper than in the entire field. This includes joint work with Arjen K. Lenstra [**181**]. The speedup of XTR, a related cryptosystem, using the

techniques of Chapter 3 is discussed in Section 4.6 and is based on the aforementioned [**180**]. Chapter 5 discusses an improved Montgomery-like representation for elliptic curves over binary fields [**179**]. An overview of similar techniques for curves over prime fields is included.

The final chapter of this thesis is not concerned with subgroups but with black box groups. Black box groups are groups for which a black box is used to implement the group operations. Chapter 6 discusses efficient implementation of a specific black box secret sharing scheme over an arbitrary finite abelian group. This requires careful balancing between finding short addition chains and picking appropriate parameters. This is joint work with R. Cramer.

Contents

List of Algorithms

1

Introduction

1.1. Cryptology

1.1.1. An Ancient Art. An ancient art, cryptography was already practised by the Romans. Messages were made unintelligible to enemies by shifting the letters of the alphabet by three, so writing 'd' where 'a' is meant, 'e' where 'b' is meant, etc. Such a method to hide messages from the opponent is called a cipher. The shift-by-three cipher is called the Caesar cipher —after Julius Caesar— and is one of the oldest ciphers known. In the centuries that followed cryptology remained focused on finding ways to send messages that could only be read by the intended recipient (cryptography): any eavesdropper should be unable to determine the original message. Attempts to try and break such systems were mounted nevertheless (cryptanalysis). Many ciphers were invented and subsequently broken.

All the ciphers had one thing in common. If the intended recipient can decrypt a ciphertext but an eavesdropper cannot, then the recipient must possess something the eavesdropper does not. Over the years this something has taken several forms. For instance, for the Caesar cipher it is the knowledge that the ciphertext was obtained from the plaintext by shifting the letters of the alphabet by three. For the Enigma, a famous cipher used by the Germans during the Second World War, both sender and recipient had a special mechanical device. In addition, the recipient needed to know the initial setting of the machine used by the sender. By changing the initial setting the device could be used over and over again, assuming both sender and receiver used the same settings. In a way, this transforms the problem of securely sending messages to agreeing upon initial states, which can be done before the message is known.

1.1.2. A Scientific Approach. The security of a cipher such as the Enigma seemed to rely on two factors. An adversary did not have an Enigma device, nor did he know the initial setting used. Kerckhoffs' principle states that the security should reside in the secrecy of the initial setting alone (in fact, the allies did manage to obtain an Enigma device and subsequently they broke the cipher, allowing them to effectively eavesdrop on a significant part of Germany's confidential communications and giving them an important strategical advantage). For a precise description of Kerckhoffs' principle a more formal description of what constitutes a cipher is required. Here the notion of an algorithm is useful.

An algorithm is similar to a recipe: it is a "finite set of rules that gives a sequence of operations for solving a specific type of problem" (Knuth, [**89**, p. 4]).

An algorithm should have some input, some output and the number of operations should be finite. An algorithm should also be definite (meaning that it is crystal clear what has to be done) and effective (it should be known how to perform all the steps).

A cipher is made up of an encryption algorithm and a decryption algorithm. The encryption algorithm has as inputs the message that needs to be transmitted, called the plaintext, and the initial setting, called the key. It outputs a ciphertext. The decryption algorithm has as inputs the ciphertext and the key and outputs the plaintext. Kerckhoffs' principle states that even when the encryption and decryption algorithms are given to an adversary, he should not be able to determine the plaintext from a ciphertext without knowing the key. If the inner workings of a cryptosystem are not be disclosed to the general public, this is often derogatorily called security through obscurity.

In the 1940s, Shannon laid the scientific foundations of cryptology by applying his newly found information theory to encryption systems [**168, 169**]. Shannon's work provides a useful measure of the amount of information in a message and a key. The main theorem of information-theoretic cryptography, is that the amount of information that can be sent absolutely secure is at most as much as the information present in the key. If more information is transmitted, an adversary can theoretically retrieve part of the secret message (when given unlimited computing power). In practice, adversaries are limited in their capabilities, which allows sending messages longer than the key. But if there is a flaw in the system, a malicious eavesdropper might be successful. Shannon's pioneering work remained confidential for a few years and cryptology remained an almost exclusively military affair with a strong emphasis on the confidential transmission of data for a few decades.

1.1.3. A Modern Science. About thirty years ago this has changed and today a large crypto-community is at work in academia and industry. This change came about for two important reasons: the rise of the computer (and the Internet) and the introduction of public key cryptography by Diffie and Hellman [**58**]. Traditionally, cryptography was based on the premise that the sender and the intended receiver shared a common secret in advance. This secret was then used to encrypt and decrypt the message. Public key cryptography uses a different point of departure. The receiver has some secret information which she keeps to herself, but a related piece of information is made known to the world. The secret information is called the private key, the information that is known to everyone the public key. A sender uses the public key to encrypt the message. The recipient uses her private key to decrypt the ciphertext. An important advantage of this setup over the 'traditional' one, is that the sender and receiver no longer need to share a common secret. In fact, it suffices if the sender would look up the public key of the receiver in a directory (phone book), as long as it is clear that the public key truly belongs to the person it purports to.

At the core of public key cryptography are one-way functions. A one-way function is a function that is easy to compute but hard to invert, at least in practice. The public key is often a one-way function applied to the private key. Diffie and

Hellman proposed to use exponentiation of integers modulo a prime as one-way function (the concept of a one-way function was known before). For encryption a *trapdoor* one-way function is used: the function is hard to invert, except for someone holding some extra piece of information. Although Diffie and Hellman introduced the concept of a trapdoor one-way function, the world had to wait until 1978 for Rivest, Shamir and Adleman to publish their paper [**155**] that contained the first explicit trapdoor one-way function. It was, or rather is, based on the assumed intractability to extract roots modulo a composite. The factorization of the modulus allows easy root extraction, thus building in a trapdoor. The assumed intractability of factoring assures that only the person who constructs the modulus by multiplying two (or more) primes, knows this factorization.

A disadvantage of public key cryptography is that it is unknown how to prove that a function is one-way. Proving that f can be computed efficiently is usually not very difficult, but proving it is hard to invert is, well, hard. In practice, functions are used that most people believe are hard to invert. If someone manages to find a way to invert such a function efficiently (or significantly more efficient than previously known), the cryptosystems that are based on this assumption fall apart or, in the optimistic scenario, need to increase their key size.

The beauty of public key cryptography is that it not only changes the way we encrypt messages, it also opens up the way for a variety of other possibilities. Still reasonably close to encryption is key agreement: two parties want to agree on a key, but they have to start from scratch and can only communicate over an (authenticated) public channel (compare this with arranging a date in the presence of your evil ex). A revolutionary new application are digital signatures. Diffie and Hellman realized that, since only the 'owner' of the trapdoor could invert the one-way function, the ability to compute the inverse of the message could effectively prove that the message originates from the owner and also that the message has not been tampered with. If hand-written signatures are used, it is actually much more difficult to show that a message has (not) been tampered with.

In the 1980s cryptology really took off in the scientific community. New cryptosystems were proposed, proven secure using a not-quite-right notion of security, and subsequently broken. In the late 1980s the definitions appeared what constitutes a good public key encryption scheme or a good digital signature scheme. Essential in these definitions is a separation of concerns. There is broad description of the type of scheme. For instance for a public key encryption scheme there are three algorithms: one for encryption, one for decryption and one for key generation (this is suddenly more important when a public key algorithm is used). Second, there is a list of possible adversaries. There is for instance a difference between an adversary who only knows the public key and an adversary who can decrypt almost all ciphertexts. Finally, there is a list of goals for an adversary. Does he want to retrieve the entire ciphertext or is he satisfied if he can check whether a certain ciphertext and plaintext belong together? A cryptosystem is secure if even the most powerful adversary having only this most modest goal cannot succeed. A proof of security shows that if the adversary would be successful, he would also be able to solve some

well-known mathematical problem that (almost) everyone believes to be very hard, such as factoring integers.

At the same time, new exciting applications emerged, such as zero-knowledge proofs and multi-party computation. A zero-knowledge proof allows someone to convince another of a statement without revealing anything apart from the statement itself and its validity. Loosely speaking, multi-party computation involves a large number of players, each of whom has his own input and who together want to compute something that depends on each of the inputs, for instance a list with all their inputs sorted lexicographically. The challenge arises from the fact that no player wants to reveal the specific input he has and some of the players might be corrupt. In both zero-knowledge proofs and multi-party computation choices have to be made: what to protect information-theoretically and what to protect computationally. The theory of both areas is still developing further.

The *Handbook of Applied Cryptography* [**121**] and Schneier's *Applied Cryptography* [**160**] both illustrate the range of cryptography. It is interesting to see that the more recent *Handbook* is more structured and scientific, almost as a silent witness to the developments of cryptology. Bellare and Goldwasser's lecture notes [**66**] give a more theoretical treatment of (mainly) public key cryptography.

1.1.4. Cryptography in Practice. The advances in cryptology resulted that cryptography slowly sips through in everyday life, albeit most often unnoticed. Especially in banking applications and on the Internet cryptography plays an important role. Encryption of large portions of data is still based on modern versions of ciphers where sender and recipient use the same key. But public key cryptosystems are used to 'transmit' this key and digital signatures are used to prove authenticity of the message and protect it from tampering. Today digital signatures have a legal status in several countries. Cryptology also finds it way on an increasing number of smart cards. These cards can be used as an electronic purse or as a means of identification, needed before receiving some privilege.

In all cases the attractiveness of using a public key cryptosystem strongly depends on the time the system takes. If it would take several minutes to pay with a smart card, hardly anybody will use it. An obvious solution is to use a faster smart card, but a faster smart card is undoubtedly more expensive. A better approach is to reduce the amount of work the smart card has to do, without affecting the security of the scheme (the smart card is only used as an example). The efficiency of cryptosystems has been studied for a long time. Almost every new cryptosystem calls for a new efficiency optimization and sometimes a tweak of the cryptosystem is more efficient and just as secure.

In this light the work of this thesis should be seen: the efficiency of several cryptosystems is examined and optimized.

As a final remark of this section, a word on a problem not addressed in this thesis, but that forms a serious obstacle in the widespread acceptance of cryptography. The weakest point of almost any system is often the human factor. Cryptosystems seem to suffer from this 'condition' even more than any other system. A small anecdote from everyday life will serve well to illustrate this.

Bike theft is a serious problem in the Netherlands. A chain of hardened steel used to attach the bike to some fixed object, such as a lamppost, might discourage potential thieves. A foreigner not quite aware of the Dutch situation once used the chain to attach his bike to a small pole only standing one meter high. Without too much effort a thief can lift the chain over the pole, thereby releasing the bike from the pole and enabling him to steal the bike (he can worry about detaching the chain from the bike in his workshop). Even though the chain is good, it fails to give proper protection as a result of improper use. Fortunately, if you do not lock your bike properly, it gets stolen, which you will notice. Next time, common sense suffices to make proper use of chain.

If you use a weak cryptosystem it is not clear how to find out that your opponent is actually reading all your messages. As a result, there is much less awareness of properly using cryptosystems. An extra problem is that most of what a cryptosystem does, happens somewhere in the background and a computer is used (and needed) to check the result. There is very little the average user can do to check whether his computer program actually does what it should do (it is already quite difficult to distinguish between a compressed file and an encrypted compressed file).

1.2. Subgroup Cryptosystems

1.2.1. Exponentiation. For a long time after the introduction of public key cryptography in the late 1970s, it was hoped —and believed— that a rich and varied field of one-way functions would be discovered. In their seminal paper, Diffie and Hellman [**58**] exploit the (assumed) hardness of taking discrete logarithms. Shortly after, Rivest et al. [**155**] describe a system ultimately based on the hardness of factoring integers (once more assumed). Numerous attempts have been made to base cryptosystems on problems as diverse as knapsacks, lattices, braid groups, coding theory, and non-commutative groups, but the mainstay of todays public key cryptosystems is based on the hardness of either factoring integers or computing discrete logarithms in finite abelian groups of known order. In this thesis we will concentrate on speeding up the latter type of cryptosystem.

For now, and for the rest of this thesis, let G denote a finite abelian group and let $\langle g \rangle$ denote the subgroup of G generated by some element $g \in \mathsf{G}$. If a group is cyclic of known order x this will be made explicit by the notation G_x. The group G will be denoted multiplicatively; the group multiplication will either be denoted explicitly by $\cdot$ or by juxtaposition. The identity element is denoted by *id*. For historical reasons the group G will be represented additively in the last two chapters of this thesis.

For an element $g \in \mathsf{G}$ and a positive integer n performing an exponentiation means computing

$$g^n = \overbrace{g \cdot g \cdot \ldots \cdot g}^{n} \ . \tag{1}$$

This is well defined since $\cdot$ is associative. Commutativity is not needed here, nor the existence of inverses or the identity. This leads some authors to consider the problem of exponentiation over monoids instead of finite abelian groups. Alternatively, one

can also emphasize on the group being a $\mathbb{Z}$-module, which will have applications in Chapter 6. For negative integers n the exponentiation g^n is defined as $(g^{-1})^{-n}$, where g^{-1} is the inverse of g in the group. Finally $g^0 = id$ by definition, so the exponents behave as elements in the additive group $\mathbb{Z}$ (modulo the group order).

The naive way to perform an exponentiation is by repeated multiplication, as described in the algorithm below. (The style of presentation of algorithms is based on Knuth's example [**89**].)

ALGORITHM 1.1 (Unary Exponentiation).
The algorithm takes a group element g and a nonnegative integer n as input and returns g^n. The algorithm maintains as invariant:

$$0 \leq a \leq n, \quad A = g^a \ .$$

1. [Initialization] Set $a \leftarrow 0$ and $A \leftarrow id$.
2. [Finished?] If $a = n$ terminate with output A.
3. [Increase a] Set $A \leftarrow gA$ and $a \leftarrow a + 1$ and go back to the previous step.

This algorithm takes $n - 1$ group multiplications for positive n. If n is a k-bit exponent, that is $k = \lceil \lg(|n| + 1) \rceil$ or, if $n > 0$, $k = \lfloor \lg n \rfloor + 1$, we see that the runtime of Algorithm 1.1 is exponential in the bitlength k, which is out of the question for values of k that are relevant for cryptographic applications. In Chapter 2 efficient algorithms will be reviewed that require at most $2k$ multiplications for an exponentiation. (The notation lg stands for logarithm to base 2. The natural logarithm is denoted by ln. When the base is inconsequential, log is used. The function $\lceil x \rceil$ returns the smallest integer $\geq x$ and $\lfloor x \rfloor$ returns the greatest integer $\leq x$.)

1.2.2. The Discrete Logarithm Problem (DLP). Whereas computing $h = g^n$ is fairly easy, the opposite of computing n when given g and h can be quite hard. The value n is called the discrete logarithm of h with respect to g, or $n = \log_g h$. Computing n is aptly known as the discrete logarithm problem (DLP). We will assume that the DLP is hard in the groups discussed in this thesis.

Related to the discrete logarithm problem are the computational and decisional Diffie-Hellman problems. The computational Diffie-Hellman (CDH) problem entails computing g^{nm} given g, g^n, and g^m. The decisional Diffie-Hellman (DDH) problem requires one to determine whether a given group element h equals g^{nm}, where g, g^n, and g^m are also given.

Although the descriptions of the problems above are easy enough, the formal definition of hardness of for instance the DLP is somewhat involved. Typically it involves a family $\mathcal{G}$ of finite cyclic groups and a security parameter k. The larger k, the bigger the cardinality of the group chosen from $\mathcal{G}$ and the smaller the success rate for an adversary trying to find discrete logarithms in the group. The subtleties concerning assumptions based on the DLP are discussed in full by Sadeghi and Steiner [**158**]. McCurley [**120**] and Odlyzko [**139**] review the DLP itself and Boneh [**24**] gives extensive treatment of the DDH problem.

The security of a lot of cryptographic protocols relies on the assumed hardness of the CDH or DDH problem. Only assuming the hardness of the DLP is not sufficient for these protocols. In some groups the DDH problem is easy, but the CDH problem is still assumed to be hard. There are also some (black box) separation results.

Diffie and Hellman [**58**] suggested the use of the multiplicative group of the integers modulo a prime or more generally, the multiplicative group of a finite field for their famous key agreement protocol. We will write $\mathbb{Z}_p$ for the integers modulo p and $\mathbb{F}_{p^d}$ for a finite field of characteristic p and extension degree d. The multiplicative groups are $\mathbb{Z}_p^*$ and $\mathbb{F}_{p^d}^*$. Schnorr [**161**] proposed to use only a subgroup of prime order q of $\mathbb{Z}_p^*$. The exponents can be reduced modulo q, which will typically be significantly shorter than p, without affecting the security of the system. A natural generalization is a prime order subgroup of $\mathbb{F}_{p^d}^*$. Nowadays it is common to use a prime order subgroup of a larger, better understood group, hence the name subgroup cryptosystem. Section 1.2.4 contains some information on attacking the DLP in these groups.

In 1985 Koblitz [**93**] and Miller [**126**] suggested to use groups based on elliptic curves. The advantage of elliptic curves is that no subexponential algorithm is known to solve the DLP over an elliptic curve. As a consequence the security parameter k can be chosen relatively small (or the group order can be chosen small). In Chapter 5 we discuss some aspects of the efficient implementation of elliptic curve cryptograpy. A more thorough treatment is given by Blake et al. [**17**].

The use of hyperelliptic curves in cryptography was proposed later [**94**]. Efficient arithmetic on hyperelliptic curves is considered by Lange [**100**].

1.2.3. Cryptographic Applications. Of the big pool of cryptosystems based on the DLP, CDH and DDH assumptions, we will pick just a few, that cover the basics of key agreement, encryption and signatures. Again, this is just a very small selection to illustrate DLP-based cryptography and many other interesting applications of subgroup cryptosystems exist.

All the examples below involve two parties, called Anna and Bob. We will assume that they have already agreed upon some group G_q of order q and a generator g of G_q. The discrete logarithm problem in G_q is assumed to be hard. For simplicity we will assume that q is a prime.

Key Agreement. The idea of key agreement is that two parties that start without sharing secret information, arrive at a common secret string at the end of the key agreement protocol. The common secret string can subsequently be used as the key for a symmetric encryption scheme. The scheme below is by Diffie and Hellman [**58**].

Anna picks some element $a \in \mathbb{Z}_q$ and she sends $A = g^a$ to Bob. At the same time, Bob picks some element $b \in \mathbb{Z}_q$ and he sends $B = g^b$ to Anna. Both are now capable of computing g^{ab}. Bob by computing A^b and Anna by B^a. An adversary knowing g and G_q and observing g^a and g^b needs to solve the CDH problem to obtain the agreed upon key g^{ab}. The assumed hardness of the CDH problem implies security of the key against eavesdroppers.

Encryption. It took surprisingly long before a 'pure' encryption scheme based on the DLP appeared in the literature. In 1985, ElGamal [**60**] proposed an encryption scheme that is basically a Diffie-Hellman key agreement followed by group multiplication of the message and the key. (Sometimes Diffie-Hellman key agreement followed by an encryption 'outside the group' is called hybrid ElGamal encryption.)

Anna begins by picking an arbitrary $a \in \mathbb{Z}_q$ as her private key. She publishes $A = g^a$, her public key. When Bob wants to send a message to Anna, he first encodes the message as an element $m \in \mathsf{G}_q$. He picks $r \in \mathbb{Z}_q$ at random, computes g^r and A^r and sends the pair (mA^r, g^r) to Anna. Upon receiving this pair, Anna can compute A^r herself as $(g^r)^a$, after which recovery of the message follows by a division.

An interesting property of ElGamal encryption is that it is homomorphic: if encryptions of m_1 and m_2 are given, the pointwise product of the ciphertexts decrypts to $m_1 m_2$. ElGamal encryption is semantically secure under the DDH assumption, but it is important that the message is really an element in G_q. In particular it is required that the message has order q. Embedding the message in a (cyclic) supergroup of G_q in such a way that m does not necessarily have order q is a common mistake [**109, 157**]. Unfortunately, for subgroup cryptosystems it is not always known how to efficiently embed a message in G_q.

Signature Schemes. There does not seem to be a single signature scheme based on discrete logarithms that is used almost exclusively. The Digital Signature Algorithm (DSA) has the benefit of being standardized by the NIST [**136**]. There should be an efficient map embedding the group G_x into $\mathbb{Z}_x$, such that first exponentiating and then applying this map is a one-way function (the composition of a one-way function and an ordinary function is not necessarily one-way). To avoid technical problems, we assume that the group has prime order. The public key of Anna is $A = g^a$ and her private key is a.

To sign a message m, Anna first picks $r \in \mathbb{Z}_x$ at random. She computes $R = g^r$ and $s = r^{-1}(m + aR) \bmod x$, where R is mapped into $\mathbb{Z}_x$ and m is hashed into $\mathbb{Z}_x$ using a cryptographically strong hash function. Anna sends the pair (s, R) along with the message to Bob. If Bob knows Anna's public key, he can verify the signature by computing $g^{ms^{-1}} A^{Rs^{-1}}$. If this value equals R, the signature is valid.

1.2.4. Attacking the DLP.

Algorithms that attempt to solve the DLP can be classified according to the extent in which the group structure (representation) is exploited. Shoup [**172**] shows that if the group is regarded as a black box (cf. 6), the number of group operations to solve the DLP in a group of order n is at least $O(\sqrt{n})$. This lower bound is achieved by Shank's deterministic baby-step-giant-step method [**167**], also requiring $O(\sqrt{n})$ memory. Only requiring constant memory but probabilistic in nature, are Pollard's ρ and λ methods [**153, 154**]. The ρ method runs in time $O(\sqrt{n})$ and the λ method in $O(\sqrt{B})$ where B is an upper bound on the exponent (possibly smaller than n, moreover the group order does not need to be known in advance).

Pohlig and Hellman [**152**] show that the discrete logarithm problem in a cyclic group of composite order decomposes into the DLPs in the respective subgroups of prime order. In theory this means that the group order should contain at least one large prime factor to obtain a secure cryptosystem. In practice, the order of the subgroup that is used, is often chosen to be prime itself (there are exceptions, notably for protocols that also rely on the intractability of factoring). As a notational convenience, G_q will always denote a subgroup of prime order q and, unless specified otherwise, q will be a k-bit prime. Current practice based on conservative estimates of the computational power (and prowess) of an adversary, suggests to use k in the range 160 to 200.

For finite fields it is possible to use the index calculus method [**70, 159**] to compute discrete logarithms. For fixed extension degree d and characteristic p tending to infinity, the heuristic asymptotic runtime of the index calculus method is $\exp((1.923 + o(1))(\ln p^d)^{1/3}(\ln\ln p^d)^{2/3})$. Coppersmith [**49**] describes an improvement for $p = 2$ and d growing. Later work generalizes this to fixed p and growing d with a heuristic asymptotic runtime of $\exp((1.526 + o(1))(\ln p^d)^{1/3}(\ln\ln p^d)^{2/3})$. Similar runtimes can be obtained for d fixed and p growing but of a special form (for instance close to a power of two). For fixed d, it is customary to pick p such that $d \lg p$ is in the range 1024 to 4096. These figures are based on experimental results and runtimes of the index calculus method (also when applied to factor integers). More detailed analysis and recommendations of key sizes can be found in the literature [**105, 110**].

Lenstra [**104**] discusses the implications the Pohlig-Hellman algorithm [**152**] has for the discrete logarithm problem in finite fields $\mathbb{F}_{p^d}$. The multiplicative group $\mathbb{F}^*_{p^d}$ has order $p^d - 1$, which factors as the product of all $\Phi_s(p)$ where s is a divisor of d and $\Phi_s(p)$ is the s-th cyclotomic polynomial evaluated in p (for background on finite fields and cyclotomic polynomials, Lidl and Niederreiter [**111**] are quite thorough). The subgroup of order $\Phi_s(p)$ can be efficiently embedded in $\mathbb{F}^*_{p^s}$. Hence, for proper divisors s of d the resulting discrete logarithm problem reduces to the discrete logarithm in a proper subfield of $\mathbb{F}_{p^d}$ which is widely believed to be considerably easier than the problem in the extension field. The hardness of computing logarithms in the group $\mathbb{F}^*_{p^d}$ must therefore reside in its cyclotomic subgroup of order $\Phi_d(p)$.

In line with the above, in Chapter 4 the discrete logarithm problem in finite fields will be considered in a subgroup G_q of large prime order q and with $\mathsf{G}_q \subseteq \mathsf{G}_{\Phi_d(p)}$. The necessary condition $q|\Phi_d(p)$ also proves sufficient, in that G_q cannot be embedded into any proper subfield of $\mathbb{F}_{p^d}$ due to the following lemma [**104**, Lemma 2.4].

LEMMA 1.2 (Lenstra). *Let $q > d$ be a prime factor of $\Phi_d(p)$. Then q does not divide any $\Phi_s(p)$ for divisors s of d with $s < d$.*

Menezes and Vanstone [**122**] suggested that using the group $\mathsf{G}_{p^2-p+1} \subseteq \mathbb{F}^*_{p^6}$ might just be using supersingular curves in disguise. Indeed, the order of the group, $p^2 - p + 1$, corresponds nicely with the order of a supersingular curve over $\mathbb{F}_{p^2}$ and the Weil or Tate pairing gives an embedding from the supersingular curve into the cyclotomic subgroup of $\mathbb{F}^*_{p^6}$. It has been shown by Verheul [**194**] that the existence

of an efficient homomorphism from the group $\mathsf{G}_{p^2-p+1} \subseteq \mathbb{F}_{p^6}^*$ to a supersingular curve implies that the CDH problem can be efficiently solved for both groups as well. This provides strong evidence that the group G_{p^2-p+1} is not a supersingular elliptic curve of the same cardinality in disguise.

1.2.5. Attacking a Subgroup Cryptosystem. Suppose that a cryptosystem is provably secure under the DLP assumption (for some suitable definition of secure) and that G_q is a group in which computing discrete logarithms is infeasible. In some cases, it might still be possible for an adversary to launch an attack on the system, especially if the users are not cautious. The two examples below show that even with mathematically sound cryptosystems there are pitfalls to avoid when using the system. (Our examples are slightly artificial, but the techniques do carry over to more realistic settings.)

Suppose that Anna and Bob are performing a Diffie-Hellman key agreement protocol (Section 1.2.3) and that Bob wants to know the discrete logarithm a of the value $A = g^a$ Anna sends him. Suppose that somehow Anna sends Bob the key she thinks they have agreed upon *as soon* as she has computed it. There are two ways Bob might be able to learn a, one active and one passive method.

To begin with the passive method, Anna has to compute B^a, where B is the group element g^b given by Bob. Because Anna sends the result B^a to Bob as soon as she is finished computing it, Bob knows exactly how long it took Anna to compute B^a. If Anna has used Algorithm 1.1 to compute B^a, the time taken is proportional to $a-1$. The time needed by Anna therefore reveals a to Bob. Even though in practice Anna will not use Algorithm 1.1 —and if she does, then Bob can be expected to run through the algorithm in the same time as Anna, obtaining a anyway—, information on how long it took Anna to compute B^a using more advanced algorithms can be very dangerous in the hands of an adversary. Especially smart cards suffer from so called side-channel attacks, where an adversary monitors the power consumption and duration of cryptographic operations using a secret key. Several countermeasures are proposed in the literature to make presumably safer exponentiation routines, but we will largely ignore this issue in Chapters 2 and 3.

Bob can also launch an active attack to try and find out a. Suppose that Bob sends Anna some value B that is not in G_q but in another group for which Bob knows how to solve the DLP. If Anna computes B^a and returns this value to Bob, he will be able to determine a. If Anna would check that $B \in \mathsf{G}_q$, she does not have to fear this attack (since q is prime and the DLP self-reducible). Checking that B is indeed an element of G_q can be quite expensive. Often it is necessary to first check that B is in some (cyclic) supergroup G' of G_q and subsequently that the order of B is indeed q. The latter check usually requires a full exponentiation if G_q is a proper subgroup of G'. The cost of membership testing of G' differs greatly per cryptosystem. If the order of G' is only a small factor c bigger than that of G_q, an alternative solution is enforcing an element $B \in \mathsf{G}'$ into G_q by computing B^c and use this value in the remainder of the protocol. In Chapter 4 we will address the issue in more detail for the subgroups under consideration.

1.3. Recurrences

Exponentiation in a group can be generalized to computing remote terms in a (linear) recurrence. There are several proposals to replace the exponentiation in cryptographic protocols by the evaluation of remote terms in a recurrence relation.

The most famous recurrence relation is without doubt the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, etc. The initial values are $F_0 = 0$ and $F_1 = 1$. Subsequent terms are defined recursively by $F_{i+1} = F_i + F_{i-1}$ for $i > 0$. The Fibonacci sequence is an example of a second order linear recurrence over the integers. Lucas [**116**] has studied the sequence extensively and introduced the following two families of functions (now known as Lucas functions):

$$U_{n+2}(P,Q) = PU_{n+1} - QU_n, \quad U_0(P,Q) = 0, \quad U_1(P,Q) = 1 \; ; \tag{2a}$$

$$V_{n+2}(P,Q) = PV_{n+1} - QV_n, \quad V_0(P,Q) = 2, \quad V_1(P,Q) = P \; . \tag{2b}$$

The Fibonacci numbers correspond to $U_n(1,-1)$. The numbers occurring in the sequence $V_n(1,-1)$ are known as Lucas numbers. Computing the Lucas functions modulo a prime or an RSA-like composite has cryptographic applications, such as the LUC cryptosystem [**175**] (discussed in Section 4.4).

Using modular arithmetic for the computation of the Lucas numbers shows that it is possible to define recurrences over other commutative unital rings. Another example are the Chebyshev polynomials $T_n(x) = \cos(n \cos^{-1}(x))$. They satisfy the recurrence $T_m(x)T_n(x) = \frac{1}{2}(T_{n+m}(x) + T_{n-m}(x))$ over $\mathbb{Q}[x]$. Filling in $m = 1$ gives the linear recurrence $T_{n+1} = 2xT_n - T_{n-1}$. This formula already incorporates the initial value $T_1 = x$. The other initial value is $T_0 = 1$.

Another possibility is considering higher order recurrences. An example of a third order linear recurrence is the Perrin sequence [**1, 145**], defined by

$$A_{n+3} = A_{n+1} + A_n \tag{3}$$

and initial condition $A_0 = 3, A_1 = 0$, and $A_2 = 2$. There does not seem to exist a direct application of the Perrin sequence, but there is a pseudo-primality test based on the Perrin sequence [**1**] (similar to a test based on the Lucas functions). In Chapter 3 we use the adjective 'Perrin' for some third order analogues of second order objects (that bear the adjective 'Lucas').

The recurrence relations we are interested in are of a slightly different form. For a second order recurrence we consider the case where the term v_{n+m} can be computed as a function of v_n, v_m, and v_{n-m}. (Loosely speaking, this corresponds to being able to multiply two elements in a group only if their quotient is already known.) Initial values v_0 and v_1 are given. Moreover, we will assume that v_0 is fixed. It will be denoted by $v_0 = id$ (corresponding to $g^0 = id$). The value v_1 is called the base; the subscript will usually be dropped. An example of such a recurrence are the Chebyshev polynomials. For a third order recurrence the term c_{n+m} can be computed as a function of c_n, c_m, c_{n-m}, and c_{n-2m}. The initial values are $c_0 = id, c_1$ and c_{-1}.

In Chapter 3 we will discuss efficient algorithms to evaluate remote terms in one of these recurrences (not necessarily linear). In Chapter 4 second and third

order recurrences for LUC respectively XTR [**107**] are encountered. Both can be rewritten as linear recurrences. The second order recurrences in Chapter 5 arising from Montgomery's elliptic curve representation are not linear.

As already mentioned, evaluating the n-th term in a recurrence can be regarded as a generalization of exponentiation. Similarly, the DLP can also be reformulated: given a recurrence (not necessarily linear, but preferably cyclic), and a ring element r in the sequence, find n such that $x_n = r$. In a group $\langle g \rangle$ the discrete logarithm is unique modulo the order of the group. This property is lost even for cyclic linear recurrences. The CDH and DDH problems allow even more freedom. Given two ring elements r_1 and r_2, the computational Diffie-Hellman problem requires finding some r_3 such that n_1 and n_2 exist satisfying $r_1 = x_{n_1}, r_2 = x_{n_2}$, and $r_3 = x_{n_1 n_2}$.

Although higher order recurrences are not used in this thesis, we will briefly discuss some relevant theory. This helps to understand the DLP based on recurrences and gives some backdrop for Chapters 3 and 4. There is also a small link with Chapter 6. Higher order recurrences have cryptographic applications of their own in stream cipher design and pseudorandomness generation.

A k-th order linear recurrence over a ring R is defined by

$$x_{n+k} = f_0 x_n + f_1 x_{n+1} + \cdots + f_{k-1} x_{n+k-1}, \quad n \geq 0 \ , \tag{4}$$

where the coefficients $f_i \in R$ for $i = 0, \ldots, k-1$ are fixed. Initial values $x_i \in R$ for $i = 0, \ldots k-1$ are also given. The companion matrix of the recurrence (4) is defined by

$$M = \begin{pmatrix} 0 & 0 & \ldots & 0 & f_0 \\ 1 & 0 & \ldots & 0 & f_1 \\ 0 & 1 & \ldots & 0 & f_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & f_{k-1} \end{pmatrix} \ . \tag{5}$$

This matrix has the property that, for $n \geq 0$, the vector $(x_n, \ldots, x_{n+k-1})$ can be obtained from the vector of initial values $(x_0, \ldots, x_{k-1})$ using the following formula:

$$(x_n, \ldots, x_{n+k-1}) = (x_0, \ldots, x_{k-1}) M^n \ . \tag{6}$$

If f_0 has a multiplicative inverse in R (equivalently, if M is invertible), the recurrence is extended to negative terms by allowing $n < 0$ in (6). A recurrence is cyclic iff $M^e = I$ for some positive integer e. The characteristic polynomial of the recurrence is $f(X) = X^k - f_{k-1} X^{k-1} - \cdots - f_1 X - f_0$.

The DLP based on a recurrence can be reformulated in terms of the ring $R[X]/(f)$. Let the linear transform L from $R[X]/(f)$ to R be defined by $L(X^i) = x_i$ for $0 \leq i < k$, so L is based on the initial values of the recurrence. The transform L is well-defined regardless of the initial values iff f is irreducible over R. By induction it follows that in this case also $L(X^i) = x_i$ for $i \geq k$. The DLP for a recurrence therefore reduces to "inverting" L, that is given $r \in R$, computing an element $s \in \langle X \rangle \subseteq R[X]/(f)$ subject to $L(s) = r$, and solving the DLP in $R[X]/(f)$. For the CDH and DDH problem similar arguments hold.

In the special case that R is a finite field $\mathbb{F}_{p^d}$ and f is irreducible over R of degree e, the ring $R[X]/(f)$ is isomorphic to $\mathbb{F}_{p^{de}}$. As a consequence, the DLP for an e-th order recurrence over $\mathbb{F}_{p^d}$ is as hard as the DLP in $\mathbb{F}_{p^{ed}}$ (inverting L poses no problems). Index calculus methods apply, as well as Lenstra's observation that the order of the recurrence should be a divisor of $\Phi_{ed}(p)$. In particular this implies that the security of LUC is the same as that in $\mathbb{F}_{p^2}$ and the security of XTR as that in $\mathbb{F}_{p^6}$.

For the sake of completeness, we list some results for fast evaluation of the n-th value in a k-th order recurrence. Miller and Spencer Brown [**125**] show that it suffices to compute the n-th power of the $(k \times k)$-matrix M. This method runs in time logarithmic in n and was reinvented by Urbanek [**189**], who gives running time $O(k^3 \log n)$. Since matrix multiplication in general can be done in $O(k^{\lg 7})$ due to Strassen [**182**] and theoretically even faster [**50**], this is clearly not optimal. Gries and Levin [**73**] exploit the special form of the companion matrix to arrive at an $O(k^2 \log n)$ algorithm for evaluating x_n. Fiduccia [**63**] notes that it is more efficient to compute the n-th power of X modulo the characteristic polynomial, leading to an $O((k \log k) \log n)$ algorithm. (Fiduccia gives $O(k^{\lg 3} \log n)$ in general and only the faster $O((k \log k) \log n)$ if the recurrence is defined over a ring supporting a fast Fourier transform. With Kaminski's polynomial multiplication [**83**] the ring no longer needs to support a fast Fourier transform to allow an $O((k \log k) \log n)$ algorithm.)

1.4. Computational Model

The ultimate goal in speeding up a cryptosystem is simple: it should take users as little time as possible to do what they want or alternatively, they are able to do more in the same time. It would therefore be natural to measure the 'speed' of a cryptosystem in seconds. Measurements in seconds can be obtained by actually implementing a cryptosystem and running it on a computer. Using different computers, different operating systems or different compiler options can lead to different timings. Although a comparison between two cryptosystems can be made this way, there is a disturbing influence of outside factors on the timings acquired. A solution is to measure the number of operations (which we already did in some of the previous paragraphs). The question is what sort of operations to measure. The closest approximation to the actual runtime is the number of word operations. Unfortunately, wordsizes are a property of the architecture, which would introduce an unwanted outside influence in our timing.

Another approach to compare systems accurately is based on the algebraic nature of the schemes. In the 1970s algebraic complexity theory was developed [**143, 183, 184**]. Loosely speaking, an operation in some algebraic structure is measured in the number of basic operations. The actual runtime of a system is then obtained by assigning to each of the basic operations in the algebraic structure the amount of time it takes to perform this operation. Although algebraic complexity theory puts a strong emphasis on lower bounds, upper bounds and asymptotic results and we are more interested in optimizing specific instances of limited size.

An important choice to be made is pointing out the basic operations. Fortunately, the systems described in this thesis mostly use modular arithmetic. In other words, they are based on a common algebraic structure, namely some field $\mathbb{F}_p$. Measurements made in the number of operations in $\mathbb{F}_p$ allow a good comparison of systems (as long as they are based on roughly equally large primes p). Below we will give a brief overview of $\mathbb{F}_p$-arithmetic (our notation is based on Cohen and Lenstra [**47**]). We conclude with Assumption 1.3 in order to measure all operations in the number of $\mathbb{F}_p$-multiplications. This makes it easier to compare systems and it is a logical choice, since the dominating factor for the runtime of an algorithm is usually the number of modular multiplications and squarings.

The algorithms discussed in Chapters 2 and 3 for performing an exponentiation work for any group. The runtime is therefore first expressed in the number of group operations required. In Chapters 4 and 5 the group operations are described and the required number of $\mathbb{F}_p$-operations per group operation is given. Combination of the two types of measurement leads to the cost of an exponentiation for a specific subgroup cryptosystem in the number of $\mathbb{F}_p$-multiplications. In Section 5.4 the number of $\mathbb{F}_{2^l}$-multiplications is used as main measure and in Chapter 6 the group is not specified, so only the number of group operations is counted.

1.4.1. Modular Arithmetic. We consider multiplying two l-bit numbers modulo a third l-bit prime p. Although it is possible to perform the multiplication and the reduction interleaved, usually both are performed separately. We use M to denote the cost of multiplying two l-bit numbers (without modular reduction) and D for reducing a $2l$-bit number modulo an l-bit number. Since squaring a number tends to be cheaper than a multiplication, we use S for the cost of squaring an l-bit number (without modular reduction). The cost for performing a multiplicative inverse modulo an l-bit number is denoted by I (so a division costs $I+M+D$). We use A_1 for adding two l-bit numbers (including a reduction if needed), and A_2 for adding two $2l$-bit numbers (no reduction). A modular addition (of cost A_1) typically boils down to two or three plain l-bit additions (which makes it hard to determine whether $A_1 > A_2$ or vice versa). Consequently, the stated numbers of additions should be taken with a grain of salt. As another example, in Lemma 4.5 on page 74 the cost of subgroup squaring is approximated by $2S+2D+A_1$, assuming that the cost of subtracting one or multiplying by two is negligible compared to A_1 and A_2.

1.4.2. Multiplication. The schoolbook method for multiplication requires $O(l^2)$ bit operations. Karatsuba's often used method runs in $O(l^{\lg 3})$ bit operations. Asymptotically faster methods exist, but for the range of l we are interested in these methods are still more expensive. Knuth [**90**, Section 4.3] and Bernstein [**15**] give good overviews of fast multiplication methods.

An important factor, the platform on which the multiplication is performed, is hidden in the big O notation. As a small illustration, Brown et al. [**36**] describe an implementation of modular arithmetic in assembly language on a Pentium II 400 MHz PC using the classical method where a multiplication of 192 bit numbers takes about as long as 8 to 9 additions (of 192 bit numbers on the same machine).

Addition is performed in $O(l)$, so multiplication should be $O(l)$ slower. That Brown et al. report only a factor 8 to 9 (and not 192) is a result of using a 32-bit architecture (the factor 8 to 9 is reasonably close to 192/32).

If several multiplications are to be performed, having a common or fixed multiplicand can lead to a speedup. This is especially true in a different computational model, where the building block of l-bit (modular) multiplication is l-bit addition [**18, 41–43, 82**], which can be regarded as "exponentiation" in $(\mathbb{Z}_n, +)$. It is an aspect we do not take into account.

Squaring can usually be done slightly faster than a multiplication. This holds both in theory and in practice. If for some reason squaring happens to be more than twice as fast as a multiplication, the multiplications can often be replaced by two squarings, since $ab = ((a+b)^2 - (a-b)^2)/4$.

1.4.3. Reduction. Bosselaers et al. [**28**] compare three popular techniques to take care of the modular reduction. The schoolbook method of reducing a $2l$-bit number by a l-bit modulus is by means of a long division. A disadvantage of this classic method is its inability to reap profit from faster multiplication routines. In both Barret and Montgomery reduction the division is replaced by multiplications (and some overhead). Furthermore, the cost of the reduction is hardly affected if it is fed numbers slightly larger than $2l$-bits to reduce. Bosselaers et al. conclude that all three methods give similar performance for moduli of cryptographic interest, although their figures favour Montgomery-reduction by 10%.

Barrett reduction [**10**] is based on the observation that $z \mod n = z - n\lfloor \frac{z}{n} \rfloor$ and, assuming $z < 2^{2l}$, $\lfloor \frac{z}{n} \rfloor \approx \lfloor \lfloor \frac{z}{2^{l-1}} \rfloor \lfloor \frac{2^{2l}}{n} \rfloor 2^{1-l} \rfloor$. If $\lfloor \frac{2^{2l}}{n} \rfloor$ is precomputed, the reduction takes two (partial) multiplications plus some shifts and additions.

Montgomery [**129**] proposes to represent a number $x \mod n$ by $xR \mod n$, where R is a sufficiently large radix coprime to n. Multiplication of $xR \mod n$ and $yR \mod n$ gives a $2l$-bit number congruent to xyR^2 modulo n. The reduction operation mapping z to $zR^{-1} \mod n$ can be implemented based on the fact that $zR^{-1} \equiv (z + (zn' \mod R)n)/R \mod n$, where $n' = -n^{-1} \mod R$. By using a suitable R (typically related to the word size), the reduction can be implemented using only two (partial) multiplications plus some shifts and additions.

Reductions are significantly cheaper if the modulus has a 'nice' form. An example is a modulus of the form $2^l - 1$, requiring only a shift and an addition for the reduction of a $2l$-bit number. Several generalizations are known. Special moduli are used in elliptic curve cryptography (Chapter 5).

1.4.4. Inverses. Computing modular inverses is by far the most expensive basic operation. Usually some variant of the extended Euclidean algorithm is used. For bitlengths of cryptographic relevance Brown et al. [**36**] report a whopping 80 to 1 ratio for the costs $I : (M + D)$. The reduction here is in fact negligible since the modulus is 'nice'. A lower ratio of 23 is reported by De Win et al. [**198**] using a different inversion algorithm [**7**]. In this case, the reduction is real thereby lowering the ratio.

If several inversions have to be performed at the same time, Montgomery [**130**] shows it is possible to trade all but one of the inversions for three $\mathbb{F}_p$-multiplications each [**45**, Algorithm 10.3.4].

1.4.5. Simplification. In the literature often a simplified model is used, where an l-bit modular multiplication is the unit of measurement, a squaring costs 80% of a modular multiplication, and additions are considered negligible.

ASSUMPTION 1.3. *Let p be an l-bit prime. Let the runtime of an algorithm be given in the number of operations A_1, A_2, M, S, and D (using modulus p). If the total number of additions A_1 and A_2 is independent of l, then the runtime of the algorithm measured in the number of $\mathbb{F}_p$-multiplications can be realistically estimated by assuming that $A_1 = A_2 = 0$, $M = D = 0.5$, and $S = 0.3$ measured in the number of $\mathbb{F}_p$-multiplications.*

For exponentiations we always switch back to the simplified version, in order to facilitate comparisons with other results given in the literature. When inversions are needed, this will be explicitly mentioned.

1.4.6. Binary Fields. In Chapter 5 elliptic curves over $\mathbb{F}_{2^l}$ will be discussed. In this case A_1, D, M, S, and I refer to the respective $\mathbb{F}_{2^l}$-operations based on polynomial representation: addition l-bit addition "with reduction"; reduction of a $2l$-degree polynomial modulo an l-degree polynomial; multiplication of two l-degree polynomials (without reduction); squaring an l-degree polynomial (without reduction); and finally inverting an l-degree polynomial modulo another l-degree polynomial (including reduction).

Hankerson et al. [**74**] provide an overview and comparison of several algorithms to implement $\mathbb{F}_{2^l}$ arithmetic based on a polynomial basis. Elements of $\mathbb{F}_{2^l}$ are represented as polynomials over $\mathbb{F}_2$ of degree lower than l and computations are performed modulo an irreducible polynomial of degree l. The irreducible polynomial chosen is usually a trinomial or pentanomial, resulting in a reduction costing only a couple of additions (and some shifts). Additions themselves are merely (l-bit) XOR operations. In $\mathbb{F}_2[x]$ squaring is cheap since $(\sum_{i=0}^{l-1} b_i x^i)^2 = \sum_{i=0}^{l-1} b_i x^{2i}$, requiring only some shuffling (the reduction is counted separately).

Multiplication of two degree l-bit polynomials can take two different directions. Quite common is to base it on l-bit additions, giving rise to algorithms similar to those used for exponentiation. In this setting it pays off if several multiplications have a common multiplicand. The other method is more akin to the large integer arithmetic. Both polynomials are split in small parts and then several of the smaller polynomials are multiplied and the results combined (for instance using Karatsuba's technique).

Inversions in $\mathbb{F}_{2^l}^*$ can be performed using an extended Euclidean algorithm but alternatives exist. Measured in the number of $\mathbb{F}_{2^l}$-multiplications an inversion is said to cost between 3 multiplications by De Win et al. [**198**] and 10 multiplications by Hankerson et al. [**74**]. We will adhere to 10 multiplications to ease comparison with Hankerson et al. in Chapter 5.

2

Addition Chains

In this chapter several well known results and algorithms concerning addition chains are reviewed. Addition chains are used to efficiently compute an exponentiation or, more generally, several exponentiations. They are used for this purpose in Chapters 4, 5, and 6. The algorithms for higher order chains discussed in Chapter 3 are based on the algorithms to compute 'ordinary' addition chains. No new results are presented in this chapter, apart from possibly Lemma 2.29.

2.1. Introduction

In this chapter algorithms are described for speeding up exponentiations, i.e., the computation of g^n, by reducing the number of group operations. Related to the computation of g^n and of practical importance, is the efficient computation of $g^n h^m$, known as a double exponentiation. The most prominent application of double exponentiation in cryptography is signature verification in ElGamal-like signatures [**60, 75, 124**]. Also relevant in practice is the simultaneous computation of g^n and g^m, called a twofold exponentiation. The encryption scheme by Cramer and Shoup [**55**] is just one example of a cryptosystem that benefits from exploiting simultaneous exponentiations.

When implementing a cryptosystem, using Algorithm 1.1 to perform the exponentiations takes far too many steps (multiplications). More efficient methods are needed. But what is the most economical way to compute g^n?

Suppose we would keep track of all group elements that are successively computed by an algorithm that computes g^n (on input g and n). This list would begin with g and eventually end with g^n. Each new element will necessarily be formed by multiplying two elements already appearing on the list (what else is there to do?). Clearly, all intermediate values are powers of g. Since the exponents are additive, the problem can be reformulated in terms of addition chains. Addition chains (Additionsketten) were introduced by Scholz [**164**].

In Section 2.2.1 the relevant definitions of addition chains are given, incorporating among others multi-exponentiation and simultaneous exponentiation. Algorithms for multi-exponentiation and those for simultaneous exponentiation are strongly related. The underlying principle, called duality, is discussed in Section 2.2.2. Section 2.2.3 wraps up the theoretical part of this chapter, containing for instance Pippenger's bound on the asymptotic worst-case behaviour of addition chains for simultaneous multi-exponentiation.

Finding short addition chains is not the same as finding good exponentiation routines. In fact, although for each exponentiation routine there is a unique addition chain, the converse is not always true and the different exponentiation routines mapping to the same addition chain might not be equally efficient. In the theory of addition chains it is customary to count the number of multiplications without distinction. In practice, group squarings are often cheaper than group multiplications. A real-life situation might also put constraints on the memory available. We will even encounter situations where longer addition chains lead to faster exponentiation routines. In Section 2.1.1 some notation is introduced that corresponds better with reality.

There is a vast body of literature about finding short addition chains and fast exponentiation routines. Bernstein [**16**], Gordon [**71**], Knuth [**90**] and Menezes et al. [**121**] all provide overviews (and there are more). We will concentrate on a selection of algorithms that is most relevant to the remainder of this thesis. Hence the emphasis is on exponents of bitlengths up to about 200. We distinguish between two classes of algorithms: square-and-multiply algorithms and Euclidean algorithms.

The first category is the most popular and a large number of variations exist. Best known are the binary algorithms discussed in Section 2.3.1. Especially the left-to-right version (Algorithm 2.7) is easy to implement, relatively fast and it requires only little memory. In Section 2.3.2 a large number of more advanced square-and-multiply algorithms are discussed, including algorithms suitable for simultaneous or multi-exponentiation. Some of these algorithms exploit cheap inversions in the group.

The Euclidean algorithms are discussed in Section 2.3.3. In Section 2.3.4 a technique is discussed that exploits an endomorphism (other than inversion) and that is used in Chapter 4. We briefly consider the possibility of (off-line) precomputation for single exponentiation in Section 2.3.5. At the expense of memory, this can speed up an exponentiation considerably if the base is fixed.

We ignore the possibility to speed up an exponentiation by adding a multiple of the group order to the exponent [**44**]; we ignore susceptibility to side-channel attacks; we ignore possibilities to parallelize; we hardly consider space requirements; and finally, we largely ignore the time the control code of the algorithm takes.

2.1.1. Notation. Algorithm 1.1 takes $n-1$ group multiplications. When expressing the runtime of an exponentiation algorithm the cost of a single group multiplication will be denoted by α, so in the present case we have that the algorithm costs $(n-1)\alpha$. A more refined runtime analysis is possible, since the first group multiplication is actually a group squaring. The cost of a single group squaring will be denoted by δ since group squarings are often cheaper to perform than group multiplications. The cost of an inversion in the group is denoted by ν. The runtime of the algorithm can then be written down as $(n-2)\alpha + \delta$.

The choice of symbols α, δ, and ν is based on additive terminology in English and the "canonical" mapping from the Latin alphabet to the Greek one. The use of additive terminology is motivated on the one hand by the use of addition chains

TABLE 2.1. Cost examples measured in the number of $\mathbb{F}_p$-multiplications

Group	Cost of α	Cost of $\dot{\alpha}$	Cost of δ
Section 4.3, $\mathsf{G}_{p+1} \subseteq \mathbb{F}_{p^2}^*$	2.5	2.5	1.8
Section 4.7.1, $\mathbb{F}_{p^2}^*/\mathbb{F}_p^*$	2.5	2	2
Section 4.5, $\mathsf{G}_{p^2-p+1} \subseteq \mathbb{F}_{p^6}^*$	12	12	6
Section 5.5.1, mixed Chudnovsky Jacobian	6.4	6.4	3.2

for fast exponentiation routines (Chapter 2) and on the other a desire to reserve multiplicative terminology for the often underlying field arithmetic (Section 1.4).

Implicit in the notation are more or less identical runtimes for all multiplications in the group. For finite abelian groups, this is a realistic assumption. But for instance for the integers the bitlength of the operands influences the cost of a multiplication. (A suitable model for integer exponentiation is also known [**72, 119**].)

In some cases, such as Algorithm 2.7 (the left-to-right binary algorithm), one of the multiplicands will always be the same. The cost of these special multiplications, with one multiplicand fixed, will be denoted by $\dot{\alpha}$. For the algorithm above this would yield a runtime of $(n-2)\dot{\alpha}+\delta$. The separation of $\dot{\alpha}$ and α is especially useful if the exponentiation routine is going to be used for mixed coordinate elliptic curve cryptosystems: the exponentiation routine involves an affine part and a subsequent projective part. Even if the affine part consists of the unary algorithm only, the costs of the affine part will be counted using α instead of $\dot{\alpha}$. The projective part can now use $\dot{\alpha}$ when one of the affinely precomputed values is used.

This example demonstrates that the use of $\dot{\alpha}$ is useful, but also somewhat volatile. It is not invariant under duality (Section 2.2.2) and often tricky when considering precomputation (Section 2.3). Table 2.1 contains an overview of some examples based on modular arithmetic. The cost is measured in the number of $\mathbb{F}_p$-multiplications based on Assumption 1.3. The first two rows can be based on the same primes (of at least 512 bits long) and the two bottom rows can both be based on primes of at least 170 bits long.

2.2. Theoretical Aspects

2.2.1. Definitions. An algorithm to compute g^n will compute intermediate powers of g before finally arriving at g^n. The list of these intermediate powers is called an addition chain. The first element in this list is 1, which corresponds to g and the last element in this list is n, corresponding to the desired power g^n. More generally, an element c_i in the addition chain corresponds to the computation of g^{c_i} in the exponentiation algorithm. The elements c_i in an addition chain reflect the fact that g^{c_i} can only be computed by multiplying to known powers of g.

DEFINITION 2.1 (Traditional Addition Chain). An addition chain for a nonnegative integer n is an increasing sequence

$$C = \langle c_0, c_1, \ldots, c_k \rangle \ ,$$

with all c_i nonnegative integers, such that $c_0 = 1$ and $c_k = n$. Moreover, for all $0 < i \leq k$ there exist j and j', $0 \leq j \leq j' < i$, such that $c_i = c_j + c_{j'}$.

The length of C is defined to be $l(C) = k$. We denote by $l(n)$ the smallest integer l such that there exists an addition chain C for n with $l(C) = l$.

As a small example to demonstrate the correspondence between addition chains and exponentiations, consider the chain $\langle 1, 2, 4, 5, 6, 11\rangle$ of length 5. This chain proves that g^{11} can be computed using only 5 multiplications by consecutively computing $g^2, g^4 = (g^2)^2, g^5 = g^4 \cdot g, g^6 = g^4 \cdot g^2$, and finally $g^{11} = g^5 \cdot g^6$. Note that g^6 could also have been computed as $g^5 \cdot g$, showing that the mapping from exponentiation algorithms to addition chains is indeed many-to-one.

Computing a short addition chain for a single exponent naturally generalizes to computing a chain for multiple exponents. Consider for instance the simultaneous computation of both g^2 and g^4 It is clear that $g^n = g^2$ is a simple byproduct of the computation of $g^m = g^4 = (g^2)^2$. It follows that the simultaneous computation of two powers can be advantageous: the costs (number of multiplications) are lower than the total costs of performing both exponentiations separately. In the more general setting p exponents $n_1, \ldots, n_p$ are given and g^{n_1} up to g^{n_p} have to be computed simultaneously, corresponding to an addition chain that contains all n_i. Such an addition chain is commonly referred to as an addition sequence. For convenience, we write $g^{\mathbf{n}}$ for the simultaneous exponentiation, where $\mathbf{n}$ is the (row) vector $(n_1, \ldots, n_p)$. This problem was first posed by Knuth [**88**, Section 4.6.3, Exercise 22].

Another possibility is the computation of $g^n h^m$, known as a double exponentiation. The example $n = 2$ and $m = 4$ shows that multiplying the result of the single exponentiations g^n and h^m is not always the most efficient method: computing g^2h^4 as $(gh^2)^2$ takes only 3 multiplications whereas computing g^2, h^2, h^4 and then g^2h^4 takes 4. In the more general problem of multi-exponentiation a (row) vector $\mathbf{g} = (g_1, \ldots, g_m)$ of m group elements and a (column) vector $\mathbf{n} = (n_1, \ldots, n_m)^T$ of m exponents is given. The multi-exponentiation $\prod_{i=1}^{m} g_i^{n_i}$ is denoted $\mathbf{g}^{\mathbf{n}}$. Bellman [**12**] posed the question how to perform multi-exponentiation efficiently. In response, Straus [**185**] remarked that the problem of minimizing the number of multiplications to compute $\mathbf{g}^{\mathbf{n}}$ corresponds to finding a shortest addition chain for the column vector $\mathbf{n}$. Each base g_i corresponds to a basis (column) vector $\mathbf{e}_i$ (consisting of zeros and a single one at position i).

The combination of multi-exponentiation and simultaneous exponentiation leads in a natural way to simultaneous multi-exponentiation. Let $\mathbf{g}$ be a vector of m group elements an let $\mathbf{n}_1, \ldots, \mathbf{n}_p$ be vectors of m exponents each. Let N be the matrix having the vectors $\mathbf{n}_i$ as its columns. The simultaneous computation of $\mathbf{g}^{\mathbf{n}_1}$ up to $\mathbf{g}^{\mathbf{n}_p}$ is denoted by $\mathbf{g}^N$.

The following definition, taken from Knuth and Papadimitriou [**92**], captures both generalizations at the same time.

DEFINITION 2.2 (Addition Chain). Let $N = [n_{ij}]$ be an $m \times p$ matrix of nonnegative integers such that no row or column of N is entirely 0. An addition chain

for N is a sequence

$$C = \langle \mathbf{c}_{-m+1}, \mathbf{c}_{-m+2}, \ldots, \mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_k, \mathbf{c}_{k+1}, \ldots, \mathbf{c}_{k+p} \rangle$$

of $m \times 1$ vectors, such that:

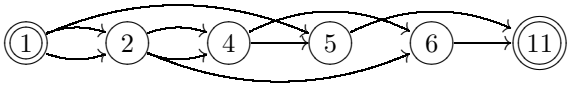
i. The vectors $\mathbf{c}_{-m+1}, \ldots, \mathbf{c}_0$ are the m possible unit $m \times 1$ vectors, in their natural order.

ii. For each $j, 1 \leq j \leq p$, the vector $\mathbf{c}_{k+j}$ equals the jth column of N.

iii. For each $i, 1 \leq i \leq k+p$, there is a (smallest) integer $r(i) \geq 1$, and a sequence of $r(i)$ integers $0 \leq j(i,1), \ldots, j(i,r(i)) \leq \min(i-1, k)$, such that $\mathbf{c}_i = \sum_{q=1}^{r(i)} \mathbf{c}_{j(i,q)}$.

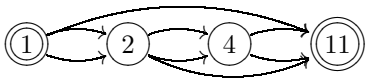
The length of C is defined to be $l(C) = \sum_{i=1}^{k+p}(r(i) - 1)$. We denote by $l(N)$ the smallest integer l such that there exists an addition chain C for N with $l(C) = l$.

Closely related to the definition above, is a graph-theoretic interpretation of addition chains (attributed to Pippenger [**149**] by Knuth and Papadimitriou [**92**], but Pippenger himself refers to Lupanov [**117**]). Given an addition chain C for N, a directed acyclic graph $G_C = (V_C, E_C)$ is defined by letting the vertices $V_C = \{v_{-m+1}, \ldots, v_{k+p}\}$ correspond to the vectors in C. The set of arcs E_C is given by $(v_j, v_i) \in E_C$ iff $j = j(i,q)$ for some $q \leq r(i)$. Note that multiple arcs are possible (e.g., for $c_1 = 2c_0$ if $m = 1$) and that given any chain, the graph is not necessarily unique. The sources of the graph, m in total, correspond to the bases of the exponentiation (or the vectors c_i with $i \leq 0$). The values to be computed, that is c_i with $i > k$, are represented by the p sinks of the graph. The point is that for each vertex the corresponding value in the addition chain can be computed by adding the 'vertices' at the other ends of the incoming arcs.

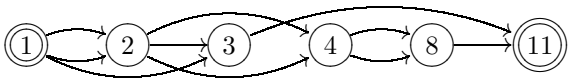


The figure above shows the graph for the chain $\langle 1, 2, 4, 5, 6, 11 \rangle$, which is an optimal chain for 11. All points (apart from the source) have indegree exactly 2, nicely adhering to Definition 2.1. The restriction that the indegree should be exactly 2 is not imposed by Definition 2.2. This allows a more compressed representation of an addition chain. Whenever a point has outdegree 1, the two points at both ends of the arc can be joined. The same holds when a point has indegree 1. (An exception should be made for sources and sinks if compression would result in changing their being a source respectively a sink.) This compression method identifies chains that are identical under the associativity and commutativity of addition (or, when used as an exponentiation routine, under the associativity and commutativity of the group operation).

In the example above, the points 5 and 6 have outdegree 1. Joining both with 11 leads to the condensed chain $\langle 1, 2, 4, 11 \rangle$. (Note that the vertex 11 is still a sink). The chain still has length 5, but this might not be immediately clear. The graph below corresponds to $\langle 1, 2, 4, 11 \rangle$. It is now easier to see that the length of the chain is indeed 5 by counting the total number of arcs minus the number of non-sources $(8 - 3 = 5)$.

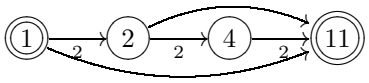


Surprisingly, compressing chains can also give rise to a more efficient algorithm to perform a specific exponentiation by trading multiplications for squarings or reducing the amount of memory required. As an example, the chain $\langle 1, 2, 4, 5, 6, 11\rangle$ can be computed for $2\delta + 2\dot{\alpha} + \alpha$. The chain $\langle 1, 2, 3, 4, 8, 11\rangle$ (depicted by the graph below) has the same compressed chain, but it can be computed for $3\delta + \dot{\alpha} + \alpha$. Depending on the relative costs of $\dot{\alpha}$ and δ, exponentiation based on one or the other chain will be cheaper.



Addition-Subtraction Chains. In some groups, such as those based on elliptic curves, inversion is relatively cheap. Allowing inversions during the exponentiation routine corresponds to taking the additive inverse of an element in the addition chain. Definition 2.1 can be adapted to allow subtractions by requiring that for each a_k a pair $i, j < k$ exists such that either $a_i + a_j = a_k$ or $a_i - a_j = a_k$. This relaxation gives rise to so-called addition-subtraction chains. The shortest addition-subtraction chain is evidently at most as long as the shortest addition chain.

Word Chains. Negation is a special case of a group endomorphism. Sometimes other endomorphisms are cheap to compute, for instance the Frobenius endomorphism in finite fields. Since the endomorphism group of a finite cyclic group of order x is isomorphic to $\mathbb{Z}_x$, the application of an endomorphism is equivalent to taking the appropriate power (in the case of Frobenius, p-th powering where p is the characteristic of the field). For addition chains this translates to (free) multiplication with a constant (p in the case of Frobenius). Von zur Gathen [**195**] defines a p-addition chain as an addition chain where, in the terminology of Definition 2.1, either $c_i = c_j + c_{j'}$ or $c_i = pc_j$. This definition has the undesired side-effect that application of the endomorphism increases the length of the chain, although this can easily be resolved. In the graph-theoretic representation, the application of an endomorphism corresponds to labelling an arc (with p for Frobenius, with -1 for an inversion). The value corresponding to a vertex can now be found by adding the vertices of the incoming arcs, after multiplication by the label belonging to that arc. As an example, we could redraw the graph for $\langle 1, 2, 4, 11\rangle$ in order to make the squarings explicit.



Von zur Gathen also introduces word chains. Initially some finite alphabet of cardinality p is given —these are the one-letter words. In each step two words already in the chain may be concatenated. The link with the p-addition chains is readily made.

2.2.2. Duality. Suppose we have an efficient algorithm to simultaneously compute g^n and g^m. What does this buy us when we want to compute $g^n h^m$? Olivos [**142**] studied this problem for general $\mathbf{n}$, tightly linking addition sequences and vector addition chains. The following theorem works for any combination of simultaneous multi-exponentiation. It is due to Knuth and Papadimitriou [**92**], although Pippenger [**149**] already seems to exploit duality.

THEOREM 2.3 (Knuth and Papadimitriou). *Let N be as in Definition 2.2. Then $l(N) - m = l(N^T) - p$.*

Proof: The gist of the proof is as follows. Given an addition chain C for N, consider its graph-theoretical representation. Knuth and Papadimitriou observe that for any $1 \le i \le m$ and $0 \le j \le k + p$ the number of distinct paths from v_{-m+i} to v_j equals the ith component of the vector c_j. Conversely, any directed acyclic graph G with m sources and p sinks gives rise to some addition chain for an $m \times p$ matrix. Reversing the directions of all arcs results in an addition chain for N^T. Moreover, the length $C(G)$ is the sum of the indegree, minus one, over all nodes except the sources. *Q.E.D.*

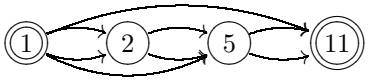
As a concrete example we consider the dual of Algorithm 1.1 (the version to compute g^i for all $0 < i \le n$ and not only g^n). The dual algorithm will compute $\mathbf{g^n} = \prod_{i=1}^{n} g_i^i$, where g_i are arbitrary group elements and $\mathbf{n} = (1, \ldots, n)^T$. The algorithm is also described by Knuth [**90**] and Brickell et al. [**32**].

ALGORITHM 2.4 (Unary Multi-Exponentiation).
On input $\mathbf{g}$ and $\mathbf{n} = (1, \ldots, n)^T$, the algorithm returns $\mathbf{g^n}$. It maintains as invariant:

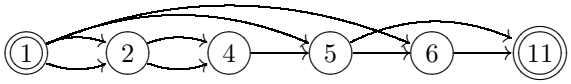
$$0 < j \le n, \quad A = \prod_{i=j}^{n} g_i^{i-j+1}, \quad B = \prod_{i=j}^{n} g_i \ .$$

1. [Initialize] Set $A \leftarrow g_n, B \leftarrow g_n$, and $j \leftarrow n$.
2. [Finished?] If $j = 1$ terminate with output A.
3. [Decrease j] Set $B \leftarrow g_{j-1}B, A \leftarrow AB$, and decrease j by one. Return to the previous step.

Knuth and Papadimitriou show that the concept of duality is already interesting for single exponentiation. For instance, reversing the arcs of the graph belonging to $\langle 1, 2, 4, 11 \rangle$ leads to the graph below for the addition chain $\langle 1, 2, 5, 11 \rangle$.



An expanded version of this addition chain is $\langle 1, 2, 4, 5, 6, 11\rangle$, as shown by the graph below. This is somewhat curious, since $\langle 1, 2, 4, 5, 6, 11\rangle$ can also compress to $\langle 1, 2, 4, 11\rangle$.



The duality of addition chains is closely related to the duality of matrix multiplication: given an $m \times p$ matrix N (over a ring R) and column vectors $\mathbf{g}$ of length p and $\mathbf{g}'$ of length m, consider computing $N\mathbf{g}$ and $N^T\mathbf{g}'$ using only ring additions and ring multiplications by constants (i.e., the linear complexity). Ignoring the ring multiplications, matrix multiplication can be regarded as an exponentiation over an additive group. Kaminski et al. [**84**] consider the duality aspects of matrix multiplication. Interestingly, instead of defining linear complexity by means of straight line programs, Kaminski et al. employ an equivalent, graph-theoretic description closely related to the one used for addition chains. Indeed, the additions are modelled identically and by labelling each arc by a nonzero ring element the multiplications enter into the model. If a graph computes $N\mathbf{g}$, then reversing the arcs (but without changing the labelling) will produce a graph that computes $N\mathbf{g}'$ (on appropriate inputs). Clearly the number of multiplications by any specific ring element is exactly the same for both N and N^T (Fiduccia [**62**] already concluded that the total number of multiplications required is identical, but he does not take into account the number of additions). Translated to addition chains, this means that for instance the optimal way to compute $\mathbf{g}^{N^T}$ requires exactly the same amount of squarings as the optimal computation of $\mathbf{g}^N$. And the same holds for the number of inversions, Frobenius endomorphisms, etc.

Bernstein [**16**] provides more thoughts on the duality of addition chains and its relation with matrix multiplication.

2.2.3. Lower Bounds and Asymptotic Behaviour. Knuth [**90**, Section 4.6.3] gives a fascinating account of the search for $l(n)$ for general n and the theoretical questions that arise from it. Relatively easy to prove is the observation that $l(nm) \leq l(n) + l(m)$. Based on an exhaustive search for $n < 2^{20}$, Bleichenbacher and Flammenkamp [**21**] analyse the occurrence of $l(nm) < l(n) + l(m)$, which is surprisingly often if nm is odd. In the more general setting $l(NM) \leq l(N)+l(M)$, where N and M should have compatible dimensions. A big difference with the scalar version, is that integers factor uniquely, whereas matrices do not decompose uniquely. Some decompositions are more natural than others (Section 6.4.1 contains an example). Most exponentiation algorithms can be rephrased in some sort of decomposition.

The asymptotic behaviour of $l(N)$ for general N has been studied by Pippenger [**149**–**151**], who gives the following theorem.

THEOREM 2.5 (Pippenger). *Denote by $L(m, p, B)$ the maximum of $l(N)$ over all $m \times p$ matrices N with entries $\leq B$. If $p = (B + 1)^{o(m)}$ and $m = (B + 1)^{o(p)}$,*

then

$$L(m,p,B) = \min(m,p)\lg B + H/\lg H + \\ + O(H\sqrt{(\lg\lg H)/(\lg H)^3}) + O(\max(m,p))$$

where $H = mp\lg(B+1)$.

The simple bound $\lfloor \lg n \rfloor \le l(n) \le 2\lfloor \lg n \rfloor$ implies that the problem of determining whether an integer n admits an addition chain of length smaller than a given l is in NP: the set of integers in the chain would form a witness of bitlength at most $2(\lg n)^2$. A shorter witness is possible by encoding for all elements i in the chain the indices j and j' (from Definition 2.1). This witness has bitlength at most $2\lg n \lg(2\lg n)$.

Downey et al. [**59**] show that a graph with m edges and n vertices $\{u_i, v_i\}, 0 < i \le m$ has a minimum vertex cover of cardinality k iff $l(1, 2, \ldots, 2^{An}, 2^{Au_1} + 2^{Av_1} + 1, \ldots, 2^{Au_m} + 2^{Av_m} + 1) = An + m + k$ for all $A \ge 9m^2$. Since the minimum vertex cover is NP-complete, so is the shortest addition sequence problem. Although technically this does not imply that finding shortest addition chains for integers (or fixed size N) is NP-hard, it certainly does not bode well.

The shortest addition chain for n can be found by checking all certificates (witnesses), taking time $O(n^{c\log\log n})$ for some constant $c > 0$. This is certainly not polynomial in its input size $\lfloor \lg n \rfloor$; it is not even polynomial in n. However, by clever searching and good pruning bounds to cut down the tree, Bleichenbacher and Flammenkamp [**21**] determined $l(n)$ for all $n \le 2^{20}$. Using a different approach (and more time), Thurber [**186**] found all shortest addition chains for a given n, for a large number of small values of n.

2.3. Exponentiation Algorithms

2.3.1. The Binary Algorithms. The unary algorithm (Algorithm 1.1) runs in time $\dot\alpha(n-2) + \delta$ returning all g^i for $0 \le i \le n$. Since these are $n+1$ values only two of which come for free (namely g^0 and g^1), the addition chain for the list $(1, \ldots, n)$ is minimal. In a group where $\delta < \dot\alpha$, that is where squaring is cheaper than multiplication by a fixed multiplicand, the resulting exponentiation routine is not the fastest though. All even powers can be computed with a squaring instead of a multiplication. This leads to the algorithm below. (Note that if only g^n is needed, this algorithm requires much more storage than Algorithm 1.1).

Algorithm 2.6 (Improved Unary Exponentiation).
Let g be a group element and n a nonnegative integer. This algorithm maintains $0 \le a \le n$ and $G_i = g^i$ for all $0 \le i \le a$ as invariant. It returns g^i for all $0 \le i \le n$.

1. [Initialization] Set $a \leftarrow 0$ and $G_0 \leftarrow id$.
2. [Finished?] If $a = n$ terminate with output $(G_0, \ldots, G_n)$.
3. [Increase a] If a is even, set $G_{a+1} \leftarrow G_a g$ else ($a+1$ is even) set $G_{a+1} \leftarrow G^2_{(a+1)/2}$. Increase a by one and return to the previous step.

The runtime of this algorithm is $\lceil \frac{n-1}{2} \rceil \dot{\alpha} + \lceil \frac{n}{2} \rceil \delta$. Often the algorithm is used for precomputation in which case the fixed base cannot be (fully) exploited, since the output would then require postprocessing to speed up the further use of the precomputed G_i in subsequent multiplications.

If only g^n is needed, a large number of G_i are computed that are not subsequently used for the computation of g^n. By removing those excess computations a leaner algorithm emerges, better known as the left-to-right binary algorithm. The algorithm is quite often also referred to as the square-and-multiply algorithm. Even when the inevitability for exponentiation routines to depend on squarings and multiplications is discarded, this term is misleading since a large family of algorithms have a similar square-and-multiply structure, as discussed below.

ALGORITHM 2.7 (Left-to-Right Binary Exponentiation).
Let g be a group element and n some exponent with binary expansion $\sum_{i=0}^{k-1} b_i 2^i$ where all $b_i \in \{0, 1\}$. The algorithm returns g^n. It maintains as invariant

$$0 \leq j \leq k, \qquad a = \sum_{i=j}^{k-1} b_i 2^{i-j}, \qquad A = g^a \ .$$

1. [Initialization] Set $j \leftarrow k, a \leftarrow 0$, and $A \leftarrow id$.
2. [Finished?] If $j = 0$ terminate with output A.
3. [Square] (If $j < k$) Set $a \leftarrow 2a$ and $A \leftarrow A^2$.
4. [Multiply] If $b_{j-1} = 1$ set $a \leftarrow a + 1$ and $A \leftarrow Ag$.
5. [Decrease j] Decrease j by one. Go back to Step 2.

If the exponent is chosen uniformly at random from the interval $[2^{k-1}, 2^k)$, half of the b_i's in the binary expansion can be expected to be nonzero (for $0 \leq i < k-1$). Taking into account that in Step 4 one of the multiplicands is always the same, Algorithm 2.7 takes on average $(k-1)\delta + (\frac{k-1}{2})\dot{\alpha}$.

The algorithm is related to Horner's rule. If $n(x)$ is defined as the polynomial $\sum_{i=0}^{k-1} b_i x^i$, then $n = n(2)$. The evaluation in $x = 2$ using Horner's rule would perform the same steps as the algorithm above, with the only difference that A is not needed and a is output instead.

EXAMPLE 2.8. Let $n = 367$. The binary expansion of 367 is $(101101111)_2$, which has length $k = 9$. Algorithm 2.7 therefore initializes with $j \leftarrow 9, a \leftarrow 0$, and $A \leftarrow id$. We can skip Step 3 since $j = k$; performing it nevertheless would not change anything, for $a = 2a = 0$ and $A = A^2 = id$. In Step 4 we see that $b_8 = 1$, so we have to set $a \leftarrow 1$ and $A \leftarrow g$. Note that these substitutions do not require any arithmetic operations. In Step 5 j is decreased by one and we return to Step 2. Performing Steps 2 up to 5 once is called a round. Each round is labelled using the value of j at the beginning of the round. For $n = 367$ we start with round 9 and round 1 is the last full round (round 0 terminates in Step 2). In Table 2.2, the values of the important variables of the algorithm are given at the end of Step 4 for each round. (In future examples we will only mention the elements of the addition chain, corresponding to a here, and take the corresponding group elements for granted. In

practice, only the group elements are important and a can be ignored.) The column (Ini) stands for the situation just after the initialization (Step 1). The cost of each round is also included in the table; adding up the values in this row give a total cost of $8\delta + 6\dot{\alpha}$.

The dual of the left-to-right algorithm is the right-to-left algorithm. Conceptually all powers g^{2^i} are computed for $i < k$. The powers for which b_i equals 1, are needed: these are all multiplied together. By reading the bit representation of the exponent from right to left, it is not needed to store all powers g^{2^i} simultaneously. Nevertheless, the right-to-left binary algorithm needs more memory than its left-to-right counterpart. It is also potentially slower, since the group multiplications no longer have a common multiplicand. The runtime is $((\frac{k-1}{2})\alpha + (k-1)\delta$ on average.

ALGORITHM 2.9 (Right-to-Left Binary Exponentiation).
Given a group element g and an exponent $n = \sum_{i=0}^{k-1} b_i 2^i$, this algorithm computes g^n. It maintains the following invariant:

$$0 \le j \le k, \quad a = \sum_{i=0}^{j-1} b_i 2^i, \quad b = 2^j, \quad A = g^a, \quad B = g^b \ .$$

1. [Initialization] Set $j \leftarrow 0, a \leftarrow 0, b \leftarrow 1$ and $A \leftarrow id, B \leftarrow g$.
2. [Finished?] If $j = k$ terminate with output A.
3. [Multiply] If $b_j = 1$ set $a \leftarrow a + b$ and $A \leftarrow AB$.
4. [Square] (If $j < k - 1$) Set $b \leftarrow 2b$ and $B = B^2$.
5. [Increase j] Increase j by one and return to Step 2.

EXAMPLE 2.10. Let $n = 367$ as above. Algorithm 2.9 initializes with $j \leftarrow 0, a \leftarrow 0$, and $b \leftarrow 1$ (ignoring the group elements A and B). In Table 2.3, the values of a and b at the end of Step 4 for each round are given. The cost of each round is also included in the table; adding up the values in this row gives a total cost of $6\alpha + 8\delta$ for the algorithm. If $\alpha > \dot{\alpha}$, this is slower than the left-to-right algorithm applied to $n = 367$.

2.3.2. Generic Square-and-Multiply Algorithms. Several generalizations of the binary algorithms are known that still have a distinctive square-and-multiply structure. The main difference is their increased freedom in what is multiplied in and the ability to handle multi-exponentiation.

TABLE 2.2. The Left-to-Right Binary algorithm used for $n = 367$

j	(Ini)	9	8	7	6	5	4	3	2	1
b_{j-1}	$-$	1	0	1	1	0	1	1	1	1
a	0	1	2	5	11	22	45	91	183	367
A	id	g	g^2	g^5	g^{11}	g^{22}	g^{45}	g^{91}	g^{183}	g^{367}
Cost	0	0	δ	$\dot{\alpha}+\delta$	$\dot{\alpha}+\delta$	δ	$\dot{\alpha}+\delta$	$\dot{\alpha}+\delta$	$\dot{\alpha}+\delta$	$\dot{\alpha}+\delta$

In all single exponentiation routines the first element to be computed is g^2. This raises the question whether it is possible to use g^2 as well in Step 4 of Algorithm 2.7. If the exponent is (re)written as $n = \sum_{i=0}^{k'-1}(b_{i,1} + 2b_{i,2})2^i$ where both $b_{i,1}$ and $b_{i,2}$ are either 0 or 1, Step 4 can be replaced by:

4'. [Multiply] If $b_{j-1,1} = 1$ set $a \leftarrow a+1$ and $A \leftarrow Ag$. If $b_{j-1,2} = 1$ set $a \leftarrow a+2$ and $A \leftarrow Ag^2$.

It is possible that both $b_{i,1} = 1$ and $b_{i,2} = 1$ for the same i, resulting in two multiplications during Step 4 for this i. The length of the expansion, k', can deviate slightly from the bitlength of n. It might seem for a moment that for this reason the "new" method saves a squaring relative to the ordinary binary method by using $k' = k - 2$ and $b_{k-2,2} = 1$. This however is deceiving. The first time we need to set $A \leftarrow Ag^2$ in Step 4, the element g^2 has not been computed yet. It needs to be precomputed beforehand. The resulting algorithm costs exactly the same as Algorithm 2.7 (in terms of group operations).

Much ado about nothing, then? Not really. The representation of $n > 1$ using both $b_{i,1}$ and $b_{i,2}$ is certainly not unique. Suppose we somehow pick one that maximizes the number of i for which both $b_{i,1} = 1$ and $b_{i,2} = 1$. For all these i we have to perform two multiplications in Step 4, first by g and then by g^2. If g^3 would also be precomputed, we can replace the two multiplications by only one, leading to a reduction of the runtime.

EXAMPLE 2.11. Let $n = 367$. We can write this exponent as

$$367 = 2 \cdot 2^7 + 1 \cdot 2^6 + 2 \cdot 2^4 + (1+2) \cdot 2^2 + (1+2) \cdot 2^0 \ .$$

and base the modified Algorithm 2.7 on the resulting $b_{j,1}$ and b_{j_2}. If no more modifications are made, the total costs are $8\delta + 6\dot{\alpha}$, as shown by the first cost row. (Here we can actually see one of the inadequacies of our notation, since the δ in the precomputation, or initialization, might have different costs attached to it than the other δ.) The second cost row captures the situation where g^3 is precomputed as well. The total costs are $\alpha + 8\delta + 4\dot{\alpha}$.

This leads us to the general case. Let $\mathcal{D}$ be some set of integers, called the dictionary (if inversion is cheap, $\mathcal{D}$ can contain negative numbers). First all g^d are precomputed with $d \in \mathcal{D}$. Then the exponent is rewritten as $n = \sum_{i=0}^{k'-1} b_{i,d} d 2^i$ where $b_{i,d} \in \{0, 1\}$ for all $0 \le i < k'$ and $d \in \mathcal{D}$. This is the encoding. The left-to-right (no-longer-binary) algorithm is run on this encoding of the exponent.

TABLE 2.3. The Right-to-Left Binary algorithm used for $n = 367$

j	(Ini)	0	1	2	3	4	5	6	7	8
b_j	–	1	1	1	1	0	1	1	0	1
a	0	1	3	7	15	15	47	111	111	367
b	1	2	4	8	16	32	64	128	256	256
Cost	0	δ	$\alpha+\delta$	$\alpha+\delta$	$\alpha+\delta$	δ	$\alpha+\delta$	$\alpha+\delta$	δ	α

TABLE 2.4. The quaternary method used for $n = 367$

j	(Ini)	7	6	5	4	3	2	1	0
$b_{j,1}$		0	1	0	0	0	1	0	1
$b_{j,2}$		1	0	0	1	0	1	0	1
a		2	5	10	22	44	91	182	367
Cost	δ	0	$\dot{\alpha}+\delta$	δ	$\dot{\alpha}+\delta$	δ	$2\dot{\alpha}+\delta$	δ	$2\dot{\alpha}+\delta$
Cost$'$	$\alpha+\delta$	0	$\dot{\alpha}+\delta$	δ	$\dot{\alpha}+\delta$	δ	$\dot{\alpha}+\delta$	δ	$\dot{\alpha}+\delta$

ALGORITHM 2.12 (Generic Left-to-Right Exponentiation).
Let $\mathbf{g}$ be a vector of m group elements and let $\mathbf{n}$ be a vector of m exponents. The algorithm returns $\mathbf{g}^{\mathbf{n}}$. Let $\mathcal{D}$ be some predetermined set of (column) vectors of length m (that span $\mathbb{Z}^m$) and assume that the exponent can be written as $\mathbf{n} = \sum_{i=0}^{k'-1}(\sum_{\mathbf{d}\in\mathcal{D}} b_{i,\mathbf{d}}\mathbf{d})2^i$ where $b_{i,\mathbf{d}} \in \{0,1\}$. The algorithm maintains as invariant (from Step 3 onward):

$$0 \le j \le k', \quad \mathbf{a} = \sum_{i=j}^{k'-1}(\sum_{\mathbf{d}\in\mathcal{D}} b_{i,\mathbf{d}}\mathbf{d})2^{i-j}, \quad A = \mathbf{g}^{\mathbf{a}} \ .$$

1. [Exponent Recoding] Write $\mathbf{n} = \sum_{i=0}^{k'-1}(\sum_{\mathbf{d}\in\mathcal{D}} b_{i,\mathbf{d}}\mathbf{d})2^i$ where all $b_{i,\mathbf{d}} \in \{0,1\}$.
2. [Precomputation] Compute $G_{\mathbf{d}} = \mathbf{g}^{\mathbf{d}}$ for all $\mathbf{d} \in \mathcal{D}$ for which $b_{i,\mathbf{d}} = 1$ for some $0 \le i < k'$.
3. [Initialization] Set $j \leftarrow k', \mathbf{a} \leftarrow 0$ and $A \leftarrow id$.
4. [Finished?] If $j = 0$ terminate with output A.
5. [Square] (If $j < k'$) Set $\mathbf{a} \leftarrow 2\mathbf{a}$ and $A \leftarrow A^2$.
6. [Multiply] For all $\mathbf{d} \in \mathcal{D}$ with $b_{j,\mathbf{d}} = 1$ consecutively compute $\mathbf{a} \leftarrow \mathbf{a} + \mathbf{d}$ and $A \leftarrow A \cdot G_{\mathbf{d}}$.
7. [Decrease j] Set $j \leftarrow j - 1$ and go back to Step 4.

The algorithm above incorporates a generalization to multi-exponentiation. Let us forget this for a moment and assume $m = 1$. Given a dictionary, an exponent is written as $n = \sum_{i=0}^{k'-1} b_{i,d}d2^i$. The number of nonzero $b_{i,d}$ is called the *weight* of the representation. Any exponent may have several representations valid for a given dictionary, possibly with different weights and different lengths. A description of how to perform the encoding should be given alongside $\mathcal{D}$.

Once the encoding is fixed, the precomputation requires the computation of all g^d for $d \in \mathcal{D}$. This is a simultaneous exponentiation. For most dictionaries an efficient implementation is obvious, for others the precomputation can be involved (and subject to optimization). By doing the precomputation after the encoding, the values in the dictionary that are not used in the encoding can be ignored, possibly saving some multiplications.

After encoding and precomputation Steps 3 to 7 can be regarded as a multi-exponentiation. Define $n_d = \sum_{i=0}^{k'-1} b_{i,d}2^i$ for all $d \in \mathcal{D}$ and let $G_d = g^d$ as in the algorithm, then $n = \sum_{d\in\mathcal{D}} n_d d$ and $g^n = \prod_{d\in\mathcal{D}} G_d^{n_d}$. This multi-exponentiation

is computed using a binary algorithm, reading all the exponents n_d simultaneously and interleaving the squarings. In fact, it does exactly what the entire algorithm would do for a multi-exponentiation based on the trivial dictionary consisting of the unit vectors only.

The situation for a multi-exponentiation, i.e., $m > 0$, is similar. The precomputation of all elements $G_{\mathbf{d}} = \mathbf{g}^{\mathbf{d}}$ is a simultaneous multi-exponentiation, but Steps 3 to 7 are still a single multi-exponentiation, where $n_{\mathbf{d}} = \sum_{i=0}^{k'-1} b_{i,\mathbf{d}} 2^i$ and $\mathbf{g}^{\mathbf{n}} = \prod_{\mathbf{d}\in\mathcal{D}} G_{\mathbf{d}}^{n_{\mathbf{d}}}$.

EXAMPLE 2.13. Example 2.11 encompassed two cases: one with the precomputation of g^3 and one without. Let us consider the latter first. The precomputed elements g and g^2 correspond to the dictionary $\mathcal{D} = \{1, 2\}$. The exponent n is rewritten as $367 = 1 \cdot (2^6 + 2^2 + 2^0) + 2 \cdot (2^7 + 2^4 + 2^2 + 2^0)$; in other words, $n_1 = 69$ and $n_2 = 149$. If g^3 is precomputed as well, the dictionary is effectively $\mathcal{D} = \{1, 2, 3\}$ and n is rewritten as $n = 1 \cdot 2^6 + 2 \cdot (2^7 + 2^4) + 3 \cdot (2^2 + 2^0)$, or $n_1 = 64, n_2 = 144$, and $n_3 = 5$.

A different interpretation is the following. First the dictionary $\mathcal{D} = \{1, 2\}$ is used to transform the single exponentiation with $n = 367$ into a double exponentiation with $\mathbf{n} = \binom{69}{149}$. This double exponentiation is subsequently executed using $\mathcal{D}' = \{\binom{1}{0}, \binom{0}{1}, \binom{1}{1}\}$ (in terms of the single exponentiation, the vector $\binom{1}{1}$ corresponds to g^3). This second dictionary is used to rewrite the exponent $\mathbf{n}$ as $\binom{69}{149} = 64\binom{1}{0} + 144\binom{0}{1} + 5\binom{1}{1}$.

This two stage description is more natural if for instance the dictionary is $\mathcal{D} = \{1, 16\}$. The exponent can be rewritten as $367 = 22 \cdot 16 + 15$, corresponding to a double exponentiation $\binom{15}{22} = \binom{(01111)_2}{(10110)_2}$. Using $\mathcal{D}' = \{\binom{1}{0}, \binom{0}{1}, \binom{1}{1}\}$ is beneficial here as well. The vector $\binom{1}{1}$ corresponds to g^{17}, but using $\mathcal{D} = \{1, 16, 17\}$ immediately seems less clear.

The dual algorithm below performs the simultaneous exponentiation $g^{\mathbf{n}}$ by reversing the order. In the first stage the exponents are read from right-to-left to compute $\tilde{G}_{\mathbf{d}} = g^{n_{\mathbf{d}}}$ for all $\mathbf{d} \in \mathcal{D}$, which is a simultaneous exponentiation. In the second stage $g^{\mathbf{n}}$ is computed as $g^{n_j} = \prod_{\mathbf{d}\in\mathcal{D}} \tilde{G}_{\mathbf{d}}^{d_j}$ for $j = 1, \ldots, p$. This stage is the dual of the precomputation of Algorithm 2.12 and, as such is a simultaneous multi-exponentiation.

The cost of the left-to-right algorithm is that of the precomputation plus a multiplication for each nonzero $b_{i,\mathbf{d}}$ and $k' - 1$ squarings for the processing of the exponent. (In Step 6 initially $a = 0$ and $A = id$, so no real multiplication is needed for $A \leftarrow A \cdot G_{\mathbf{d}}$, 'saving' a multiplication). More precise statements of the costs follow when the dictionaries are being discussed. Conform Theorem 2.3, the dual algorithm is slightly cheaper.

ALGORITHM 2.14 (Generic Right-to-Left Exponentiation).
Let g be a group element and let $\mathbf{n}$ be a vector of p exponents. The algorithm returns $g^{\mathbf{n}}$. Let $\mathcal{D}$ be some predetermined set of (column) vectors of length p (that span $\mathbb{Z}^p$) and assume that the exponent can be written as $\mathbf{n} = \sum_{i=0}^{k'-1} (\sum_{\mathbf{d}\in\mathcal{D}} b_{i,\mathbf{d}} \mathbf{d}) 2^i$ where

$b_{i,\mathbf{d}} \in \{0,1\}$. The algorithm maintains as invariant (Steps 2–5):

$$0 \le j \le k', \quad b = 2^j, \quad B = g^b, \quad a_{\mathbf{d}} = \sum_{i=0}^{j-1}(\sum_{\mathbf{d}\in\mathcal{D}} b_{i,\mathbf{d}})2^i, \quad A_{\mathbf{d}} = g^{a_{\mathbf{d}}} \ .$$

1. [Exponent Recoding] Write $\mathbf{n} = \sum_{i=0}^{k'}(\sum_{\mathbf{d}\in\mathcal{D}} b_{i,\mathbf{d}}\mathbf{d})2^i$ where all $b_{i,\mathbf{d}} \in \{0,1\}$.
2. [Initialization] Set $j \leftarrow 0, b \leftarrow 1$, and $B \leftarrow g$. For all $\mathbf{d} \in \mathcal{D}$ set $a_{\mathbf{d}} \leftarrow 0$ and $A_{\mathbf{d}} \leftarrow id$.
3. [Multiply] For all $\mathbf{d} \in \mathcal{D}$ with $b_{j,\mathbf{d}} = 1$ set $a_{\mathbf{d}} \leftarrow a_{\mathbf{d}} + b$ and $A_{\mathbf{d}} \leftarrow A_{\mathbf{d}}B$.
4. [Square] (If $j < k' - 1$) Set $b \leftarrow 2b$ and $B \leftarrow B^2$.
5. [Increase j] Set $j \leftarrow j + 1$.
6. [Almost finished?] If $j < k'$ return to Step 3.
7. [Collecting] Compute $g^{n_j} = \prod_{\mathbf{d}\in\mathcal{D}} \tilde{G}_{\mathbf{d}}^{d_j}$ for $j = 1, \ldots, p$ using the dual of the precomputation that comes with the dictionary.

Dictionaries for Single Exponentiation. If $\mathcal{D} = \{1\}$, the exponent recoding is unique, being the binary representation of n, and precomputation is trivial, since nothing needs to be precomputed. As a whole, the resulting algorithm corresponds to the left-to-right binary algorithm above.

If inversion, i.e., the computation of g^{-1}, is cheap then single exponentiation can be sped up by using a signed digit representation for the exponent corresponding to $\mathcal{D} = \{-1, 1\}$. The representation is no longer unique, not even for minimal weight. The non-adjacent form (NAF) is the most common minimal weight representation [**132, 191**]. It derives its name from the property that there are no adjacent nonzeros (equivalently, $(\sum_{d\in\mathcal{D}} b_{i,d})(\sum_{d\in\mathcal{D}} b_{i+1,d}) = 0$ for all $0 \le i < k' - 1$). The NAF is the unique representation with this property and it provably has minimal weight. On average, the weight is $k/3$ and the length k' is at most one more than k. A disadvantage of the NAF is that computing it from the binary representation is a right-to-left operation, making interleaving the exponent encoding and the actual (left-to-right) exponentiation troublesome. Joye and Yen [**80**] describe an alternative encoding called the star form which has the same weight as the NAF, but that can be obtained from left-to-right. The average runtime of the resulting algorithms is $k\delta + \frac{1}{3}k\dot{\alpha}$.

Example 2.15. We only discuss the NAF for $n = 367$. The binary representation of 367 is $(101101111)_2$. To obtain the NAF of 367, we start reading the binary expansion from the right until we hit a 0, so we read $(01111)_2 = 15$. There are four adjacent nonzeros here, which is not allowed. By replacing 15 by 16-1 this problem can be solved. As customary in the literature, we write $\bar{1}$ for a -1 (or, more precisely, a 1 means that for that position $b_{i,1} = 1$ and $b_{i,-1} = 0$, a $\bar{1}$ means that $b_{i,1} = 0$ and $b_{i,-1} = 1$, and a 0 means that $b_{i,1} = b_{i,-1} = 0$). Replacing 15 by 16-1 means replacing $(01111)_2$ by $(1000\bar{1})_2$, leading to overall exponent $(101110001\bar{1})_2$. We continue reading until we hit a 0 again. This time we read $(0111)_2$ —where the least significant 1 is the one we just inserted— which we replace by $(100\bar{1})_2$. The exponent is now $(1100\bar{1}000\bar{1})_2$. There are still two adjacent

b

TABLE 2.5. The NAF used for $n = 367$

j	(Ini)	10	9	8	7	6	5	4	3	2	1
b_{j-1}	$-$	1	0	-1	0	0	-1	0	0	0	-1
a	0	1	2	3	6	12	23	46	92	184	367
Cost	0	0	δ	$\dot\alpha+\delta$	δ	δ	$\dot\alpha+\delta$	δ	δ	δ	$\dot\alpha+\delta$

1's. By thinking a zero in front of them, we can replace $(011)_2$ by $(10\bar1)_2$ giving $367 = 512 - 128 - 16 - 1 = (10\bar100\bar1000\bar1)_2$. Table 2.5 gives a brief overview of the important variables of Algorithm 2.12 based on this representation of the exponent. The length of the new representation is one bit longer than the original, so $k' = 10$. The weight of the representation is 4. Algorithm 2.12 will take $9\delta + 3\dot\alpha$.

In the last step of the recoding, the length of the exponent was increased without decreasing the weight. Using $(1100\bar1000\bar1)_2$ as encoding leads to a runtime of only $8\delta + 3\dot\alpha$. Either way, an improvement over the binary method is obtained.

The 2^w-ary method, as implicitly described by Brauer [**31**], uses the dictionary $\mathcal{D} = \{d|1 \le d < 2^w\}$ where w is a parameter of the method. The exponent n is simply written in the 2^w-ary number system as $n = \sum_{i=0}^{\lceil \frac{k}{w} \rceil - 1} d_i 2^{wi}$ where $0 \le d_i < 2^w$, i.e., $b_{i,d} = 1$ iff $i \equiv 0 \bmod w$ and $d_{i/w} = d$. The average weight is $(1-1/2^w)\lfloor k/w \rfloor + 1$ and the length $k' = k - (k \mod w)$. A slightly more efficient variation starts to read the blocks in the alternate direction, i.e., from left-to-right, which leads to the slightly shorter length $k' = k - w$ for the encoding. The precomputation can be done with the improved unary algorithm on input g and $2^w - 1$. It has often been observed that the even elements in the dictionary can be removed. Suppose that an element $d \in \mathcal{D}$ is divisible by 2^j for $j > 0$ (and not by 2^{j+1}). For all $0 \le i < k'$ with $b_{i,d} = 1$ we know that $b_{i-j,d/2^j} = 0$ since $i - j \not\equiv 0 \pmod w$. If we change the representation by setting $b_{i,d} = 0$ and $b_{i-j,d/2^j} = 1$ the same number is encoded. The average weight of the 2^w-ary method is $1 + \frac{2^w-1}{2^w}(\lceil \frac{k}{w} \rceil - 1)$.

EXAMPLE 2.16. Let $n = 367 = (101101111)_2$. The 4-ary method clumps together two bits at a time, so the encoding becomes $(1\underline{01}\,\underline{10}\,\underline{11}\,\underline{11})_2 = (101020303)_2$. The elements that need to be precomputed are g^2 and g^3, which costs $\alpha + \delta$. Working through the exponent costs $8\delta + 4\dot\alpha$, so the total cost is $\alpha + 9\delta + 4\dot\alpha$, which only is an improvement over the binary method if $\delta + \alpha < 2\dot\alpha$. Table 2.4 shows the computation in more detail.

The occurrence of 2 in the encoding can be avoided without changing the weight, since $(101020303)_2 = (101100303)_2$. This is the improved 2^w-ary method. For the present case $w = 2$ there is little difference, since g^2 is always needed for the computation of g^3 (however, in the improved version g^2 can soon be forgotten, saving some memory). Starting the encoding from the left results in a shorter encoding with $k' = 8$. The encoding is $(\underline{10}\,\underline{11}\,\underline{01}\,\underline{11}\,1)_2 = (20301031)_2$ and the resulting exponentiation costs $\alpha + 8\delta + 4\dot\alpha$.

As an alternative to using radix 2^w, one could also pick any other radix R, giving rise to the R-ary method. If R is not a power of two, the recoding of the exponent will be more expensive and there will relatively more multiplications compared to squarings (since taking an R-th power, the spine of the algorithm, will require multiplications if R is not a power of two).

EXAMPLE 2.17. To give an impression of the disadvantages of using a radix that is not a power of two, we examine $R = 3$ and $R = 5$. If $R = 3$ the value g^2 needs to be precomputed, costing δ. The ternary representation of n is $367 = (111121)_3$. It has length 6, requiring $6 - 1 = 5$ cubings costing $\alpha + \delta$ each and —we are a bit unlucky here— weight 6, requiring $6 - 1 = 5$ multiplications with either g or the precomputed g^2. The total cost is $5\alpha+6\delta+5\dot{\alpha}$, which is significantly more expensive than the binary method. If $R = 5$ is used, three values have to be precomputed, costing $\alpha + 2\delta$ in total. The pentary representation is $367 = (2432)_5$. Taking a fifth power costs $\alpha+2\delta$, and has to be performed thrice. Three additional multiplications are needed, since the weight is 4. The total cost using the pentary representation is $4\alpha + 8\delta + 3\dot{\alpha}$. This too is more expensive than the binary method.

A well-known improvement over the 2^w-ary method is the sliding window technique. The dictionary consists of all positive odd integers smaller than 2^w. The binary representation of n is then scanned either from left-to-right or from right-to-left, each time trying to use as large a window as possible. This has the effect of virtually increasing the window size by one. The precomputation requires the computation of g^2 and then one multiplication for each element in the dictionary (apart from 1). The sliding window method costs on average $\approx 2^{w-1}\alpha + k\delta + \frac{k}{w+1}\dot{\alpha}$ (based on an analysis by Cohen [**46**], who gives the more accurate runtime estimate $(2^{w-1} - 1)\alpha + (k - \frac{w(w-1)}{2(w+1)})\delta + (\frac{k}{w+1} - \frac{w(w+3)}{2(w+1)^2})\dot{\alpha})$.

EXAMPLE 2.18. The sliding window method with $w = 2$ recodes the exponent 367 as $(\underline{1}\,0\underline{11}\,0\underline{11}\,\underline{11}\,)_2 = (100300303)_2$. The precomputation of g^3 costs $\alpha + \delta$ and the processing of the exponent costs $8\delta + 3\dot{\alpha}$, bringing the total cost to $8\delta + 3\dot{\alpha} + \alpha$. A small improvement can be obtained by writing $(20300303)_2$, since it happens that g^2 is a byproduct of the precomputation. This variation costs only $8\delta + 3\dot{\alpha} + \alpha$.

The window methods can also be adapted to include cheap inversions. Avanzi [**6**] gives an overview of several of these methods. As a rule of thumb, the availability of inversions make your window effectively one bit longer. The signed sliding window method [**177**] uses the dictionary $\mathcal{D} = \{-2^w + 1, -2^w + 3, \ldots, 2^w - 3, 2^w - 1\}$, which can be precomputed for $(2^{w-1} - 1)\alpha + \delta$. The exponent is recoded in such a way that of any $w + 1$ consecutive positions at most one of the $b_{i,d}$ is set. The average behaviour has been studied by Cohen [**46**]. Roughly speaking, the algorithm will on average take $2^{w-1}\alpha + k\delta + \frac{k}{w+2}\dot{\alpha}$.

Bos and Coster [**25, 26**] observe that if w grows, more and more elements in the dictionary will not be used in the exponent recoding, so it makes no sense to precompute these elements and it pays off to try and find an optimal addition chain for those values that are used. Bos and Coster describe several heuristics, mostly

based on Euclidean techniques (discussed in Section 2.3.3). Bos and Coster claim there method produces chains of length 605 on average for a 512-bit exponent. This is 5% faster than the sliding window method (with $w = 5$). As an aside, Bos and Coster conclude with the remark that they expect improvements using simulated annealing could get the figure below 600. Nedjah and de Macedo Mourelle [**137**] describe an algorithm based on genetic algorithms for the computation of an addition chain for one exponent only, but finding these chains is reported to take 4 to 6 seconds.

EXAMPLE 2.19. Suppose that Bos and Coster's heuristic is used for $n = 367$ with initial window size 4. The exponent is then cut in pieces as $(\underline{1011}\,0\underline{1111}\,)_2 = 11 \cdot 2^5 + 15$. The simultaneous computation of g^{11} and g^{15} can be done for $\alpha + 3\delta + \dot{\alpha}$ based on the chain $\langle 1, 2, 3, 4, 8, 11, 15\rangle$. The actual computation takes $\alpha + 6\delta$, so the total costs are $2\alpha + 9\delta + \dot{\alpha}$. (Note that we use an ordinary α and not a $\dot{\alpha}$ in the final stage.)

A different approach is taken by Yacobi [**200**]. He proposes to use Lempel-Ziv compression [**205, 206**] to recode the exponent. The elements in the dictionary are considered as binary words (the binary representation of the element) and initially the dictionary is empty. The binary expansion of the exponent is scanned from left to right. Each time the longest word from the dictionary that fits as prefix of the unscanned part of the exponent is selected and extended with the next symbol (bit) in the exponent. This newly formed word is added to the dictionary and the scanning of the exponent continues. The result of this method is that the dictionary is slowly formed and a natural addition sequence for the elements in the dictionary exists, costing δ if the 'extension bit' is a 0 and $\alpha + \delta$ for a 1. (Actually, if the extension bit is a 1, the dictionary could also be expanded with both extension bit 0 and 1.) Yacobi remarks that in the context of exponentiation a 'sliding' version of the Lempel-Ziv algorithm can be used. Whenever the unscanned part of an exponent has trailing zeroes, these zeroes can be skipped. Obviously the method also works for other alphabets (to use compression terminology). In fact, it also works for multi-exponentiation by using an alphabet consisting of vectors.

EXAMPLE 2.20. Suppose the original Lempel-Ziv algorithm is used on the exponent 376. The exponent is then encoded as $(\underline{1}\,\underline{0}\,\underline{11}\,\underline{01}\,\underline{111}\,)_2 = 1 \cdot 2^8 + 0 \cdot 2^7 + 3 \cdot 2^5 + 1 \cdot 2^3 + 7 \cdot 2^0$. Computing the dictionary costs $2\alpha + 2\delta$ or $3\alpha + 3\delta$ if the algorithm does not exploit that $(01)_2$ is already known. Processing the encoding costs $3\alpha + 8\delta$ or $4\alpha + 8\delta$ if the algorithm uses a multiplication for $\underline{0}$. The total costs are $5\alpha + 10\delta$ or $7\alpha + 11\delta$ for a silly implementation. Using the sliding version eases the computation. The exponent is encoded as $(\underline{1}\,0\underline{11}\,0\underline{111}\,\underline{1}\,)_2 = 1 \cdot 2^8 + 3 \cdot 2^5 + 7 \cdot 2 + 1 \cdot 2^0$. Precomputation takes $2\alpha + 2\delta$ and processing $8\delta + 3\alpha$, for a total cost of $10\delta + 5\alpha$. The Lempel-Ziv algorithm can also be run with initial dictionary $\{(1)_2\}$. In this case the exponent is encoded as $(\underline{10}\,\underline{11}\,0\underline{111}\,\underline{1}\,)_2 = 2 \cdot 2^7 + 3 \cdot 2^5 + 7 \cdot 2 + 1 \cdot 2^0$. If in the precomputation stage $(10)_2$ and $(11)_2$ are used simultaneously, the total costs decrease to $9\delta + 5\alpha$.

Exponentiation based on the Lempel-Ziv algorithm is attractive if the entropy of the source generating the exponent is low (and especially if the number of ones is high). Other data compression methods can also be used for exponentiation routines. Bocharova and Kudryashov [**23**] propose to use a variable-to-fixed length source coding algorithm by Tunstall [**188**]. If the entropy of the source is 1, this leads to the sliding window method [**16**]. (Well, to the same dictionary as used by the sliding window method and the same encoding. If the dictionary is constructed on-the-fly, as the dictionary is being built by Tunstall's algorithm, the precomputation seems to be costlier for Bocharova and Kudryashov's version of the sliding window method.) Gollman et al. [**67**] propose to use a dictionary that consists of elements 2^i-1 for $0 < i \leq w$, corresponding to all-one sequences in the binary representation. Precomputation for the dictionary takes $\alpha(w-1)+\delta(w-1)$ and the average weight of an encoding is $\approx \frac{2^{w-2}}{2^w-1}k$, so the total costs of this method are $(k+w-2)\delta+(w-1)\alpha+\frac{2^{w-2}}{2^w-1}k\dot{\alpha}$. A more detailed analysis of the algorithm is provided by O'Connor [**138**].

Example 2.21. We will not give an example of using Tunstall source coding and restrict ourselves to an example of Gollman et al.'s proposal. Let $n = 367$ be the exponent and suppose $w = 4$. The dictionary consists of $\{1, 3, 7, 15\}$ and an addition sequence for this dictionary is $\langle 1, 2, 3, 6, 7, 14, 15\rangle$. The exponent is subsequently recoded as $(\underline{1}\,0\underline{11}\,0\underline{1111}\,)_2 = 1\cdot 2^8+3\cdot 2^5+15\cdot 2^0$. The precomputation takes $3\alpha + 3\delta$ and the processing $8\delta + 2\dot{\alpha}$, giving a total cost of $3Ga + 11\delta + 2\dot{\alpha}$.

Pippenger [**149**] and Yao [**202**] suggest to split the exponent in $\lceil \frac{k}{w} \rceil$ pieces of length w each based on the 2^w-ary representation. (trailing zeroes might reduce the effective length of some pieces). The difference with the 2^w-ary method, is that the dictionary $\mathcal{D} = \{1, 2^w, 2^{2w}, \ldots, 2^{\lfloor \frac{k-1}{w} \rfloor w}\}$ is used for this splitting. Precomputation for this dictionary takes $\lfloor \frac{k-1}{w} \rfloor w\delta$.

Yao uses a dual algorithm of the 2^w-ary method. If $n = \sum_{i=0}^{k'-1} n_i 2^{iw}$, he first computes for all $1 \leq d < 2^w$ the product $\tilde{G}_d = g^{\tilde{n}_d}$ with exponent $\tilde{n}_d = \sum_{0\leq i<k', n_i=d} 2^{iw}$. Since all the powers $g^{2^{iw}}$ have already been computed, this costs at most $k' = \lceil \frac{k}{w} \rceil$ group multiplications. Combining everything can be done using Algorithm 2.4 or the dual of Algorithm 2.6. (Yao uses a less efficient algorithm for this final step [**16**].) The total costs are then at most $(2^w + \lceil \frac{k}{w} \rceil - 2)\alpha + (2^w + \lfloor \frac{k-1}{w} \rfloor w - 2)\delta$. Yao suggests to pick $w = \lfloor \lg k - 3 \lg \lg k \rfloor$ to obtain good asymptotic results, but for $60 < k < 200$ it makes more sense to use $w = 3$. A big advantage of Yao's algorithm —actually his original motivation— is its cheap processing of the exponent. Most of the work is done during the preprocessing, independent of the exponent. As a result the algorithm is very well suited for simultaneous exponentiation. As a concrete example, suppose that $k = 160$ and $w = 3$. For the precomputation 159 squarings are needed, but subsequently only 60 multiplications and 6 squarings per exponent are needed.

Pippenger tackles the resulting multi-exponentiation by using a dictionary that contains all $(0, 1)$-vectors of length $\lceil \frac{k}{w} \rceil$. If all these vectors would actually occur in the encoding of the exponent, the precomputation of all these vectors would take

$(2^{\lceil \frac{k}{w} \rceil} - \lceil \frac{k}{w} \rceil)\alpha$, but for decreasing window size w this will be less likely. Pippenger uses an ingenious algorithm to compute only those vectors that are needed (part of the algorithm will be described later). It is not too hard to see that the number of multiplications needed cannot exceed the number of ones in the binary representation of the exponent, giving an average upper bound of $\frac{k}{2}\alpha$. (In fact, if this value is used the algorithm is no more efficient than the binary algorithms). The processing itself can be expected to take slightly less than $(\alpha+\delta)(w-1)$, so the total costs are at most $(2^{\lceil \frac{k}{w} \rceil} - \lceil \frac{k}{w} \rceil + w - 1)\alpha + (\lfloor \frac{k-1}{w} \rfloor w + w - 1)\delta$.

EXAMPLE 2.22. Suppose $w = 3$ and the exponent is $n = 367$. The dictionary is $\{1, 2^3, 2^6\}$ so it takes 6δ to compute the corresponding values G_d. The exponent is rewritten as $(\underline{101}\,\underline{101}\,\underline{111}\,)_2 = 5 \cdot 2^6 + 5 \cdot 2^3 + 7$. The resulting multi-exponentiation is based on the matrix

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

Computing the columns in this matrix costs 2α (only $(1,1,1)^T$ needs to be computed). The processing of the exponent costs $2\alpha + 2\delta$. The total costs are $4\alpha + 8\delta$.

Dictionaries for Double Exponentiation and Beyond. The number of direct applications of multi-exponentiation (beyond double exponentiation) in cryptography is limited. Its main relevance lies in its interwovenness with single and double exponentiation, which already showed a little from the generic square-and-multiply algorithm but which will become even more potent when taking off-line precomputation into account.

For a double exponentiation, i.e., the computation of $g^n h^m$ for given group elements g and h and two random k-bit exponents n and m, the trivial dictionary is $\mathcal{D} = \{\binom{1}{0}, \binom{0}{1}\}$. This requires no precomputation and the encoding is unique, following the binary expansions of both exponents. Hence, the weight of the combined representation is $k-1$ on average. The total runtime is $(k-1)\dot{\alpha} + (k-1)\delta$.

Interestingly, this is already faster than using two separate binary exponentiations (and multiplying the results). One set of $k-1$ group squarings is saved. Actually, there is no reason to restrict this interleaving to the binary method. Any two single exponentiation routines can be combined this way, saving on the number of squarings. Generalization to more exponents is straightforward. Möller [**127**] discusses several combinations, also taking into account the possibility of cheap inversions. We will limit ourselves to the sliding window and signed sliding window method, depending on whether we can use a signed method or not.

A different approach is taken by Straus [**185**]. In the easiest case of double exponentiation, the dictionary is chosen as $\mathcal{D} = \{\binom{0}{1}, \binom{1}{0}, \binom{1}{1}\}$. The precomputation requires one multiplication (of gh, corresponding to $\binom{1}{1}$). Both exponents are expanded binary and whenever possible $\binom{1}{1}$ is used. The resulting weight of this method, which is commonly called Shamir's method, is only $\frac{3}{4}(k-1)$, one quarter less than the trivial method. Note the resemblance with the method discussed at

the beginning of this section (which itself is nothing other than the 4-ary method in disguise).

In the more general case, Straus proposed to precompute all vectors whose components lie between 0 and 2^w (exclusive). The exponent is chopped up in pieces of w bits each, just like in the w-ary algorithm. Straus' dictionary has cardinality $2^{mw} - 1$, including m unit vectors. Note that the case $w = 1$ corresponds to the multi-exponentiation performed by the version of Pippenger's algorithm we already described. Straus did not distinguish between squarings and multiplications and needs $2^{mw} - m - 1$ operations for the precomputation. Those $G_{\mathbf{d}}$ for which d contains even numbers only can be computed using a squaring. This affects $2^{(w-1)m} - 1$ elements. Computation of these elements can be skipped altogether by using a sliding window technique, as proposed by Yen et al. [**204**]. Avanzi [**6**] gives an analysis of the method. He also considers the method based on the NAFs of the exponents instead of the binary expansions.

Pippenger also describes another algorithm based on the work of Lupanov [**117**]. Suppose m exponents of bitlength k are given, where $km < 2^m$. This is the kind of situation that arises from splitting the exponent as described for a single exponentiation (the splitting technique can easily be extended for multi-exponentiation, although the use of duality might be preferable). Precomputing *all* 2^m $(0, 1)$ vectors is a bad idea. Most of these will not be used and the trivial dictionary consisting of the m unit vectors will give better results. There is a big gap between these two extremes. A gap that can be filled by partitioning the bases (or the input). Let t be a parameter, for simplicity we will assume that t divides m. Partition the m bases in $\frac{m}{t}$ groups of t each and compute for each of these set of t exponents all 2^t different products. The cost of this precomputation is $\frac{m}{t}(2^t - t - 1)\alpha$ and the main computation (Steps 4–7 of Algorithm 2.12) will cost approximately $(\frac{m}{t}\frac{2^t-1}{2^t}k)\alpha + (k-1)\delta$. The two extremes correspond to the trivial divisors $t = 1$ and $t = m$.

If inversion is for free, one could consider combining the NAF with Shamir's trick based on $\mathcal{D} = \{\binom{0}{1}, \binom{1}{0}, \binom{1}{1}, \binom{1}{-1}\}$ and their negations. Given two random exponents, each having a NAF of length about k and an expected number of $2k/3$ zeroes, on average in $\frac{4}{9}$ of the positions we encounter $\binom{0}{0}$. This leaves $\frac{5}{9}k$ nonzero $b_{i,\mathbf{d}}$'s, resulting in a runtime of $2\alpha + (\delta + \frac{5}{9}\dot{\alpha})k$.

Solinas [**178**] notes that computing the NAFs independent of each other might not be optimal to minimize the number of nonzero $b_{i,\mathbf{d}}$. As an alternative he proposes the joint sparse form. One of its most important properties is that there are at most two consecutive nonzero $b_{i,\mathbf{d}}$'s. The joint sparse form can be efficiently computed and on average, half of the resulting $b_{i,\mathbf{d}}$ will be nonzero. The running time is therefore $2\alpha + (\delta + \frac{1}{2}\dot{\alpha})k$. A window of size two requires more precomputation, but is generally faster for larger bitlengths, with running time $10\alpha + (\delta + \frac{3}{8}\dot{\alpha})k$, as analysed by Avanzi [**6**].

2.3.3. Euclidean Algorithms. In its additive form, to perform a multiplication using only additions, doublings and halvings, Algorithm 2.9 has been dated back to

the Egyptians living 2000 B.C. Exponentiation routines can also be based on a somewhat younger algorithm, the basics of which are described by Euclid (c. 300 B.C.). Euclid's algorithm returns the greatest common divisor of two positive integers.

ALGORITHM 2.23 (Euclid's GCD Algorithm).
On input two nonnegative integers n and m this algorithm outputs $\gcd(n, m)$. It keeps invariant

$$0 \le e < d, \quad \gcd(d, e) = \gcd(n, m) \ .$$

1. [Initialize] If $n > m$ set $(d, e) \leftarrow (n, m)$ else set $(d, e) \leftarrow (m, n)$.
2. [Finished?] If $e = 0$, terminate with output d.
3. [Decrease (d, e)] Set $(d, e) \leftarrow (e, d \bmod e)$ and return to the previous step.

A slight adaptation of the algorithm leads to the extended Euclidean algorithm: it also returns how the greatest common divisor can be seen as a $\mathbb{Z}$-linear combination of the input. Over the years Euclid's algorithm has been successfully ported to numerous other settings. Knuth [**89, 90**] and Cohen [**45**] give excellent coverage of the algorithm. Akhavi and Vallée [**4**] give an overview, and average-bit complexity analysis, of several variations to the classical version. The elementary equation behind the classical version is $d = \lfloor \frac{d}{e} \rfloor e + (d \bmod e)$ where $0 \le d \mod e < e$. Redefining mod (and *div* corresponding to $\lfloor \frac{d}{e} \rfloor$) such that $-e < (d \bmod e) \le 0$ produces the surprisingly slow by-excess algorithm. If mod is redefined such that $-e/2 < (d \bmod e) < e/2$ the centred variation emerges. In the subtractive algorithm (d, e) is substituted with $(d - e, e)$ (give or take the ordering). The binary algorithm, finally, has (d, e) substituted with $(e, (d - e)/2^b)$ (here 2^b is the largest power dividing $d - e$).

We now turn our attention to double exponentiation with bases g and h and exponents n and m. Suppose that g and h are in the same cyclic group $\langle f \rangle$, so we could write $g = f^\kappa$ and $h = f^\lambda$ (it does not matter that we most likely cannot actually compute κ and λ, existence is sufficient here). The double exponentiation can be rewritten as $g^n h^m = f^{n\kappa + m\lambda}$. Semba [**165**] and Bergeron et al. [**14**] exploit this in a clever way.

ALGORITHM 2.24 (Euclidean Double Exponentiation).
On input two bases $g = f^\kappa$ and $h = f^\lambda$ and two nonnegative exponents n and m this algorithm outputs $g^n h^m$. It keeps invariant

$$0 \le e < d, \quad \gcd(d, e) = \gcd(n, m), \quad ad + be = n\kappa + m\lambda, \quad A = g^a, \quad B = g^b \ . \tag{7}$$

1. [Initialize] If $n < m$ set $(d, e) \leftarrow (m, n), (a, b) \leftarrow (\lambda, \kappa)$ and $(A, B) \leftarrow (h, g)$ else set $(d, e) \leftarrow (n, m), (a, b) \leftarrow (\kappa, \lambda)$ and $(A, B) \leftarrow (g, h)$.
2. [Finished?] If $e = 0$, compute A^d using a single exponentiation routine and terminate with output A^d.
3. [Decrease (d, e)] Compute $A^{\lfloor \frac{d}{e} \rfloor}$ using a single exponentiation routine. Set $(a, b) \leftarrow (\lfloor \frac{d}{e} \rfloor a + b, a), (A, B) \leftarrow (A^{\lfloor \frac{d}{e} \rfloor} B, A)$, and $(d, e) \leftarrow (e, d \mod e)$. Return to the previous step.

In the algorithm above the restriction that g and h have to be in the same cyclic subgroup can be removed by slightly changing the invariant (7) into

$$0 \leq e < d, \quad \gcd(d, e) = \gcd(n, m), \quad A^d B^e = g^n h^m \ .$$

The dual of the algorithm above can be used to compute a twofold exponentiation. One way of implementing the algorithm is by using a recursive version instead of an iterative one.

Algorithm 2.25 (Euclidean Twofold Exponentiation).
On input a base g and two positive exponents n and m this algorithm outputs g^n and g^m.

1. [Initialize] Let $u = \gcd(n, m)$. Compute g^u using a single exponentiation routine. Set $(A, B) \leftarrow (g^u, g^u)$. If $n > m$ set $(d, e) \leftarrow (m/u, n/u)$ else set $(d, e) \leftarrow (n/u, m/u)$.

2. [Finished?] If $e = 0$, terminate with output A and *id*.

3. [Decrease (d, e)] Set $(A, B) \leftarrow (g^e, g^{d \bmod e})$ using a recursive call to this algorithm (Step 1 will have $u = 1$). Compute $A^{\lfloor \frac{d}{e} \rfloor}$ using a single exponentiation routine and terminate with output $A^{\lfloor \frac{d}{e} \rfloor} B$ and A.

The runtime of both algorithms will clearly depend on the single exponentiation routine that is used. In the first iteration d and e might differ in bitlength considerably, requiring a significant single exponentiation. However, in subsequent iterations d and e will be approximately equally large and the single exponentiations will only be small. Bos [**26**] reports a difference between the binary method and optimal addition chains of only 2% (although his figure is not entirely general). Simulations suggest an estimated average runtime of $(0.72\delta + 0.87\alpha)k$ for k bits exponents. A very crude application of the analysis of the Euclidean algorithm also gives this result. The probability that the quotient $\lfloor \frac{d}{e} \rfloor$ in Step 3 equals c is about $\lg((c+1)^2/((c+1)^2 - 1))$ and processing a quotient c costs $(\lfloor \lg c \rfloor - 1)\delta$ plus one α for each 1 in the binary representation of c (assuming the binary algorithm is used for the single exponentiation in Step 3). This results in an estimated average cost for each time Step 3 has to be performed. The average can be numerically approximated by assuming that the quotient $\lfloor \frac{d}{e} \rfloor$ does not exceed some bound (the higher the bound, the smaller the error in the estimate). The number of times Step 3 has to be performed for a k-bit exponent is approximately $12(\ln 2/\pi)^2 k$. Multiplying this with the average costs per step gives the aforementioned runtime $(0.72\delta + 0.87\alpha)k$.

Example 2.26. Let $n = 22$ and $m = 15$ and suppose two bases g and h are given. Table 2.6 shows the values of the important variables during the performance of Algorithm 2.24. The total cost is $5\alpha + 3\delta$.

Both algorithms can be generalized to more than two exponents. For a simultaneous exponentiation, let $\mathcal{S}$ be the set of exponents that need to be computed. Let d and e be the two largest elements of $\mathcal{S}$ with $d > e$. Replace d in $\mathcal{S}$ with $d \bmod e$ and use the algorithm to perform the simultaneous exponentiation with this modified set of exponents. Afterwards, g^d can be reconstructed by $(g^e)^{\lfloor \frac{d}{e} \rfloor}(g^{d \bmod e})$ using a

TABLE 2.6. Euclidean double exponentiation for $n = 22$ and $m = 15$

Round	(Ini)	1	2	3
Based on		$22 = 1 \cdot 15 + 7$	$15 = 2 \cdot 7 + 1$	$7 = 7 \cdot 1 + 0$
d	22	15	7	1
e	15	7	1	0
A	g	gh	$(gh)^2 \cdot g = g^3h^2$	$(g^3h^2)^7 \cdot gh = g^{22}h^{15}$
B	h	g	gh	g^3h^2
Cost	0	α	$\alpha + \delta$	$\alpha + (2\alpha + 2\delta)$

single exponentiation routine for the exponentiation to the power $\lfloor \frac{d}{e} \rfloor$. The exact analysis of the resulting algorithm seems quite hard [**51**], but there are a couple of easy observations. The most important one is that the number of steps will likely increase and at the same time the average value of $\lfloor \frac{d}{e} \rfloor$ will decrease. The number of steps with $\lfloor \frac{d}{e} \rfloor = 1$ in particular will increase, and these steps are relatively inefficient. For a triple exponentiation, simulations show an average runtime of $(1.20\alpha + 0.56\delta)k$ for three random k-bit exponents.

Bergeron et al. [**13**] present a framework for performing single exponentiations based on the Euclidean algorithms above. A single exponentiation g^n is replaced by the twofold exponentiation g^n and g^m with $0 < m < n$. Algorithm 2.25 is called to obtain g^n and g^m, the latter of which is discarded. (Alternatively, a double exponentiation $g^{n-m}g^m, 0 < m < n$ can be used.)

ALGORITHM 2.27 (Euclidean Single Exponentiation).
On input a base g and a positive exponent n this algorithm outputs g^n.

1. [Special cases] If n is a power of 2, compute g^n using repeated squarings. If $n = 3$, compute g^3 using a squaring and a multiplication. In both these cases, terminate with output g^n.
2. [Pick m] Pick m based on n using any chosen strategy (subject to $0 < m < n$).
3. [Call twofold algorithm] Compute g^n and g^m using a call to Algorithm 2.25 on appropriate input. Terminate with output g^n.

Bergeron et al. give a list of several strategies to pick m given n. Some of these are non-deterministic, which requires one to exhaustively search all possibilities for the optimal one before starting to do any group operations. Of special interest are the deterministic strategies.

In the binary strategy $m = \lfloor \frac{n}{2} \rfloor$ is used. The resulting single exponentiation is equivalent to the left-to-right binary algorithm: the same group elements are computed. The right-to-left binary algorithm emerges by choosing $m = 2$ (for $n > 3$). Bergeron et al. also give the co-binary strategy $m = \lfloor \frac{n+1}{2} \rfloor$.

The dichotomic strategy picks $m = \lfloor \frac{n}{2^{\lceil \lfloor \lg n \rfloor / 2 \rceil}} \rfloor$. The first time Step 3 is encountered, $\lfloor \frac{d}{e} \rfloor = \lfloor \frac{n}{m} \rfloor = 2^{\lfloor \frac{k}{2} \rfloor}$ or maybe one more. The costs for this step are $\approx 0.5\delta$ and afterwards a more or less random double (or twofold) $\frac{k}{2}$-bit exponentiation has to be performed, which can be expected to cost $\approx (0.87\alpha + 0.72\delta)k/2$. The total costs

for a single exponentiation are $(0.43\alpha + 0.86\delta)k$ which corresponds to a simulation of 1000 random single k-bit exponentiations for $k \in \{20, 40, \ldots, 2000\}$.

The third and final deterministic strategy, the factor strategy, is of theoretic interest only. If n is prime it uses $m = n - 1$, otherwise m is chosen to be the smallest prime divisor of n. This choice of m effectively requires one to factor the exponent. For 160-bit exponents this is a not entirely trivial task that would be too time consuming in a practical cryptographic application, for 1024-bit exponents this would constitute a breakthrough in integer factorization.

EXAMPLE 2.28. Let $n = 367$. The dichotomic strategy picks $m = 22$. The first time in Step 3 (of Algorithm 2.25) $\lfloor \frac{367}{22} \rfloor = 16$ and a recursive call with exponents 22 and 367 mod 22 = 15 is made. The costs are $\alpha + 4\delta$ for computing the power of 16 and combining the result and $5\alpha + 3\delta$ for the recursive call (see Example 2.26), or $6\alpha + 7\delta$ in total.

Using the factor method initially picks $m = 366$, since 367 is prime. Computing g^{366} is based on the factorization $366 = 2 \cdot 3 \cdot 61$. The small primes take δ and $\alpha + \delta$ and for 61 the factorization $60 = 2^2 \cdot 3 \cdot 5$ is used, costing $2\alpha + 5\delta$. Twice a large prime was encountered, so the total cost is $5\alpha + 6\delta$.

2.3.4. Exploiting Endomorphisms. We already saw that the availability of free inversion can reduce the costs of an exponentiation. Other easy-to-compute endomorphisms can also speed up an exponentiation.

In a finite field $\mathbb{F}_{p^e}$ the Frobenius endomorphism (p-th powering) can be used to split an exponent $0 < n < p^e$ in e pieces all smaller than p. A multi-exponentiation algorithm can be used to deal with the resulting exponentiation. If the characteristic p is small, and the extension e large, the use of word chains and compression based addition chains pays off: the cheap endomorphism takes on the role of squaring, greatly reducing the total cost of the exponentiation.

A similar effect occurs with Koblitz curves or other elliptic curves where the coefficients that define the curve are chosen from a smaller field than the points on the curve (cf. Chapter 5). The Frobenius endomorphism ϕ satisfies a quadratic equation that allows writing an exponent as $n = \sum_{i=0}^{k'} n_i \phi^i$ over $\mathbb{Z}[\phi]$ where the n_i are smaller than the size of the field of the coefficients. (Blake et al. [**17**] give more details.)

For a lot of elliptic curves used in cryptography the points are defined over the same field from which the coefficients are drawn. In this case the Frobenius endomorphism is actually the identity function and hence of little use. Gallant et al. [**64**] point out that for some curves other relatively cheap endomorphisms can be used to speed up the exponentiation. Suppose that the endomorphism corresponds to (scalar) multiplication by p, then the exponent is written as $n \equiv n_1 + n_2 p \bmod q$ where q is the group order and n_1 and n_2 are integers of roughly the same size as $\sqrt{q}$. The exponent has been split in two pieces and a k-bit single exponentiation is turned into a $\frac{k}{2}$-bit double exponentiation.

Gallant et al. give several examples and an algorithm to perform the splitting but they do not provide any proof that the resulting n_1 and n_2 are of the desired

magnitude. In a special case, the lemma below shows that the split is indeed giving small n_1 and n_2.

LEMMA 2.29. *Let $q|(p^2-p+1)$ and let $n \in \mathbb{Z}$, then there exist $n_1, n_2 \in \mathbb{Z}$ such that $n_1 + n_2 p \equiv n \bmod q$ and $|n_1|, |n_2| < 2\sqrt{q}$.*

Proof: Let L be the two-dimensional integral lattice $\{(e_1, e_2)^T \in \mathbb{Z}^2 : e_1 + e_2 p \equiv 0 \bmod q\}$. If $(e_1, e_2)^T \in L$, then

$$(e_1 + e_2) - e_1 p \equiv -e_2 p + e_2 + e_2 p^2 = e_2(p^2 - p + 1) \equiv 0 \bmod q$$

so that $(e_1 + e_2, -e_1)^T \in L$. Let $\mathbf{v}_1 = (e_1, e_2)^T$ be the shortest non-zero vector of L (using the L_2-norm). It may be assumed that $e_1 \geq 0$. It follows that $e_2 \geq 0$, because otherwise $(e_1 + e_2, -e_1)^T$ or $(-e_2, e_1 + e_2)^T \in L$ would be shorter than $\mathbf{v}_1$. If $\mathbf{v}_2$ is the shortest of $(e_1 + e_2, -e_1)^T, (-e_2, e_1 + e_2)^T \in L$, then $|\mathbf{v}_2| < 2|\mathbf{v}_1|$ and $\{\mathbf{v}_1, \mathbf{v}_2\}$ is easily seen to be a shortest basis for L, with $e_1^2 + e_1 e_2 + e_2^2 = q$ and $e_1, e_2 \leq \sqrt{q}$. This implies that given $\{\mathbf{v}_1, \mathbf{v}_2\}$ and any integer vector $(-n, 0)^T$, there is a vector $(n_1, n_2)^T$ with $0 \leq n_1, n_2 \leq 2\sqrt{q}$ such that $(-n + n_1, n_2)^T \in L$. It follows that $-n + n_1 + n_2 p \equiv 0 \bmod q$, i.e., $n \equiv n_1 + n_2 p \bmod q$ as desired. Using the initial basis $\{(q, 0)^T, (-p, 1)^T\}$, the vector $\mathbf{v}_1$ can be found quickly [**45**, Algorithm 1.3.14], and for any n the vector $(n_1, n_2)^T$ can easily be computed. *Q.E.D.*

The case $q|(p^2 + 1)$ can be dealt with similarly. A more general and thorough treatment is given by Sica et al. [**174**].

EXAMPLE 2.30. Suppose $p = 409$ and $q = 769$, so $q|(p^2 - p + 1)$, and let $g \in \mathsf{G}_q$. The vector $\mathbf{v}_1$ from the proof above equals $(17, 15)^T$ and indeed $17 + 15 \cdot 409 = 8 \cdot 769$. It follows that $\mathbf{v}_2 = (-15, 32)^T$. The vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ satisfy $(g, g^p)^{\mathbf{v}_1} = (g, g^p)^{\mathbf{v}_2} = id$. The exponent $n = 367$ corresponds to $(367, 0)^T = 15\mathbf{v}_1 - 7\mathbf{v}_2 + (7, -1)^T$, which means that $g^{367} = g^7 \cdot (g^p)^{-1}$. If inversions are problematic, we can also use $g^{367} = g^{24} \cdot (g^p)^{14}$. The last double exponentiation can be computed in $4\delta + 4\alpha$.

2.3.5. Exploiting Precomputation. If the base is fixed, it pays off to precompute certain powers of the base. These precomputed values are then stored and can be reused for each exponentiation with that base that follows. For instance if single exponentiation is based on Algorithm 2.12 all values g^d with $d \in \mathcal{D}$ can be precomputed (instead of the on-line "precomputation" in Step 2 of the algorithm).

During the discussion of the generic square-and-multiply algorithm it was already remarked that the first phase (precomputation) consists of the simultaneous computation of several powers of g (that is taken off-line now) and that the second phase is in fact a multi-exponentiation combining the precomputed values. Along these lines, Brickell et al. [**32**] write an exponent as $n = \sum_{i=0}^{k'-1} b_i m_i R^i$ with all $0 \leq b_i < h$ and $m_i \in M$, where M is some predetermined set of integers and R is the radix. All powers g^{mR^i} with $m \in M$ and $0 \leq i \leq \lceil \log_R n \rceil$ have to be precomputed and stored. The exponentiation itself is based on Algorithm 2.4. Brickel et al. give several suggestions for R, M, and h, also exploiting inversions whenever possible. If R is a power of two, $M = \{1\}$ and $h = R - 1$ the algorithm is similar to Yao's algorithm.

Precomputation essentially offers a space-time trade-off. The more elements are precomputed, the cheaper the exponentiations become. However, the first few elements give the most spectacular improvement. After a while the gain per precomputed element decreases. For a 160-bit exponent, Brickell et al. describe a method where 2244 elements are precomputed and on average 21.68 multiplications are still needed for the exponentiation. They also describe another method requiring 2751 precomputed values resulting in an average runtime of 20.92 multiplications. Over 300 extra precomputed values reduce the runtime by only 3.5%. The methods by Brickell et al. might not give the optimal trade-off.

De Rooij [**56**] proposes to deal with the multi-exponentiation that results from splitting the exponent by using Algorithm 2.24. Unfortunately, this method is neither suited if the number of precomputed elements is too high nor if the exponent is split in large pieces (so if the number of precomputed elements is too low).

Lim and Lee [**112**] describe what is now known as the comb method. It is equivalent to (the simplified version of) Pippenger's algorithm.

2.4. Conclusion

There is large number of exponentiation algorithms. The Euclidean algorithms are in general slower than the square-and-multiply algorithms. The simple algorithms benefit the most from cheap $\dot{\alpha}$ compared to α. The advantage of free inversions becomes smaller if the algorithms become more complex (but still worthwhile to exploit).

Table 2.7 contains an overview of some of the methods with or without inversions. To improve readability, the runtimes are approximates; in Chapters 4 and 5 more accurate versions are used. For the applications we have in mind —based on exponents of up to 200 bits and with free inversions in the group— a typical choice is either the NAF or the signed sliding window method with $w = 4$ for a single exponentiation. For a double exponentiation the JSF seems most appropriate, depending on the specific application with or without windows. Single exponentiation with a fixed base can be done using Pippenger's algorithm with $w = k/10$ and $t = 5$ (other choices for w and t are possible of course, but this gives a very decent result based on 62 precomputed values). The only disadvantage of Pippenger's algorithm is that it does not exploit free inversions.

TABLE 2.7. Exponentiation Routines

Name	Inv	Expected Runtime	Memory Required
Single exponentiation			
binary	No	$(\delta + \frac{1}{2}\dot{\alpha})(k-1)$	2
NAF/Star form	Yes	$(\delta + \frac{1}{3}\dot{\alpha})k$	2
sliding window	No	$2^{w-1}\alpha + (\delta + \frac{1}{w+1}\dot{\alpha})k$	2^{w-1}
signed sliding window (SSW)	Yes	$2^{w-1}\alpha + (\delta + \frac{1}{w+2}\dot{\alpha})k$	2^{w-1}
Double exponentiation			
interleaved sliding windows	No	$2^{w}\alpha + (\delta + \frac{m}{w+1}\dot{\alpha})k$	$m2^{w-1} - m + 1$
interleaved signed sliding window	Yes	$2^{w}\alpha + (\delta + \frac{2}{w+2}\dot{\alpha})k$	$2^{w} - 1$
Joint Sparse Form (JSF)	Yes	$2\alpha + (\delta + \frac{1}{2}\dot{\alpha})k$	5
windowed Joint Sparse Form	Yes	$10\alpha + (\delta + \frac{3}{8}\dot{\alpha})k$	13
Single exponentiation with precomputation			
simplified Pippenger (comb)	No	$(\delta + \frac{\lceil \frac{k}{w} \rceil}{t}(\frac{2^t-1}{2^t})\dot{\alpha})(w-1)$	$\frac{\lceil \frac{k}{w} \rceil}{t}(2^t - 1)$

3

Higher Order Addition Chains

In the previous chapter we discussed addition chains for first order recurrences. In this chapter we will treat higher order addition chains for higher order recurrences of a special form. In particular we will discuss Lucas chains for second order and Perrin chains for third order recurrences. Applications of these chains in cryptography are discussed in Chapters 4 and 5.

3.1. Motivation and Definitions

Additions chains give rise to efficient algorithms to perform exponentiations in multiplicative groups. In Section 1.3 higher order recurrences were described. Two special classes of recurrences were considered: second order recurrences where the $(n+m)$-th term could be computed based on the n-th, m-th and $(n-m)$-th term; and third order where the $(n-2m)$-th term was also necessary.

3.1.1. Lucas Chains. Efficient computation of second order recurrences can be done with a special class of addition chains, known as strong addition chains, Lucas chains or, according to Knuth, Chebyshev chains. We will however make a distinction between strong addition chains and Lucas chains. For the definition of a strong addition chain we follow the traditional definition of a Lucas chain [**19, 128**], resembling Definition 2.1 of an addition chain. The inclusion of $c_0 = 0$ is motivated by the need of 0 for a doubling and the use of negative exponents to denote $c_{-n} = -c_n$ later on. The number of elements in C is $k+2$ of which the first two are for free, so the effective length is k.

Definition 3.1 (Strong addition chain). A sequence of nonnegative integers

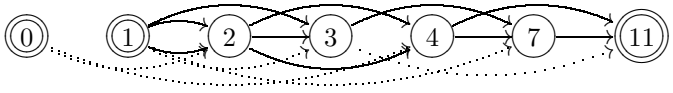
$$C = \langle c_0, c_1, \ldots, c_{k+1} \rangle \ ,$$

with $c_0 = 0$ and $c_1 = 1$ is called a strong addition chain if, for all $1 < i \leq k+1$ there exist j, j', and i', $0 \leq j, j', i' < i$, such that $c_i = c_j + c_{j'}$ and $c_{i'} = c_j - c_{j'}$.

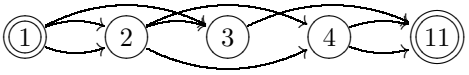
The length of the chain C is defined to be $\ell_L(C) = k$. The length of the shortest strong addition chain containing the set $S \subseteq \mathbb{Z}_{>0}$ is denoted $\ell_L(S)$.

In the previous chapter several addition chains for 11 were discussed. Neither of these chains is a strong addition chain. In the chain $\langle 1, 2, 3, 4, 8, 11 \rangle$ for instance the only option to construct 11 is by adding 8 and 3, but unfortunately the difference $8 - 3 = 5$ is not available. The chain $\langle 1, 2, 3, 4, 7, 11 \rangle$ is a strong addition chain. In

the graph corresponding to the strong addition chain, dotted lines are used to make clear which differences are needed.



Considered as ordinary addition chains $\langle 1,2,3,4,7,11\rangle$ and $\langle 1,2,3,4,8,11\rangle$ are equivalent because they have the same condensed chain $\langle 1,2,4,11\rangle$. This equivalence does not carry over to strong addition chains, since only one of them is a strong addition chain. The problem is the loss of associativity for strong addition chains. If we take another look at their common partially condensed graph,



then we see that there are three arrows pointing to 11. In the present case of strong addition chains, 11 may be computed as $(3+4)+4$, but not as $3+(4+4)$. Different orders of computation may alter the set of differences. These differences might or might not be in the chain.

Strictly speaking, 11 cannot be computed as $(3+4)+4$ either, since calculating $3+4$ requires the difference -1, which is negative (and therefore not in the chain). Instead, $4+3$ must be computed. This swapping trick always works to get rid of possibly occurring negative differences.

For the applications we have in mind, inversion will typically be free and there is no need to swap (in a way, commutativity is restored). In this case it makes sense to allow subtraction steps as well (where the sum already has to be in the chain). Allowing subtractions also gives a smoother transition to the third degree Perrin chains (Definition 3.6).

Definition 3.2 (Lucas Chain). A sequence of nonnegative integers

$$C = \langle c_0, c_1, \ldots, c_{k+1}\rangle \ ,$$

with $c_0 = 0$ and $c_1 = 1$ is called a Lucas chain if, after extending it to negative coefficients by defining $c_{-i} = -c_i$ for all $0 < i < k+1$, for all $0 < i \leq k+1$ there exist j, j', and i' all of absolute value smaller than i, such that $c_i = c_j + c_{j'}$ and $c_{i'} = c_j - c_{j'}$.

The length of C is defined to be $L_2(C) = k$. The length of the shortest Lucas chain containing the set $S \subseteq \mathbb{Z}$ is denoted $L_2(S)$.

Sometimes a double exponentiation has to be performed during a cryptographic protocol. In the present case of second order recurrences, let v_κ and v_λ be two bases and n and m two exponents, then a double exponentiation requires computing $v_{\kappa n+\lambda m}$. If κ and λ were known, this would in fact be a single exponentiation, but in general κ and λ are not known. However, if for instance $n = m = 1$, the combination of v_κ and v_λ requires the 'quotient' $v_{\kappa-\lambda}$ or $v_{\lambda-\kappa}$ (in this case we can

use an incomplete chain for $\kappa+\lambda$ by supposing that $c_j = \kappa, c_{j'} = \lambda$, and $c_{i'} = \kappa - \lambda$ after which $c_i = \kappa + \lambda$ can be added to the chain). Therefore, in order to be able to perform a double exponentiation with the bases v_κ and v_λ at least one auxiliary base is needed.

It turns out that the number of auxiliary bases to perform a multi-exponentiation given m bases, requires $2^m - m - 1$ auxiliary bases. This can be proven by a concept shown to us by P. Beelen. In accordance with Definition 2.2, vectors of length m are used to model a multi-exponentiation. If we momentarily fall back to the notation of group exponentiation of the previous chapter, we could say that $\mathbf{g}^{\mathbf{x}}$ and $\mathbf{g}^{\mathbf{y}}$ can only be multiplied if the quotient $\mathbf{g}^{\mathbf{x}-\mathbf{y}}$ is known. The result of the multiplication is $g^{\mathbf{x}+\mathbf{y}}$. Suppose several elements $\mathbf{g}^{\mathbf{x}}$ are known, for $\mathbf{x} \in S$. We ask ourselves the question which other elements can be constructed under multiplication, bearing in mind we need to know the quotient of two elements we want to multiply. The identity element (corresponding to the all-zero vector) and inversion are for free. By concentrating on the exponents, we get the following definition.

DEFINITION 3.3 (Lucas set). A subset $\Lambda \subseteq \mathbb{Z}^m$ is called a Lucas set iff the following holds:

(1) $0 \in \Lambda$
(2) $\mathbf{x} \in \Lambda \Rightarrow -\mathbf{x} \in \Lambda$
(3) $\mathbf{x}, \mathbf{y}, \mathbf{x}-\mathbf{y} \in \Lambda \Rightarrow \mathbf{x}+\mathbf{y} \in \Lambda$

Given a set $S \subseteq \mathbb{Z}^m$ the Lucas-closure of S is defined as the intersection of all Lucas sets (in $\mathbb{Z}^m$) containing S.

For multi-exponentiation the bases should not depend on the exponents. In other words, given m bases, it should be possible to perform a multi-exponentiation given any set of m exponents. Considered as vectors, the bases together with the auxiliary bases should span $\mathbb{Z}^m$. (The use of some negative exponents is hard to avoid entirely, since the auxiliary vectors will have negative entries).

THEOREM 3.4. *The smallest subset $S \subseteq \mathbb{Z}^m$ whose Lucas closure equals $\mathbb{Z}^m$ has cardinality $2^m - 1$.*

Proof: Let $S \subseteq \mathbb{Z}^m$ have Lucas closure $\mathbb{Z}^m$. Consider the closure rules modulo 2. Since $-\mathbf{x} \equiv \mathbf{x} \mod 2$ and $\mathbf{x}-\mathbf{y} \equiv \mathbf{x}+\mathbf{y} \mod 2$ it follows that S modulo 2 should equal $\mathbb{Z}^m$ modulo two, apart from the all zero vector which comes for free in the closure, and hence should have at least $2^m - 1$ elements.

The set S consisting of all $(0,1)$-vectors of length m without the all zero-vector has cardinality $2^m - 1$ and has Lucas closure $\mathbb{Z}^m$. The latter can be proven by induction. Let $\mathbf{z}$ be a vector all of whose entries are in $\{-1, 0, 1\}$. If for some $0 < i \leq m$ the entry $z_i = -1$ define $x_i = 0$ and $y_i = -1$, so $x_i - y_i = 1$; if $z_i = 1$ define $x_i = 1$ and $y_i = 0$, so $x_i - y_i = 1$; finally, if $z_i = 0$, define $x_i = y_i = 0$ as well. In all three cases $z_i = x_i + y_i$ and if the entire vectors are considered, $\mathbf{x}, -\mathbf{y}, \mathbf{x}-\mathbf{y}$ are all $(0,1)$-vectors, hence $\mathbf{z}$ is in the Lucas closure of S.

Suppose that for some $n \leq 1$ the Lucas closure of S contains all vectors in $\mathbb{Z}^m$ with all entries in absolute value smaller than or equal to n (we just saw this

statement is true for $n = 1$). Let $\mathbf{z}$ be a vector with all entries in $\{-(n+1), \ldots, (n+1)\}$. It can be written as $\mathbf{z} = \mathbf{x} + \mathbf{y}$ where $\mathbf{x}$'s entries are in $\{-n, \ldots, n\}$ and $\mathbf{y}$ is a $(-1, 0, 1)$-vector. The entries of the difference $\mathbf{x} - \mathbf{y}$ differ at most one from the corresponding entry in $\mathbf{x}$, but they can never be either $-(n+1)$ or $n+1$. Hence, by the induction hypothesis, all three vectors $\mathbf{x}, \mathbf{y}, \mathbf{x} - \mathbf{y}$ are already in the closure and hence by the third rule, $\mathbf{z}$ is as well. *Q.E.D.*

Although the above theorem does not exclude Lucas chains for vectors, it does make them less appealing for large m. For ordinary addition chains, no auxiliary bases were needed and for any fixed m a multi-exponentiation asymptotically costs the same as a single exponentiation. For Lucas chains, the number of auxiliary bases is exponential in m. It is an interesting open question whether the exponential number of initial vectors results in chains of exponential length as well, but even for fixed m the costs are asymptotically higher than that of a single exponentiation. Hence, we will restrict ourselves to the most relevant cases for cryptography: $m = 1$ corresponding to single exponentiation and $m = 2$ corresponding to double exponentiation.

Definition 3.5 (Vectorial Lucas Chain). A sequence of vectors in $\mathbb{Z}^2$

$$C = \langle c_0, c_1, \ldots, c_{k+3} \rangle \ ,$$

with $c_0 = (0,0)^T, c_1 = (1,0)^T, c_2 = (0,1)^T$, and $c_3 = (1,-1)^T$, is called a Lucas chain iff, after extending it to negative coefficients by defining $c_{-i} = -c_i$ for all $0 < i < k+3$, for all $0 < i \leq k+3$ there exist j, j', and i' all of absolute value smaller than i, such that $c_i = c_j + c_{j'}$ and $c_{i'} = c_j - c_{j'}$.

The length of C is defined to be $L_2(C) = k$. The length of the shortest Lucas chain containing the set $S \subseteq \mathbb{Z}^2$ is denoted $L_2(S)$.

3.1.2. Perrin Chains. Efficient computation of a third order recurrence based on $c_{n+m} = \alpha(c_n, c_m, c_{n-m}, c_{n-2m})$ calls for an even stronger version of addition chain. We call these Perrin chains, after the third order Perrin recurrence (3) on page 11. It is assumed that inversion (i.e. computing c_{-n} given c_n) is cheap, similar to addition-subtraction and Lucas chains. For the only application in this thesis, XTR in Section 4.6, this is the case. Allowing negative exponents is advantageous if $n-2m$ is negative but $n-m$ is positive. Unlike in the second order case, swapping n and m will still give one negative and one positive term. If inversion is well-defined but hard to compute, the initial value c_{-1} can be used to extend the Perrin chain to negative exponents nevertheless: whenever c_{n+m} is added to the chain, also put $c_{-(n+m)}$ in the chain based on $c_{-(n+m)} = \alpha(c_{-n}, c_{-m}, c_{m-n}, c_{m-2n})$. Using induction it follows that this always works. This method doubles, or almost doubles, the costs and is not necessarily optimal, even when the Perrin chain used is.

Definition 3.6 (Perrin Chain). A sequence of nonnegative integers

$$C = \langle c_0, c_1, \ldots, c_{k+1} \rangle \ ,$$

with $c_0 = 0$ and $c_1 = 1$ is called a Perrin chain if, after extending it to negative coefficients by defining $c_{-i} = -c_i$ for all $0 < i < k+1$, for all $0 < i \leq k+1$ there exist

j, j', i' and i'' all of absolute value smaller than i, such that $c_i = c_j + c_{j'}, c_{i'} = c_j - c_{j'}$, and $c_{i''} = c_j - 2c_{j'}$.

The length of C is defined to be $L_3(C) = k$. The length of the shortest Perrin chain containing the set $S \subseteq \mathbb{Z}$ is denoted $L_3(S)$.

For double exponentiation, a mixture of Definition 3.5 and 3.6 can be used with additional initial vector $c_4 = (1, -2)^T$. The number of required auxiliary bases for multi-exponentiation based on Perrin chains increases even more compared to Lucas chains, as demonstrated by the following theorem kindly shown to us by P. Beelen [**11**]. Perrin sets are the third order analogue of Lucas sets.

DEFINITION 3.7 (Perrin set). A subset $\Lambda \subseteq \mathbb{Z}^n$ is called a Perrin set iff the following three hold:

(1) $0 \in \Lambda$
(2) $\mathbf{x} \in \Lambda \Rightarrow -\mathbf{x} \in \Lambda$
(3) $\mathbf{x} - \mathbf{y}, \mathbf{x}, \mathbf{y}, \mathbf{x} - 2\mathbf{y} \in \Lambda \Rightarrow \mathbf{x} + \mathbf{y} \in \Lambda$

THEOREM 3.8 (P. Beelen). *Let $S \subseteq \mathbb{Z}^m$ have Perrin closure $\mathbb{Z}^m$, then the cardinality of S is at least $\frac{3^m-1}{2}$.*

Proof: Let $S \subseteq \mathbb{Z}^m$ have Perrin closure $\mathbb{Z}^m$. Consider the closure rules modulo 3. The negation rule doubles the number of nonzero elements, but the addition rule does nothing. Hence each element in S will give rise to at most two different congruency classes in $\mathbb{Z}_3^m$. Since there are 3^m of these classes and 0 comes for free, any set with Perrin closure $\mathbb{Z}^m$ has at least $\frac{3^m-1}{2}$ elements.

Let S be the set consisting of all $(-1, 0, 1)$-vectors of length m for which the first nonzero entry is positive (so without the all zero-vector). The set S has cardinality $\frac{3^m-1}{2}$. Rules (1) and (2) imply that all $(-1, 0, 1)$-vectors are in the closure. With induction similar to the proof of Theorem 3.4 it follows that the Perrin closure is indeed $\mathbb{Z}^m$. *Q.E.D.*

3.1.3. Notation. The use of v is reserved for second order and that of c for third order recurrences. Similar to the case of exponentiation in a group, we will also use α to denote the cost of applying α. The special case of computing v_{2n} or c_{2n} from v_n respectively c_n corresponds to squaring in a group. Its costs will be denoted similarly by δ. Computing c_{-n} from c_n corresponds to inversion and will be denoted by ν, but the number of inversions will not be counted (either it is very cheap or it is too expensive).

Lucas chains are used in Section 4.4 for LUC and in Chapter 5 for the Montgomery representation of elliptic curves. LUC is linear and the application of δ costs about 80% of that of α under Assumption 1.3. For the Montgomery representation a refinement is made. Since projective coordinates are used, having a fixed operand speeds up the application of α. A distinction needs to be made which of the three operands of α is fixed. We will use $\dot{\alpha}_i$ to denote these cheaper calls to the recurrence relation where the i-th argument is fixed. For curves defined over binary

fields, applying δ costs about 40% of applying α and applying $\dot{\alpha}_3$ costs about 80% of applying α.

Perrin chains are used in Section 4.6 for XTR. XTR has the somewhat unusual property that computing both c_{3n} and c_{2n} given c_n costs less than $\delta + \alpha$. For this reason triplings of this sort will be counted separately for Perrin chains, using the symbol τ. (Occasionally τ will be used for Lucas chains as well, but for the runtimes $\tau = \delta + \alpha$ is substituted.)

3.2. Binary Algorithms

3.2.1. Single Lucas. The binary Algorithms 2.7 and 2.9 for addition chains can easily be adapted to produce strong addition chains by appropriate strengthening of the invariant. Possibly as a result of this simple derivation [**118**], the algorithms have been reinvented several times, which makes it hard to give proper credit. Most popular is the left-to-right variant [**101, 128, 156, 171, 203**]. The right-to-left version seems to appear in the literature slightly later [**128, 203**]. Both iterative and recursive versions exist, we limit ourselves to the iterative version. The numerical behaviour of the algorithms when applied to the evaluation of Chebyshev polynomials is studied by Koepf [**95**]; the relevance of the algorithm for computer algebra packages is discussed by Fateman [**61**].

Algorithm 3.9 (Second Degree Left-to-Right Binary Exponentiation). This algorithm takes as input a base v and a positive exponent n and outputs v_n and v_{n+1}. It has invariant

$$0 \le j < k, \quad a = \sum_{i=j}^{k-1} n_i 2^{i-j}, \quad A = v_a, \quad B = v_{a+1} \ .$$

1. [Initialization] Set $j \leftarrow k-1, a \leftarrow 1, A \leftarrow v$, and $B \leftarrow \delta(v)$.
2. [Finished?] If $j = 0$ terminate with output A and B.
3. [Decrease j] If $n_{j-1} = 0$, set $a \leftarrow 2a$ as well as $B \leftarrow \alpha(B, A, v)$ and $A \leftarrow \delta(A)$. Otherwise, set $a \leftarrow 2a+1, A \leftarrow \alpha(B, A, v)$ and $B \leftarrow \delta(B)$. Decrease j by one.

The left-to-right algorithm has the advantage that the difference is fixed, namely the base v. This can be advantageous for instance when using the Montgomery representation for elliptic curves [**130**] where precomputation on v makes the computation of $\alpha(B, A, v)$ cheaper than that of $\alpha(B, A, C)$ for general C (see also Chapter 5 for further details).

Lemma 3.10. *Given a k-bit exponent n, Algorithm 3.9 takes $\dot{\alpha}_3(k-1) + \delta k$ operations.*

Because the algorithm returns both v_n and v_{n+1}, it can sometimes be advantageous to use exponent $n' = n - 1$, thus obtaining v_n and v_{n-1}. Moreover, if only v_n is required, there are some minor improvements possible. First of all, there is no need to compute B in the final step ($j = 1$). Secondly, if the exponent n is even, it clearly pays off to divide n by two and at the same time doubling the base v

(repeatedly until n is odd). Dividing out other primes will give a small decrease in the number of α and δ needed [**197**]. The overhead (checking divisibility by some odd prime), the lower probability that p divides n and the reduced gain if it does, make this option less attractive.

The fact that the algorithm produces v_{n+1} (or alternatively v_{n-1}) as a byproduct is beneficial when used for precomputation (Section 3.3.4) and also eases the recovery of y-coordinates for elliptic curves when the Montgomery representation is used (Section 5.3.1).

The use of Algorithm 3.9 has also been proposed as a possible countermeasure against timing and power analyses. Unlike the standard binary algorithm, Algorithm 3.9 performs the same operations per step independent of the binary representation of the exponent. Whether the bit being read is zero or one, the algorithm always performs a squaring (δ) and a multiplication (α), thus defeating timing attacks. There are some subtleties though [**34, 76, 81**].

ALGORITHM 3.11 (Second Degree Right-to-Left Binary Exponentiation). This algorithm takes input v and n and outputs v_n. It has invariant

$$0 \le j \le k, \quad a = \sum_{i=0}^{j-1} n_i 2^i, \quad A = v_a, \quad B = v_{2^j}, \quad C = v_{2^j - a} \ .$$

1. [Initialization] Set $j \leftarrow 0, a \leftarrow 0, A \leftarrow id, B \leftarrow v$, and $C \leftarrow v$.
2. [Increase j] If $n_j = 0$, set $C \leftarrow \alpha(B, C, A)$ else set $a \leftarrow a + 2^j$ and $A \leftarrow \alpha(B, A, C)$. Set $B \leftarrow \delta(B)$, and increase j by one.
3. [Is $j = k$?] If $j < k$ go back to step 2, otherwise terminate with output A.

LEMMA 3.12. *Given a k-bit exponent n, Algorithm 3.9 takes time $(\alpha + \delta)k$.*

As noted by Montgomery [**128**], the right-to-left algorithm allows a (not very impressive) precomputation trade-off, since the values v_{2^j} can be computed in advance for $j = 1, \ldots, k-1$. Each precomputed point saves one application of δ. Moreover, it can also be assumed that the B are fixed in this case, which gives us:

LEMMA 3.13. *Given a k-bit exponent n and precomputed values v_{2^j} for $j = 1, \ldots, k-1$, then (a slight adaptation of) Algorithm 3.11 takes time $\dot{\alpha}_1 k$.*

A slight change in the argumentation shows that a twofold exponentiation can be performed in time $(\delta + 2\alpha)k$ using the right-to-left algorithm.

3.2.2. Perrin Versions. Adams and Shanks [**1**] give an algorithm for efficient computation of the Perrin sequence (3) on page 11, based on the doubling formula $A_{2n} = A_n^2 - 2A_{-n}$. Their method is somewhat involved and seems to depend on the property of the Perrin sequence that A_{n+2} is missing in the recurrence (3), which enables computation of the odd term A_n based on the even terms A_{n+1} and A_{n+3}. Since there is no easy way to derive A_{-n} given A_n, the algorithm has to compute the negative exponents explicitly as well. The overall costs for computation given a k-bit exponent are $6k$ applications of δ and $4k$ relatively cheap applications of (3).

The left-to-right binary algorithm has been described for third order recurrences by Gong and Harn [**68**], and by Lenstra and Verheul [**107**]. The right-to-left method does not seem to appear in the literature, but its derivation is straightforward.

Gong and Harn [**68**] use as invariant $a = \sum_{i=j}^{k-1} n_i 2^{i-j}$, where $0 \le j \le k$. They also keep track of $A = c_a, B = c_{a+1}$, and $C = c_{a-1}$. Initialization with $j \leftarrow k, a \leftarrow 0, A \leftarrow id, B \leftarrow c$, and $C \leftarrow \nu(c)$ is easy. Decreasing j by one requires setting $a \leftarrow 2a + n_{j-1}$. If $n_{j-1} = 0$, this boils down to $A \leftarrow \delta(A), B \leftarrow \alpha(B, A, c, \nu(C))$, and $C \leftarrow \alpha(A, C, c, \nu(B))$, all at once; for $n_{j-1} = 1$ it means simultaneously setting $A \leftarrow \alpha(B, A, c, \nu(C)), B \leftarrow \delta(B)$, and $C \leftarrow \delta(A)$. A single step therefore costs $\dot{\alpha}_3 + 2\delta$ if the bit n_{j-1} is set and $2\dot{\alpha}_3 + \delta$ otherwise. On average, for a k-bit exponent, this yields $\frac{3}{2}(\dot{\alpha}_3 + \delta)k$ operations. Gong et al. [**69**] later give an improvement, based on something called the 'maximum weight signed-digit representation'. The resulting algorithm can be shown to be an elaborate equivalent of Lenstra and Verheul's algorithm (Algorithm 3.14 below).

Lenstra and Verheul [**107**] use a slightly different invariant, thereby getting rid of the asymmetry between the two steps, requiring $\alpha + 2\delta$ regardless whether the bit n_j is set or not. This has two advantages. Most importantly we can expect Lenstra and Verheul's method to be faster than Gong and Harn's, since the cost for performing δ is generally lower than that for α. Secondly, the uniformity of the steps will help thwarting timing and power analyses.

ALGORITHM 3.14 (Third Degree Left-to-Right Binary Exponentiation). On input n and c this algorithm returns c_n. Alternatively, it can output the triple $(c_{2\lfloor \frac{n}{2} \rfloor}, c_{2\lfloor \frac{n}{2} \rfloor + 1}, c_{2\lfloor \frac{n}{2} \rfloor + 2})$. It maintains as invariant

$$0 \le j < k, \quad a = 1 + \sum_{i=j+1}^{k-1} n_i 2^{i-j}, \quad A = c_a, \quad B = c_{a+1}, \quad C = c_{a-1} \ .$$

1. [Initialization] Set $j \leftarrow k-1, a \leftarrow 1$ as well as $A \leftarrow c, B \leftarrow \delta(A)$, and $C \leftarrow id$.

2. [Finished?] If $j = 0$ terminate with output C if $n_0 = 0$ and with output A otherwise.

3. [Decrease j] If $n_j = 0$, set $a \leftarrow 2a - 1$ and replace the triple (A, B, C) by $(\alpha(C, A, \nu(c), \nu(B)), \delta(A), \delta(C))$; else $(n_j = 1)$ set $a \leftarrow 2a + 1$ and simultaneously $A \leftarrow \alpha(B, A, c, \nu(C)), B \leftarrow \delta(B)$, and $C \leftarrow \delta(A)$. Decrease j by one and go back to the previous step.

Note that in principle one could benefit from the fact that some of the arguments are fixed throughout the protocols, as in Algorithm 3.9. For XTR this does not seem to be of any consequence and we are not aware of any other computation where it would be. For efficiency reasons it might be advantageous to precompute $\nu(c)$ once at the beginning.

As discussed earlier the use of negations (or inversions) seems inevitable when using third degree recurrences.

LEMMA 3.15 (Lenstra and Verheul). *Given a k-bit exponent n, Algorithm 3.14 takes time $(\dot{\alpha}_3 + 2\delta)k$.*

It is also possible to use a right-to-left algorithm. It only returns c_n and not its neighbours. It has almost the same runtime as Algorithm 3.14. Like the right-to-left Lucas algorithm, some modest speedup is possible by precomputing all values c_{2^j} for $0 < j < k$. The resulting runtime (after precomputation) is reduced to $(\dot{\alpha}_1 + \delta)k$. The adaptation for twofold exponentiation requires $(2\alpha + 3\delta)k$ for two k-bit exponents.

ALGORITHM 3.16 (Third Degree Right-to-Left Binary Exponentiation). On input a base c and a k-bit exponent n this algorithm returns c_n. It uses as invariant

$$0 \le j \le k, \quad a = \sum_{i=0}^{j-1} n_i 2^i, \quad b = 2^j, \quad A = c_a, \quad B = c_b, \quad C = c_{b-a}, \quad D = c_{b-2a} \ .$$

1. [Initialization] Set $j \leftarrow 0, a \leftarrow 0, b \leftarrow 1, A \leftarrow id$, and $B, C, D \leftarrow g$.
2. [Finished?] If $j = k$ terminate with output A.
3. [Read bit n_j] If $n_j = 0$, simultaneously set $C \leftarrow \alpha(B, C, A, \nu(D))$ and $D \leftarrow \delta(C)$; otherwise ($n_j = 1$), set $a \leftarrow a+b$ and simultaneously $A \leftarrow \alpha(B, A, C, D)$ and $D \leftarrow \delta(\nu(A))$.
4. [Increase j] Set $b \leftarrow 2b$ and $B \leftarrow \delta(B)$. Increase j by one and go to step 2.

3.2.3. Double Exponentiation. For ordinary addition chains Shamir's trick can be used to speed up multi-exponentiation. For double exponentiation this technique can also be exploited for Lucas chains, as shown by Schoenmakers [**163**]. To ease the exposition an auxiliary variable a is maintained, which is not actually needed when the algorithm is implemented. Therefore the discrete logarithms κ and λ need not be known in practice.

ALGORITHM 3.17 (Schoenmakers' Second Degree Double Exponentiation). This algorithm takes input bases v_κ, v_λ, and auxiliary $v_{\kappa-\lambda}$ and k-bit exponents n and m. It outputs $v_{n\kappa+m\lambda}$. The following invariant is maintained:

$$0 \le j \le k, \quad a = \sum_{i=j}^{k-1} (n_i\kappa + m_i\lambda)2^{i-j}, \quad A = v_a, \quad B = v_{a+\kappa}, \quad C = v_{a+\lambda} \ .$$

1. [Initialization] Set $j \leftarrow k, a \leftarrow 0, A \leftarrow id, B \leftarrow v_\kappa$ and $C \leftarrow v_\lambda$.
2. [Decrease j] Make a distinction between four cases.
i. If $n_{j-1} = 0$ and $m_{j-1} = 0$, set $a \leftarrow 2a$ and the corresponding $(A, B, C) \leftarrow (\delta(A), \alpha(B, A, v_\kappa), \alpha(C, A, v_\lambda))$.
ii. If $n_{j-1} = 1$ and $m_{j-1} = 0$, set $a \leftarrow 2a + \kappa$ as well as $(A, B, C) \leftarrow (\alpha(B, A, v_\kappa), \delta(B), \alpha(B, C, v_{\kappa-\lambda}))$.
iii. If $n_{j-1} = 0$ and $m_{j-1} = 1$, set $a \leftarrow 2a + \lambda$ as well as $(A, B, C) \leftarrow (\alpha(C, A, v_\lambda), \alpha(B, C, v_{\kappa-\lambda}), \delta(C))$.
iv. Finally, if both $n_{j-1} = 1$ and $m_{j-1} = 1$, set $a \leftarrow 2a + \kappa + \lambda$, and set $(A, B, C) \leftarrow (\alpha(B, C, v_{\kappa-\lambda}), \alpha(B, A, v_\kappa), \alpha(C, A, v_\lambda))$ followed by $(B, C) \leftarrow (\alpha(A, v_\kappa, B), \alpha(A, v_\lambda, C))$.

In all cases, decrease j by one.

3. [Finished?] If $j > 0$ go back to step 2, otherwise terminate with output A.

Unless both bits are set, a step will cost one point doubling and two fixed difference additions. If both bits are set a step will cost three fixed difference additions and two mixed additions.

LEMMA 3.18 (Schoenmakers). *Given k-bit exponents n and m, Algorithm 3.17 will on average cost $(\frac{1}{2}\dot{\alpha}_2 + 2\frac{1}{4}\dot{\alpha}_3 + \frac{3}{4}\delta)k$.*

An improvement to the above is possible, due to Akishita [**5**]. By doing some lookahead, the cost of the most expensive step is reduced to three fixed difference additions. The resulting runtime is summarized in Lemma 3.19.

LEMMA 3.19 (Akishita). *Given input bases v_κ, v_λ, and auxiliary $v_{\kappa-\lambda}$ and k-bit exponents n and m, computing $v_{n\kappa+m\lambda}$ will cost at most $(2\frac{1}{4}\dot{\alpha}_3 + \frac{3}{4}\delta)k$.*

3.2.4. Binary Perrin Double Exponentiation. Adapting either Schoenmakers' or Akishita's algorithm to third degree is troublesome. Lenstra and Verheul [**107**, Algorithm 2.4.8] give a matrix-based method for double exponentiation costing two (binary) single exponentiations. The method only works for linear recurrences (this follows from its derivation, which is based on the observation contained in (6) on page 12). Below is an improved version that get rids of the matrices, thereby giving a minor speedup and a cleaner, more elegant algorithm. The algorithm is based on a closer examination of Algorithm 3.14. The recurrence does not have to be linear anymore, but the method does require that the exponent can be reduced modulo an odd prime, say q.

Let $c_{\kappa-1}, c_\kappa$, and $c_{\kappa+1}$ be given (for possibly unknown κ). Consider running Algorithm 3.14 on k-bit input n with a modified initialization step, namely setting $A \leftarrow c_{2\kappa+1}, B \leftarrow c_{2\kappa+2}, C \leftarrow c_{2\kappa}$, corresponding to $a \leftarrow 2\kappa + 1$. The initialization $a \leftarrow 2\kappa + 1$ is used instead of $a \leftarrow \kappa$ to ensure that a is initialized to an odd value. Computing the desired values for A, B, and C from the given c's is straightforward. The invariant then contains $a = 1 + \sum_{i=j+1}^{k} n_i 2^{i-j}$ where we artificially set $n_k = \kappa$ (and not bother that most likely $\kappa \notin \{0, 1\}$). From the new invariant it follows that the algorithm will terminate with $a = 2^k\kappa + n + 1 - n_0$.

This can be used for a double exponentiation routine. To compute $c_{n\kappa+m}$, recode the exponent as

$$n\kappa + m \equiv \frac{n}{2^k}(2^k\kappa + \frac{m2^k}{n}) \pmod{q},$$

compute $\tilde{c} = c_{2^k\kappa + \frac{m2^k}{n}}$ using the tweaked single exponentiation routine just described and then $\tilde{c}_{n'}$ with an ordinary single exponentiation routine where $n' = \frac{n}{2^k}$. The result is the desired double exponentiation. The double exponentiation method is described in detail below.

ALGORITHM 3.20 (Matrix-less Third Degree Double Exponentiation).
Let m and n be integers with $0 < m, n < q$, and let $c, c_\kappa, c_{\kappa-1}$, and $c_{\kappa+1}$ be given. This algorithm computes $c_{n\kappa+m}$ provided that $c_q = id$.

1. [Exponent transformation] Set $k \leftarrow \lfloor \lg q \rfloor, n' \leftarrow n/2^k \bmod q$ and $m' \leftarrow m/n' \bmod q$.
2. [Initialization] Set $j \leftarrow k-1, a \leftarrow 2\kappa+1, A \leftarrow \alpha(c_{\kappa+1}, c_\kappa, c, \nu(c_{\kappa-1})), B \leftarrow \delta(c_{\kappa+1})$, and $C \leftarrow \delta(c_\kappa)$.
3. [Finished with $c_{2^k\kappa+m'}$?] If $j = 0$ go to step 5 after setting $\tilde{c} \leftarrow C$ if m' is odd, and $\tilde{c} \leftarrow A$ otherwise.
4. [Read bit m'_j] If $m'_j = 0$, set $a \leftarrow 2a-1$ and replace the triple (A, B, C) by $(\alpha(C, A, \nu(c), \nu(B)), \delta(A), \delta(C))$; else $(m'_j = 1)$ set $a \leftarrow 2a+1$ and simultaneously $A \leftarrow \alpha(B, A, c, \nu(C)), B \leftarrow \delta(B)$, and $C \leftarrow \delta(A)$.
5. [Decrease j] Decrease j by one, go back to step 3.
6. [Compute $\tilde{c}_{n'}$] Compute $\tilde{c}_{n'}$ using any single exponentiation routine on input $\tilde{c}$ and n'. Output the result.

The runtime of the above algorithm depends on the time taken by the single exponentiation in the final step.

LEMMA 3.21. *Algorithm 3.20 takes about* $(4\delta + 2\alpha)\lfloor \lg q \rfloor$ *assuming that in the final step one of the binary algorithms (3.14 or 3.16) is invoked.*

This is a small constant number of operations better than [**107**, Algorithm 2.4.8]. For realistic choices of q the speedup achieved using Algorithm 3.20 is thus barely noticeable. Nevertheless, it is a significant result because the fact that the matrices as required for [**107**, Algorithm 2.4.8] are no longer needed, facilitates implementation of XTR. A more substantial improvement over the double exponentiation methods just described that neither requires matrices nor a prime group order, can be achieved using a Euclidean algorithm.

3.3. Montgomery's Euclidean Algorithms

Independent from the development of Euclidean methods for ordinary addition chains, Montgomery [**128**] discovered related methods for Lucas chains as improvement over the binary (and binary-ternary) method.

ALGORITHM 3.22 (Montgomery's CFRC Algorithm).
On input a base v, an exponent n and an auxiliary 'exponent' m satisfying $0 < m < n$ and $\gcd(n, m) = 1$ this algorithm outputs v_n. It keeps invariant

$$0 \le e, \quad 0 < d, \quad \gcd(d, e) = 1, \quad ad + be = n, \quad A = v_a, \quad B = v_b, \quad C = v_{a-b} \ .$$

1. [Initialize] Set $(d, e) \leftarrow (n-m, m), (a, b) \leftarrow (1, 1)$ and $(A, B, C) \leftarrow (v, v, id)$.
2. [Finished?] If $e = 0$, terminate with output A.
3. [Decrease (d, e)] If $d > e$ set $b \leftarrow a+b, d \leftarrow d-e$, as well as $(B, C) \leftarrow (\alpha(A, B, C), \nu(B))$; else $(d \le e)$ set $a \leftarrow a+b, e \leftarrow e-d$ and $(A, C) \leftarrow (\alpha(A, B, C), A)$. Go back to the previous step.

Montgomery gives a modification to get rid of the computation of $\nu(B)$ in step 3 if $d > e$. The adaptation to double exponentiation also follows from Montgomery's work [**128, 131**]. The algorithm derives its name CFRC from continued fractions.

Interestingly, Montgomery's CFRC algorithm is based on the subtractive Euclidean algorithm and not the classical Euclidean algorithm, like Algorithm 2.24. The subtractive Euclidean algorithm has a runtime of $O((\log n)^2)$ operations if both n and $m < n$ are chosen at random [**201**]. This is much worse than the binary algorithm's $O(\log n)$. Before discussing Montgomery's solution to this problem, let us investigate what happens if we try to use the classical Euclidean algorithm for Lucas chains.

Recall that in Algorithm 2.24 the pair (d, e) is always substituted with $(e, d \bmod e)$. Since $d \mod e = d - \lfloor \frac{d}{e} \rfloor e$, this can be regarded as a linear transformation

$$S = \begin{pmatrix} 0 & 1 \\ 1 & -\lfloor \frac{d}{e} \rfloor \end{pmatrix} ,$$

so the substitution involving d and e can be rewritten as $(d, e) \leftarrow (d, e)S$. (Of course if S is allowed to depend on d and e anything can be regarded as a linear transformation as long as $(d, e) \neq (0, 0)$.) Let S' be the 2×2 matrix describing the (a, b)-substitution where $(a, b) \leftarrow (a, b)S'$. In the present case

$$S' = \begin{pmatrix} \lfloor \frac{d}{e} \rfloor & 1 \\ 1 & 0 \end{pmatrix} .$$

For exponentiation in a group and writing $\mathbf{A} = (A, B)$ this leads to $\mathbf{A} \leftarrow \mathbf{A}^{S'}$ nicely corresponding to $(A, B) \leftarrow (A^{\lfloor \frac{d}{e} \rfloor} B, A)$. Two problems emerge when dealing with Lucas chains.

The "quotient" $C = v_{a-b}$ has to be computed as well. The auxiliary value, $a - b$, will equal the first minus the second column of S'. We can make this specific by writing

$$S' = \left(\begin{array}{cc|c} \lfloor \frac{d}{e} \rfloor & 1 & \lfloor \frac{d}{e} \rfloor - 1 \\ 1 & 0 & 1 \end{array} \right) .$$

The second problem is computing $v_{a\lfloor \frac{d}{e} \rfloor + b}$ and $v_{a\lfloor \frac{d}{e} \rfloor + b - a}$. Using a single exponentiation routine to determine $v_{a\lfloor \frac{d}{e} \rfloor}$ is of no use, since the result cannot be combined with v_b without knowing $v_{a\lfloor \frac{d}{e} \rfloor - b}$. The subtractive algorithm exploits the fact that

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

If $k = \lfloor \frac{d}{e} \rfloor$ is large, this is too expensive. A much better way to implement the classical Euclidean step is based on an adaptation of Algorithm 3.11. For small values the subtractive method is faster.

Algorithm 3.23 (Classical Euclidean Step).
This algorithm takes as input bases v_κ, v_λ, auxiliary base $v_{\kappa-\lambda}$ and a k-bit exponent n. The algorithm outputs $v_{\kappa n+\lambda}$ and $v_{\kappa(n-1)+\lambda}$. It has invariant

$$0 \le j \le k, \quad b = 2^j \kappa, \quad a_1 = \lambda + \sum_{i=0}^{j-1} n_i 2^i \kappa, \quad a_2 = \lambda + \sum_{i=0}^{j-1} (n-1)_i 2^i \kappa,$$
$$A_1 = v_{a_1}, \quad B = v_b, \quad C_1 = v_{b-a_1}, \quad A_2 = v_{a_2}, \quad C_2 = v_{b-a_2} .$$

1. [Easy cases] If $n < 11$ use the subtractive Euclidean method.
2. [Initialization] Set $j \leftarrow 0, a_1, a_2 \leftarrow \lambda, A_1, A_2 \leftarrow v_\lambda, B \leftarrow v_\kappa$ and $C_1, C_2 \leftarrow v_{\kappa-\lambda}$.
3. [Read bit n_j] If $n_j = 0$, set $C_1 \leftarrow \alpha(B, C_1, A_1)$ else set $a_1 \leftarrow a_1 + 2^j$ and $A_1 \leftarrow \alpha(B, A_1, C_1)$.
4. [Read bit $(n-1)_j$] If $(n-1)_j = 0$, set $C_2 \leftarrow \alpha(B, C_2, A_2)$ else set $a_2 \leftarrow a_2 + 2^j$ and $A_2 \leftarrow \alpha(B, A_2, C_2)$.
5. [Square] (If $j < k - 1$) Set $b \leftarrow 2b, B \leftarrow \delta(B)$, and increase j by one.
6. [Finished?] If $j < k$ go back to step 3, otherwise terminate with output A_1 and A_2.

3.3.1. Double Exponentiation. We now return to Montgomery's solution. He obtains short Lucas chains by considering other Euclidean algorithms such as the binary Euclidean algorithm as well. As already remarked, the substitution $(d, e) \leftarrow (e, d \mod e)$ can be regarded as a linear transformation S. If we consider other substitutions for (d, e) this means other transformations S, so we can still write $(d, e) \leftarrow (d, e)S$ and $(a, b) \leftarrow (a, b)S'$ for some related transformation S'. In general, the transformation S is given by

$$S = \begin{pmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{pmatrix} . \tag{8}$$

To uphold the invariant, it is required that the inner product $((d, e)S, (a, b)S')$ is equal to the old inner product $ad + be$. Although the choice of S will be based on d and e, we will assume that a large number of pairs (d, e) lead to the same transformation S. The relation between the old and the new inner product should therefore hold for all a, b, d, e, implying that the adjoint (or transpose) of S' equals the inverse of S. In other words, we can write

$$S' = S^{-T} = \frac{1}{s_{11}s_{22} - s_{12}s_{21}} \begin{pmatrix} s_{22} & -s_{21} \\ -s_{12} & s_{11} \end{pmatrix} . \tag{9}$$

EXAMPLE 3.24. Consider the transformation $(d, e) \leftarrow ((d - e)/2, e)$ from the binary Euclidean algorithm. The corresponding matrices S and S' are:

$$S = \begin{pmatrix} \frac{1}{2} & 0 \\ -\frac{1}{2} & 1 \end{pmatrix}, \quad \text{and} \quad S' = \left(\begin{array}{cc|c} 2 & 1 & 1 \\ 0 & 1 & -1 \end{array}\right), \tag{10}$$

so we see that $(a, b) \leftarrow (2a, a + b)$. This step will therefore cost δ for computing the new A and α for computing the new B. Since C will be the same (corresponding to $a - b$, the third column in S'), the total costs for this step are $\alpha + \delta$.

The matrix S in the example contains rational entries. This is allowed, but the entries in S' must be integer, otherwise the substitutions for $A = v_a$ etc. cannot be performed.

We call a matrix $S \in \mathbb{Q}^{2\times 2}$ admissible for a pair of positive integers (d, e) if $S^{-1} \in \mathbb{Z}^{2\times 2}$, $(d, e)S \in \mathbb{Z}^2_{>0}$ and the greatest common divisor of the values obtained by $(d, e)S$ equals $\gcd(d, e)$.

ALGORITHM 3.25 (Montgomery's Euclidean Double Exponentiation). Given $v_\kappa, v_\lambda, v_{\kappa-\lambda}$ and integers n and m, this algorithm computes $u = \gcd(n, m)$ and $v_{(n\kappa+m\lambda)/u}$. Its invariant is

$$d > 0, \quad e \geq 0, \quad ad + be = n\kappa + m\lambda, \quad \gcd(d, e) = \gcd(n, m),$$
$$A = v_a, \quad B = v_b, \quad C = v_{a-b} \ .$$

1. [Initialize] Set $(d, e) \leftarrow (n, m)$. Moreover, set $(a, b) \leftarrow (\kappa, \lambda)$ and $(A, B, C) \leftarrow (v_\kappa, v_\lambda, v_{\kappa-\lambda})$.

2. [Finished?] If $e = 0$, terminate with output d and A.

3. [Decrease (d, e)] Pick an admissible matrix S for (d, e) (e.g., based on the first applicable rule in Table 3.1). Set $(d, e) \leftarrow (d, e)S$ and $(a, b) \leftarrow (a, b)S^{-T}$. Update A, B, and C accordingly, respecting the invariant. Go back to step 2.

The runtime strongly depends on the particular choices of S. As it is, one cannot even guarantee the algorithm to finish at all (for instance the identity-matrix is admissible). Table 3.1 gives the set of rules proposed by Montgomery. (The shorthand $\equiv_n$ is used for congruency modulo n.) The first applicable rule (from the top) will be picked by the algorithm. The first rule, M0, is simply a swap to ensure $d > e$ for the remaining steps. The rules M1, M2, M6, M7, and M8 are collectively known as the ternary rules and the rules M4, M5, and M9 as the binary rules. The remaining rule, M3, is the subtractive rule. If the subtractive rule is called with $e < d < 2e$ we speak of a Fibonacci step. If we refer to Algorithm 3.25 then Table 3.1 is used unless specified otherwise.

TABLE 3.1. Transformations proposed by Montgomery [**128**]

No.	Type	Condition	Substitution (d, e)	Costs
M0	(Swapping)	$d < e$	(e, d)	ν
M1	Ternary	$e < d \leq \frac{5}{4}e, d \equiv_3 -e$	$((2d - e)/3, (2e - d)/3)$	$3\alpha + \delta$
M2	Ternary	$e < d \leq \frac{5}{4}e, d \equiv_6 e$	$((d - e)/2, e)$	$\alpha + \delta$
M3	Subtractive	$d \leq 4e$	$(d - e, e)$	α
M4	Binary	$d \equiv_2 e$	$((d - e)/2, e)$	$\alpha + \delta$
M5	Binary	$d \equiv_2 0$	$(d/2, e)$	$\alpha + \delta$
M6	Ternary	$d \equiv_3 0$	$(d/3 - e, e)$	$3\alpha + \delta$
M7	Ternary	$d \equiv_3 -e$	$((d - 2e)/3, e)$	$3\alpha + \delta$
M8	Ternary	$d \equiv_3 e$	$((d - e)/3, e)$	$3\alpha + \delta$
M9	Binary	$e \equiv_2 0$	$(d, e/2)$	$\alpha + \delta$

Montgomery mentions several variations. A constant different from 4 can be used in M3. Another option is discarding the ternary rules (henceforth we will refer to the ternary steps as optional steps as well). For these steps it is convenient to keep track of the residue classes of d and e modulo 3. These are easily updated if any of the other steps applies, but require a division by 3 if either one of the optional steps is carried out. Although the resulting Lucas chains are on average shorter including

the ternary steps, it depends on the implementation and the platform whether or not an overall saving is obtained by including them. For most recurrences and in most software implementations it will most likely be worthwhile.

If only the subtractive rule is allowed, without the restriction $d \leq 4e$, a double-exponentiation version of Montgomery's CFRC-algorithm emerges.

3.3.2. Twofold Exponentiation. Bleichenbacher [**19**] gives a recursive adaptation of Montgomery's algorithm that can be used for twofold exponentiation. On input a base v and positive exponents n and m with $n > m$, the algorithm returns v_n, v_m, and v_{n-m}. Table 3.2 lists the set of rules derived by Bleichenbacher (the corrected version). Since Bleichenbacher was mainly interested in finding upper bounds on the length of strong addition chains for integers of which he could determine the exact value by a search algorithm, he did not allow step B2 and he did not optimize for large n and m. For convenience, we use d and e in this table instead of n and m (as they occur in the algorithm).

Algorithm 3.26 (Bleichenbacher's Twofold Adaptation).
Given v and positive integers n and m, $n > m$, this algorithm computes $u = \gcd(n, m)$ and returns u and the triple $(v_{n/u}, v_{m/u}, v_{(n-m)/u})$.

1. [Finished?] If $m = 0$, terminate with output n and (v, id, v).

2. [Decrease (n, m)] Pick an admissible transformation S for (n, m), e.g., based on Table 3.2. Set $(d, e) \leftarrow (n, m)S$ and call the algorithm recursively on input base d and exponents d and e to obtain corresponding u and (A, B, C). Reconstruct to the original n and m (for a group this would require $(A, B) \leftarrow (A, B)^{S^{-1}}$ and $C \leftarrow A/B$) and terminate with output u and (A, B, C).

Table 3.2. Transformations proposed by Bleichenbacher [**19**]

No.	Condition	Substitution(d, e)	Costs
B0	$(d-e) > e$	$(d, d-e)$	α
B1	$4(d-e) \geq e$	$(e, d-e)$	α
B2	$d \equiv_2 0$	$(d/2, d-e)$	$\alpha + \delta$
B3	$e \equiv_2 0$	$(d-e, e/2)$	$\alpha + \delta$
B4	$d \equiv_2 e$	$((d+e)/2, e)$	$\alpha + \delta$

Twofold exponentiation can alternatively be described by working out the recursion. This requires going through the algorithm one time only keeping track of d and e and the substitutions performed. Once $d = e$, work your way back up to $d = n$ and $e = m$ by performing the inverse of each step, but also keeping as invariant $A = v_d, B = v_e$, and $C = v_{d-e}$.

Algorithm 3.27 (Iterative Euclidean Twofold Exponentiation).
Given v and positive integers n and m, $n > m$, this algorithm computes $u = \gcd(n, m)$ and returns u and the triple $(v_{n/u}, v_{m/u}, v_{(n-m)/u})$. It keeps invariant

$$0 \leq e < d, \quad \gcd(d, e) = \gcd(n, m) \ ,$$

TABLE 3.3. Transformations proposed by Tsuruoka [**187**]

No.	Condition	Substitution(d, e)
R0a	$d < e$	(e, d)
R0b	$e < d - e$	$(d, d - e)$
R1a	$e \leq 1.09(d - e), d \equiv_2 0$	$(d/2, e - d/2)$
R1b	$e \geq 2.92(d - e), d \equiv_2 0$	$(d/2, e - d/2)$
R2	$e \geq 4.7(d - e), d \equiv_3 -e$	$((2d - e)/3, (2e - d)/3)$
R3	$e \geq 3.9(d - e), e \equiv_3 0, 4e > 3d$	$(e/3, 4e/3 - d)$
R4	$e \geq 3.9(d - e), e \equiv_2 0$	$(d - e/2, e/2)$
R5	$e \geq 5.8(d - e), d \equiv_3 0, d > 3(d - e)$	$(d/3, e - 2d/3)$
R6	$e \geq 8(d - e), d \equiv_6 e$	$((d + e)/2, e)$
R7	Default	$(e, d - e)$

and during the second half (steps 4 and 5) also

$$A = v_d, \quad B = v_e, \quad C = v_{d-e} \ .$$

1. [Initialize] Set $(d, e) \leftarrow (n, m)$ and $i = 0$.
2. [Halfway?] (If $e = 0$, then both equal $\gcd(n, m)$.) If $e = 0$, set $u \leftarrow d$, followed by $d \leftarrow 1$ and $A \leftarrow v, B \leftarrow id$, and $C \leftarrow v$. Go to step 4.
3. [Decrease (d, e)] Given the current (d, e) pick a transformation S_i, e.g., by using Table 3.2. Set $(d, e) \leftarrow (d, e)S_i$ and increase i by one. Go back to 2.
4. [Finished?] If $i = 0$ terminate with output u and (A, B, C).
5. [Loopback] Set $(d, e) \leftarrow (d, e)S_i^{-1}$, update A, B, and C accordingly, decrease i by one and go back step 4.

Reversing the algorithm this way is similar to the notion of duality for addition chains [**92**], which also boils down to matrix transposition. Therefore, we call the steps S and S^T duals of each other.

EXAMPLE 3.28. The dual of the transformation in the previous example, and more importantly, the corresponding substitution matrix, are:

$$S^T = \begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ 0 & 1 \end{pmatrix}, \quad \text{and} \quad S^{-1} = \left(\begin{array}{cc|c} 2 & 0 & 2 \\ 1 & 1 & 0 \end{array}\right), \tag{11}$$

It follows that the dual costs for S are $2\alpha + \delta$ since updating C requires a doubling, updating A costs two calls to α and B is unaffected. This is more than the costs for the ordinary step.

The costs for a step and its dual are not always the same which partially explains the difference in steps between for instance Montgomery and Bleichenbacher. Tsuruoka [**187**] gives a rather involved set of transformations for Algorithm 3.26 with the purpose of optimization. Table 3.3 lists these rules and although the resulting chains are indeed on average slightly shorter, it will take more time to check which rule applies.

Since the rules M2, M3, M6, and M7 do not necessarily leave $d > e$ invariant, step M0 involving swapping and negations might be required for the double exponentiation algorithm. In the twofold algorithms negations are never required, which is an advantage.

3.3.3. Timings. Based on extensive simulations, the expected practical behaviour of Montgomery's Euclidean algorithms is well understood, and the practical merits of the method are beyond doubt. However, a satisfactory theoretical analysis of Algorithms 3.25 and 3.26 is still lacking. The rules given by Montgomery and Bleichenbacher are reminiscent of the binary and subtractive Euclidean greatest common divisor algorithms. Iterations of that sort typically exhibit an unpredictable behaviour with a wide gap between worst and average case performance; see for instance [**8, 90, 190**] and the analysis attempts and open problems in [**128**].

This is further illustrated in Section 4.6.3, where a third degree version (Algorithm 3.37) is extensively tested for the specific case of XTR. Table 4.1 on page 84 shows that the average runtime is linear in the length of the exponent and that the (standard) deviation from this average is fairly small. Figure 4.1 on page 85 depicts the average runtime when the constant similar to 4 in rule M3 (Table 3.1) is changed. The remarkable shape of the curves —both with at least four local minima— is a clear indication that the exact behaviour of (in that case) Algorithm 3.37 will be hard to analyse.

CONJECTURE 3.29. *On input two random positive k-bit integers, Algorithm 3.25 takes on average $(1.49\alpha + 0.33\delta)k$ for a double exponentiation.*

This compares favourably with Akishita's runtime. The same is true for a twofold exponentiation when compared to Algorithm 3.11. The twofold exponentiation is a bit slower than the double exponentiation because it is not optimized as much.

CONJECTURE 3.30. *On input two random positive k-bit integers, Algorithm 3.26 takes on average $(1.5\alpha + 0.5\delta)k$ for a twofold exponentiation.*

3.3.4. Applications to Single Exponentiation.

With Precomputation. Single exponentiation can be sped up by precomputing $v_{2\lfloor k/2 \rfloor}$, effectively transforming a k-bit single exponentiation into a $k/2$-bit double exponentiation that can be performed using Montgomery's double exponentiation algorithm. A minor detail is that $v_{2\lfloor k/2 \rfloor - 1}$ is also required (corresponding to the difference between $2^{\lfloor k/2 \rfloor}$ and 1). Fortunately, Algorithm 3.9 returns both. In the algorithm below we assume that the order of the recurrence is q.

ALGORITHM 3.31 (Single Exponentiation with Precomputation).
Given an exponent n, a base v and precomputed values v_t and v_{t-1} for known t, this algorithm computes v_n.

1. [Split the exponent] Compute non-negative integers n_1 and n_2 such that $n = n_1 + n_2 t \bmod q$ and n_1 and n_2 are at most about $\sqrt{q}$. If $\lg(t \bmod q) \approx (\lg q)/2$ then a long division suffices to compute the desired n_1 and n_2. Otherwise, use

a lattice-based method as described in Section 2.3.5. With the proper choice of t this results in n_1 and n_2 that are small enough.

2. [Perform a double exponentiation] Use Algorithm 3.25 based on exponents n_1 and n_2 and bases v, v_t and v_{t-1} in conjunction with Algorithm 3.33 to take care of $\gcd(n_1, n_2)$ to compute $v_{n_1+n_2 t} = v_n$. Terminate with output v_n.

COROLLARY 3.32. *On input a base v and precomputed values $v_{\lfloor \frac{k}{2} \rfloor - 1}$ and $v_{\lfloor \frac{k}{2} \rfloor}$, computation of v_n takes on average $(0.75\alpha + 0.17\delta)k$ for a k-bit integer n using Algorithm 3.31.*

Variations with other values instead of $2^{\lfloor k/2 \rfloor}$ are possible. The awkwardness of (Lucas) multi-exponentiation in general makes further splitting of the exponent inefficient (Section 2.3.5). For Algorithm 3.11 precomputation of all values v_{2^j} for $j = 1, \ldots, k-1$ leads to a speedup that requires many more precomputed values and can be expected to be slower unless $\dot{\alpha}_3$ is considerably cheaper than α.

Without Precomputation. Although the algorithm in [**128**] actually describes a double exponentiation, it was only used there for single exponentiations by computing v_n as $v_{\kappa(n-r)+\lambda r}$ with $\kappa = \lambda = 1$ and r arbitrary. (A small variation could use $v_n = v_{\kappa n + \lambda r}$ with $\kappa = 1, \lambda = 0$ and r arbitrary.) The choice of r will then determine the speed of the single exponentiation. Montgomery proposed setting $(n-r)/r \approx \phi$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. This will result in $\log_\phi \sqrt{n}$ Fibonacci steps (type M3) costing one ordinary addition each, followed by what looks like a random double exponentiation with exponents of magnitude about $\sqrt{n}$. Montgomery also proposed to exploit the factorization of n —if it is known— but this does not seem to provide much additional speedup.

ALGORITHM 3.33 (Montgomery's PRAC Algorithm).
Given a base v and an exponent n, this algorithm computes v_n.

1. [Make d odd] Let f_2 be the highest power of 2 dividing n. Set $d \leftarrow (n/2^{f_2})$ and $A \leftarrow \delta^{f_2}(v)$.

2. [Make $d \not\equiv 0 \pmod 3$] Let f_3 be the highest power of 3 dividing n. Set $d \leftarrow d/3^{f_3}$ and $A \leftarrow \tau^{f_3}(A)$.

3. [$d = 1$?] If $d = 1$, the algorithm terminates with output A.

4. [Initialize new gcd calculation] Let $p > 1$ be a divisor of d, not necessarily prime (e.g., $p = d$). Set $r \leftarrow \lfloor \frac{p}{\phi} \rfloor$ (or a better value $p/2 < r < p$ if known) and set $(d, e) \leftarrow (rd/p, d - rd/p)$.

5. [Compute "A^p"] Run Algorithm 3.25 on input (A, A, id) and (d, e). Let the output be u and $\tilde{A}$. Set $d \leftarrow u$ and $A \leftarrow \tilde{A}$. Go back to step 3.

The analogue for twofold exponentiation is also known: computing both v_n and v_r is sufficient for computing v_n. Moreover, it is also necessary for at least some r. This requires changing the final step into:

5'. [Compute "A^p"] Run Algorithm 3.26 on input base A and exponents d and e. Let the output be u and $(\tilde{A}, \tilde{B}, \tilde{C})$. Set $d \leftarrow u$ and $A \leftarrow \tilde{A}$. Go back to step 3.

To analyse the effects of using $r = \lfloor \frac{p}{\phi} \rfloor$ we will assume that n is not divisible by 6 and that in step 4 the trivial divisor $p = n$ will be chosen. Moreover, we will ignore the possibility that steps M1 or M2 are used (the proof of the proposition can be extended to this case as well, since $\frac{5}{4} < \phi$).

PROPOSITION 3.34. *In the call to Algorithm 3.25 in Step 5 of Algorithm 3.33, the values of d and e in Step 3 of Algorithm 3.25 are reduced to approximately half their original sizes using a sequence of approximately $\log_\phi \sqrt{n} \approx 0.72 \lg n$ iterations using just Fibonacci steps.*

Proof: Let $m = \text{round}(\log_\phi n)$. Asymptotically for $m \to \infty$ the values d and e in Algorithm 3.33 satisfy $d/e = \phi + \epsilon_1$ with $|\epsilon_1| = O(2^{-m})$. Furthermore, for $m \to \infty$, the m-th Fibonacci number F_m satisfies $\frac{F_m}{F_{m-1}} = \phi + \epsilon_2$ with $|\epsilon_2| = O(2^{-m})$. It follows that $e = \frac{F_{m-1}}{F_m} d + \epsilon_3$, where $|\epsilon_3|$ is bounded by a small positive constant.

Define $(d_0, e_0) = (d, e)$ and $(d_i, e_i) = (e_{i-1}, d_{i-1} - e_{i-1})$ for $i > 0$. We claim that

$$d_i = \frac{F_{m-i}}{F_m} d - (-1)^i F_i \epsilon_3 \tag{12}$$

for $0 \le i < m$. For $i = 0$ it follows from $d_0 = d = \frac{F_{m-0}}{F_m} d - (-1)^0 F_0 \epsilon_3$ and for $i = 1$ it follows from $d_1 = e_0 = e = \frac{F_{m-1}}{F_m} d + \epsilon_3$. We proceed by induction on i. Suppose that the statement is true for d_i and d_{i-1}. From $d_{i+1} = e_i = d_{i-1} - e_{i-1} = d_{i-1} - d_i$ and the induction hypothesis we obtain that

$$\begin{aligned} d_{i+1} &= \frac{F_{m-(i-1)}}{F_m} d - (-1)^{i-1} F_{i-1} \epsilon_3 - (\frac{F_{m-i}}{F_m} d - (-1)^i F_i \epsilon_3) \\ &= \frac{F_{m+1-i} - F_{m-i}}{F_m} d - (-1)^{i+1} (F_i + F_{i-1}) \epsilon_3 \\ &= \frac{F_{m-(i+1)}}{F_m} d - (-1)^{i+1} F_{i+1} \epsilon_3 \ , \end{aligned}$$

proving our claim. Algorithm 3.25 as called from Algorithm 3.33 will perform Fibonacci steps as long as $e_i < d_i < 2e_i$. But as soon as $d_i > 2e_i$ this nice behaviour will be lost. From $e_i = d_{i+1}$ and (12) it follows that $d_i > 2e_i$ is equivalent to

$$\frac{F_{m-i-3}}{F_m} d < (-1)^{i-1} F_{i+3} \epsilon_3 .$$

Because F_m/d and $|\epsilon_3|$ are both bounded by small positive constants, the first time this condition will hold is when F_{m-i-3} and F_{i+3} are of the same order of magnitude, i.e., $m - i - 3 \approx i + 3$. Thus, the Fibonacci behaviour is lost after about $m/2 = \log_\phi \sqrt{n}$ iterations, at which point $d_i \approx \sqrt{n}$ (this follows from (12)). *Q.E.D.*

If insufficient precision is used in the computation of r in Step 4 of Algorithm 3.33, then ϵ_3 in the proof of Proposition 3.34 is no longer bounded by a small constant. It follows that $d_i > 2e_i$ already holds for a smaller value of i, implying that the Fibonacci behaviour is lost earlier. A precise analysis of the expected

performance degradation as a function of the lack of precision is straightforward. In practice this effect is very noticeable.

A counting argument shows that random k-bit single exponentiations also lead to random $\frac{k}{2}$-bit double exponentiations. Given two $\frac{k}{2}$-bit values d and e, there are at most two k-bit values in the Fibonacci sequence with d and e as initial values. Since there are roughly as many k-bit numbers as pairs of $\frac{k}{2}$-bit numbers, the double exponentiation resulting after the Fibonacci steps will be random.

Putting the pieces together, we obtain the following corollary.

COROLLARY 3.35. *Given a base v and a random k-bit exponent n, computing v_n using Algorithm 3.33 costs on average $(1.47\alpha+0.17\delta)k$ if Algorithm 3.25 is called in step 5 (cf. Conjecture 3.29) and on average $(1.47\alpha + 0.25\delta)k$ if Algorithm 3.26 is called instead (cf. Conjecture 3.30).*

3.3.5. Extension to Perrin Chains. Extending Algorithm 3.22 to third degree recurrences is relatively straightforward by introducing $D = c_{a-2b}$. The only difficulty arises in step 3 if $d > e$. Setting $b \leftarrow a + b$ would require $D \leftarrow c_{-(a+2b)}$ which would require an additional application of α. If a and b are swapped the problem is resolved.

ALGORITHM 3.36 (Third Degree Adaptation of CFRC).
On input a base c, an exponent n and an auxiliary 'exponent' m satisfying $0 < m < n$ and $\gcd(n, m) = 1$ this algorithm outputs c_n. It keeps invariant

$$0 \le e, \quad 0 \le d, \quad \gcd(d, e) = 1, \quad ad + be = n,$$
$$A = c_a, \quad B = c_b, \quad C = c_{a-b}, \quad D = c_{a-2b}\ .$$

1. [Initialize] Set $(d, e) \leftarrow (n - m, m)$ and $(A, B, C) \leftarrow (v, v, id)$.
2. [Finished?] If $e = 0$, terminate with output A. If $d = 0$, terminate with output B.
3. [Decrease (d, e)] If $d > e$ set $(a, b) \leftarrow (a + b, a), (d, e) \leftarrow (e, d - e)$, and $(A, B, C, D) \leftarrow (\alpha(A, B, C, D), A, B, \nu(C))$; else $(d \le e)$ set $a \leftarrow a+b, e \leftarrow e-d$ and $(A, C, D) \leftarrow (\alpha(A, B, C, D), A, C)$. Go back to the previous step.

The algorithm above is just as fast, or rather just as slow, as Montgomery's CFRC algorithm; improvements are needed. One option is implementing the classical Euclidean step based on Algorithm 3.16. We will leave this to the reader. The method below is an adaptation of Montgomery's Algorithm 3.25 to the present case of third degree sequences.

ALGORITHM 3.37 (Third Degree Euclidean Double Exponentiation).
Given bases $c_\kappa, c_\lambda, c_{\kappa-\lambda}$, and $c_{\kappa-2\lambda}$ and exponents n, m, this algorithm outputs $u = \gcd(n, m)$ and $c_{(n\kappa+m\lambda)/u}$. It uses invariant

$$d \ge 0, \quad e \ge 0, \quad ad + be = n\kappa + m\lambda, \quad \gcd(d, e) = \gcd(n, m),$$
$$A = c_a, \quad B = c_b, \quad C = c_{a-b}, \quad D = c_{a-2b}\ .$$

(a and b are carried along for expository purposes only).

1. [Initialization] Let $a = \kappa$, $b = \lambda$, $d = n$, $e = m$, $A = c_\kappa$, $B = c_\lambda$, $C = c_{\kappa-\lambda}$, $D = c_{\kappa-2\lambda}$

2. [Both even?] Set $f_2 \leftarrow 0$. As long as d and e are both even, replace (d, e) by $(d/2, e/2)$ and f_2 by $f_2 + 1$.

3. [Both triple?] Set $f_3 \leftarrow 0$. As long as d and e are both divisible by 3, replace (d, e) by $(d/3, e/3)$ and f_3 by $f_3 + 1$.

4. [Finished?] If $d = 0$ terminate with output $e2^{f_2}3^{f_3}$ and B. If $e = 0$ terminate with output $d2^{f_2}3^{f_3}$ and A.

5. [Decrease (d, e)] Pick an admissible matrix S for (d, e) (e.g., based on the first applicable rule in Table 3.4). Set $(d, e) \leftarrow (d, e)S$ and $(a, b) \leftarrow (a, b)S^{-T}$. Update A, B, C and D accordingly, respecting the invariant. Go back to the previous step.

TABLE 3.4. Transformations for Algorithm 3.37

No.	Condition	Substitution (d, e)	Costs
Substitutions if $d \geq e$			
X1	$d \leq 4e$	$(e, d - e)$	α
X2	$d \equiv_2 0$	$(d/2, e)$	$\alpha + 2\delta$
X3	$d \equiv_2 e$	$((d - e)/2, e)$	$\alpha + 2\delta$
X4	$d \equiv_3 e$	$((d - e)/3, e)$	$2\alpha + \tau$
X5	$e \equiv_2 0$	$(e/2, d)$	2δ
Substitutions if $e \geq d$			
X6	$e \leq 4d$	$(d, e - d)$	α
X7	$e \equiv_2 0$	$(e/2, d)$	2δ
X8	$d \equiv_2 1$	$((e - d)/2, d)$	$\alpha + 2\delta$
X9	$e \equiv_3 0$	$(e/3, d)$	$2\alpha + \tau$
X10	$e \equiv_3 d$	$((e - d)/3, d)$	$2\alpha + \tau$
X11	$d \equiv_2 0$	$(d/2, e)$	$\alpha + 2\delta$

The asymmetry between the case $d \leq e$ and $d \geq e$ is caused by the asymmetry between a and b, i.e., c_{a-2b} is available but c_{b-2a} is not. As a consequence, the case '$d \equiv 0 \bmod 3$' is slower than the case '$e \equiv 0 \bmod 3$' (rule X9), and its inclusion would slow down Algorithm 3.37.

Similarly to the Lucas case, we consider the ternary rules $X4, X9$, and $X10$ optional. Leaving them out slows down the algorithm a little bit.

CONJECTURE 3.38. *On input two random positive k-bit integers, Algorithm 3.37 takes on average $(1.38\alpha + 0.80\delta + 0.05\tau)k$ to perform a double exponentiation.*

Strictly speaking, steps 2 and 3 are not necessary, since this part of the greatest common divisor will come out any way. However, in practice the greatest common divisor has to be dealt with as well; in this case it helps to already know f_2 and f_3. Another small advantage is the mutual exclusion of for instance $X2, X4$, and $X5$,

which allows to play a little with their order in the table (this only affects the time the control code takes).

Twofold exponentiation is also possible. Different substitutions are needed. Table 3.5 gives an example but some room for optimization should remain, considering the algorithm is slower than its double exponentiation counterpart.

ALGORITHM 3.39 (Third Degree Euclidean Twofold Exponentiation). Given c and positive integers n and m, $n > m$, this algorithm computes $u = \gcd(n, m)$ and returns u and the ordered tuple $(c_{n/u}, c_{m/u}, c_{(n-m)/u}, c_{(n-2m)/u})$. It keeps invariant

$$0 \le e < d, \quad 2^{f_2} 3^{f_3} \gcd(d, e) = \gcd(n, m) \ ,$$

and during the second half (steps 6 and 7) also

$$A = c_d, \quad B = c_e, \quad C = c_{d-e}, \quad D = c_{d-2e} \ .$$

1. [Initialize] Set $(d, e) \leftarrow (n, m)$ and $i = 0$.
2. [Both even?] Set $f_2 \leftarrow 0$. As long as d and e are both even, replace (d, e) by $(d/2, e/2)$ and f_2 by $f_2 + 1$.
3. [Both triple?] Set $f_3 \leftarrow 0$. As long as d and e are both divisible by 3, replace (d, e) by
4. [Halfway?] If $e = 0$, set $u \leftarrow 2^{f_2} 3^{f_3} d$ followed by $d \leftarrow 0$ and $A \leftarrow c, B \leftarrow id, , C \leftarrow c$, and $D \leftarrow c$. Go to Step 6.
5. [Decrease (d, e)] Given the current (d, e) pick a transformation S_i, e.g., by using Table 3.5. Set $(d, e) \leftarrow (d, e)S_i$ and increase i by one. Go back to the previous step.
6. [Finished?] If $i = 0$ terminate with output u and (A, B, C, D).
7. [Loopback] Set $(d, e) \leftarrow (d, e)S_i^{-1}$, update A, B, C and D accordingly, decrease i by one and go back to the previous step.

TABLE 3.5. Transformations for Algorithm 3.39

No.	Condition	Substitution (d, e)	Costs
Y0	$e \le d < 2e$	$(e, d - e)$	α
Y1	$d \le 4e$	$(d - e, e)$	α
Y2	$d \equiv_2 0$	$(d/2, e)$	$\alpha + 2\delta$
Y3	$d \equiv_2 e$	$((d - e)/2, e)$	$2\alpha + 2\delta$
Y4	$e \equiv_2 0$	$(d - e, e/2)$	$\alpha + 2\delta$

CONJECTURE 3.40. *On input two random positive k-bit integers, Algorithm 3.39 takes on average $(1.7\alpha + 1.1\delta)k$ to perform a twofold exponentiation.*

Single exponentiation without precomputation follows the same idea as that for Lucas chains. In fact, Algorithm 3.33 can be used with step 5 replaced either by

5'. [Compute "A^p"] Run Algorithm 3.37 on input $(A, A, id, \nu(A))$ and d and e. Let the output be u and $\tilde{A}$. Set $d \leftarrow u$ and $A \leftarrow \tilde{A}$. Go back to Step 3.

or by

5'. [Compute "A^p"] Run Algorithm 3.39 on input A and exponents d and e. Let the output be u and $(\tilde{A}, \tilde{B}, \tilde{C}, \tilde{D})$. Set $d \leftarrow u$ and $A \leftarrow \tilde{A}$. Go back to Step 3.

Proposition 3.34 remains valid.

COROLLARY 3.41. *Given a base c and an exponent n, computing c_n using Algorithm 3.33 costs on average $1.41\alpha + 0.40\delta + 0.03\tau$ per exponent bit if Algorithm 3.37 is called in step 5 (cf. Conjecture 3.38).*

3.4. Theoretical Remarks

There are not a lot of theoretical results on strong addition chains, let alone Lucas chains, Perrin chains, vectorial Lucas chains or even vectorial Perrin chains. The binary algorithms and common sense give $\lg n \leq L_2(n) \leq \ell_L(n) \leq 2\lg n$ and $\lg n \leq L_2(n) \leq L_3(n) \leq 3\lg n$.

Montgomery proves that $\ell_L(mn) \leq \ell_L(m)\ell_L(n)$ using the same kind of composition that is also known for addition chains. He also proves that if a strong addition chain $\langle c_0, \ldots, c_l \rangle$ is ordered and doubling step $c_{i+1} = 2c_i$ occurs, then c_i divides all subsequent c_j with $j > i$. Therefore, unless n is highly composite, a chain will not contain many doubling steps. A chain with only the initial doubling step $c_2 = 2c_1$ is called a simple chain by Bleichenbacher [**19**]. Even though Lucas chains are not necessarily increasing, they can be sufficiently ordered to allow a similar argument. Simply say that if $i < j$, then either $a_i < a_j$ or it was not possible to put a_j in the chain at time i. The same decomposition result now holds.

Since strong addition chains are addition chains, a known lower bound on addition chains without doubling steps applies to simple chains. Montgomery gave a slightly stronger interpretation for simple chains that seems fairly tight (Bleichenbacher [**19**], who computed $\ell_L(n)$ for $n < 500000$). The most efficient step in a simple chain after the initial doubling is a Fibonacci step which is the basis for the lower bound. Kutz [**97**] shows that the lower bound holds for most integers (because most integers are not highly composite). Going through the proofs, it is clear that having subtractions at ones disposal does not shorten the chain, hence the same lower bound applies. The same lower bound also applies to Perrin chains. Moreover, in the general case it will be hard to prove a better bound, since it is tight for the primes in the Fibonacci sequence. Nevertheless, the Perrin chains generated by the algorithms in this chapter are significantly longer than corresponding Lucas chains. Note that the CFRC algorithm gives the same length for both Lucas and Perrin chain. This is an encouraging thought if only we knew how to pick the optimal m for a given n. (Under some conjectures Montgomery proves that this would give chains of length $\approx 1.77 \lg n$, which is slightly better than the heuristics of Algorithm 3.37 when applied to single exponentiation.)

Although Montgomery's PRAC algorithm performs splendidly, the binary algorithm still seems to give the best average case asymptotic upper bound. It is not hard to find integers n that perform worse for the PRAC algorithm than for the binary algorithm.

Montgomery posed the question whether there exists some number n for which the shortest Lucas chain is shorter than the shortest strong addition chain. We conjecture this *not* to be the case.

CONJECTURE 3.42. *The shortest strong addition chain for a positive integer n is equally long as the shortest Lucas chain for that integer.*

Meaningful lower bounds for vectorial Lucas chains or even Lucas sequences are not known. These would have some implications on duality as well, although duality does not hold the same way as it did for addition chains. A nice counterexample are the chains for the vectors $\binom{n}{1}$ and $\binom{n}{n-1}$ —these are almost equally long— and the sequences $(1, n)$ and $(n-1, n)$. The first sequence can be computed in as little as $\lg n$ doublings if n is a power of 2, whereas for the second sequence and the vectors a binary algorithm seems optimal giving a length of about $2 \lg n$.

4

Finite Field Extensions

Recall that G_x, for a positive integer x, denotes a cyclic (sub)group of order x. In this chapter we consider subgroups G_q of $\mathbb{F}_{p^d}^*$ with q a prime dividing the d-th cyclotomic polynomial Φ_d evaluated at p with a focus on the groups $\mathsf{G}_{p+1} \subset \mathbb{F}_{p^2}^*$ and $\mathsf{G}_{p^2-p+1} \subset \mathbb{F}_{p^6}^*$. For both subgroups it has been proposed to use trace maps to speed up exponentiation, resulting in respectively LUC [**175**] and XTR [**107**]. We show that computation in the subgroups can also be sped up without trace. Moreover, we speed up XTR considerably and show that XTR works for $p \equiv 3 \bmod 4$ as well without significant loss of efficiency compared to $p \equiv 2 \bmod 3$ (the original proposal). The relevance of the cyclotomic subgroups for cryptography is explained in Section 1.2.2.

4.1. Introduction

Computation in finite fields is a well studied problem. However, research tends to emphasize bilinear complexity [**102**], asymptotic complexity [**196**], or binary characteristic [**2**]. The case of large prime characteristic with small extension degree has been studied less extensively, but Cohen and Lenstra [**47**] give efficient implementations of squaring and multiplication for fields $\mathbb{F}_{p^d}$ for even $d \leq 10$ and restricted congruency classes of p. Moreover, usually the entire field is discussed. Lenstra [**104**] points out that, given a subgroup G_q of $\mathbb{F}_{p^d}^*$, unless q divides $\Phi_d(p)$, the group G_q can be embedded in a true subfield of $\mathbb{F}_{p^d}^*$, thereby making the discrete logarithm computation substantially easier given current understanding of the DLP. However, efficient computation in the cyclotomic subgroups is not addressed in full detail.

Currently the fastest exponentiation methods in the subgroups G_{p^2-p+1} and G_{p+1} use trace maps, resulting in respectively LUC [**175**] and XTR [**107**]. (We will use the notation Tr_1 for the trace from $\mathbb{F}_{p^2}$ to $\mathbb{F}_p$ and Tr_2 for the trace from $\mathbb{F}_{p^6}$ to $\mathbb{F}_{p^2}$.) LUC and XTR have two main advantages compared to ordinary representation of elements of G_q:

- It is shorter: for LUC, $\mathrm{Tr}_1(g) \in \mathbb{F}_p$ takes only half the space of representing $g \in \mathbb{F}_{p^2}$ traditionally; for XTR $\mathrm{Tr}_2(g) \in \mathbb{F}_{p^2}$, whereas representing an element of G_q requires in general an element of $\mathbb{F}_{p^6}$, i.e., three times more bits.

- It allows faster arithmetic, because given $\mathrm{Tr}(g)$ and n the value $\mathrm{Tr}(g^n)$ can be computed substantially faster than an exponentiation in the entire field ($\mathbb{F}_{p^2}$ resp. $\mathbb{F}_{p^6}$).

Application of LUC and XTR is advantageous in protocols where the subgroup operations are restricted to additions, and single or double exponentiations. But for more involved protocols that also require ordinary multiplications of subgroup elements or triple (or larger) exponentiations, they may lead to cumbersome manipulations that outweigh the computational advantages. As a consequence, using trace based representations in more complicated protocols may be inconvenient (unless of course the small representation size is crucial).

For that reason, we consider in this chapter how exponentiation speedups in G_{p+1} and G_{p^2-p+1} can be achieved in such a way that other operations are not affected, i.e., while avoiding trace based compression methods.

For quadratic extensions we show that for both $p \equiv 2 \bmod 3$ and $p \equiv 3 \bmod 4$ inversions in $\mathsf{G}_{p+1} \subset \mathbb{F}_{p^2}^*$ come for free, and that squaring in G_{p+1} is cheaper than in the field $\mathbb{F}_{p^2}$. This results in single and double exponentiations that cost about 60% and 75%, respectively, of traditional methods. Both methods are still considerably slower than LUC.

More substantial are our results concerning sixth degree extensions, i.e., the case $\mathsf{G}_{p^2-p+1} \subset \mathbb{F}_{p^6}^*$. We show that for both $p \equiv 2 \bmod 9$ and $p \equiv 5 \bmod 9$ inversions in G_{p^2-p+1} are very cheap, while squaring in G_{p^2-p+1} is substantially faster than in $\mathbb{F}_{p^6}$. Moreover, the method implied by Lemma 2.29 can be used to transform a k-bit single exponentiation into a $k/2$-bit double exponentiation (i.e., the product of two $k/2$-bit exponentiations). Using appropriate addition chains this results in a vastly improved single exponentiation routine, that takes approximately 26% of the time cited in [**107**, Lemma 2.1.2.*iii*]. The improvement for double exponentiation is less spectacular, requiring an estimated 33% compared to [**107**, Lemma 2.1.2.*iv*]. Somewhat surprisingly, the result is faster than the original XTR [**107**].

However, we also present a speedup for XTR. Part of the speedup is obtained by applying the faster $\mathbb{F}_{p^2}$ arithmetic, derived from [**47**] and described in Section 4.3. As a side result, we also show that XTR works for $p \equiv 3 \bmod 4$ just as well as for $p \equiv 2 \bmod 3$. Application of Montgomery's method as adapted to the third degree in the previous chapter gives a total speedup of 60% for double exponentiation and more than 35% for single exponentiation compared to [**107**].

After some preliminaries we describe our results on $\mathbb{F}_{p^2}$ and the group $\mathsf{G}_q \subset \mathsf{G}_{p+1} \subset \mathbb{F}_{p^2}^*$. For completeness and comparisons we have included LUC as intermezzo, before presenting our results on $\mathsf{G}_q \subset \mathsf{G}_{p^2-p+1} \subset \mathbb{F}_{p^2}^*$ and XTR. We briefly discuss alternatives based on subgroups of other finite fields and conclude with timings and comparisons.

4.2. Preliminaries

4.2.1. Finite Field Representation. In cryptography, d-th degree extensions of finite fields are most commonly represented using either polynomial or normal bases (see [**111**] for definitions and details).

With a proper choice of minimal polynomial (such as a trinomial with small coefficients), polynomial bases allow relatively efficient multiplication and squaring in the sense that the usual reduction stage from a degree $2d-2$ product to the degree $d-1$ result can be performed at the cost of cd additions in the underlying prime field, for a very small constant c. In general, this is not the case for normal bases, but they have the advantage that the Frobenius automorphism can be computed for free. For polynomial bases the Frobenius automorphism can be computed at a small but non-negligible cost.

Cohen and Lenstra [**47**] describe a class of polynomial bases combining the best of both worlds. They are based on the rings $\mathbb{Z}[\gamma]/p\mathbb{Z}[\gamma]$ where γ is a primitive n-th root of unity, n is a prime power, and p is an integer of which primality is to be determined. If p is indeed a prime and p generates $\mathbb{Z}_n^*$ then $\mathbb{Z}[\gamma]/p\mathbb{Z}[\gamma]$ is isomorphic to $\mathbb{F}_p[\gamma]$ supporting identical representations. The following theorem, a slight adaptation of [**111**, Theorem 2.47(ii)], says something about the extension degrees one obtains using cyclotomic fields.

THEOREM 4.1. *Given a field $\mathbb{F}_{p^e}$ with p prime and some n coprime to p. Then the n-th cyclotomic field over $\mathbb{F}_{p^e}$ is isomorphic to $\mathbb{F}_{p^{ed}}$ where d is the least positive integer such that $p^{ed} \equiv 1 \bmod n$.*

This theorem implies $d|\phi(n)$. We fix $e = 1$. Furthermore, we concentrate on $d = \phi(n)$, i.e., the case that $p \bmod n$ generates $\mathbb{Z}_n^*$. This requires $\mathbb{Z}_n^*$ to be cyclic, so that n is either 2, 4, the power of an odd prime, or twice the power of an odd prime. We ignore $n = 2$, since it does not lead to a proper extension.

Let $\Gamma = (\gamma, \gamma^2, \ldots, \gamma^d)$ with γ a primitive n-th root of unity, then Γ is a basis of $\mathbb{F}_{p^d}$ over $\mathbb{F}_p$. (Usually a basis is a set of elements, but a vector is easier to manipulate.) It is understood that an element $a \in \mathbb{F}_{p^d}$ is represented as $\mathbf{a} = (a_0, \ldots, a_{d-1}) \in (\mathbb{F}_p)^d$, where $a = \Gamma \cdot \mathbf{a}^T$. We abuse notation by identifying a and $\mathbf{a}$.

The cost of p-th powering in $\mathbb{F}_{p^d}$ is virtually negligible. Since $\Phi_d(p)$ divides $p^{d/2}+1$ for even d (which will be the case if $d = \phi(n), n > 2$), inversion in the group $\mathsf{G}_{\Phi_d(p)}$ comes almost for free by $g^{-1} = g^{p^{d/2}}$.

Of independent interest is membership testing for $\mathsf{G}_{\Phi_d(p)}$. We will henceforth assume that membership of the finite field has already been tested. If $d < 105$, then $\Phi_d(p) = \sum_{i\in\mathcal{P}} p^i - \sum_{i\in\mathcal{N}} p^i$ for appropriate index sets $\mathcal{P}$ and $\mathcal{N}$. (If $d \geq 105$ some of the coefficients in the cyclotomic polynomial might be larger than 1 in absolute value.) Let $a \in \mathbb{F}_{p^d}$. Since $\mathbb{F}_{p^d}^*$ is cyclic, $a \in \mathsf{G}_{\Phi_d(p)}$ if and only if $a^{\Phi_d(p)} = 1$, which is equivalent to $\prod_{i\in\mathcal{P}} a^{p^i} = \prod_{i\in\mathcal{N}} a^{p^i}$. Testing this condition requires at most d applications of the Frobenius automorphism and $|\mathcal{P}|+|\mathcal{N}|-1$ multiplications in $\mathbb{F}_{p^d}$. Thus, for fixed d testing $\mathsf{G}_{\Phi_d(p)}$-membership costs at most $\phi(d)$ multiplications in $\mathbb{F}_{p^d}$. Given membership of $x \neq 1$ in $\mathsf{G}_{\Phi_d(p)}$, membership $x \in \mathsf{G}_q$ can be established by verifying that $x^{\Phi_d(p)/q} \neq 1$, which can efficiently be done if the exponent $\Phi_d(p)/q$ is small.

The relation $\prod_{i\in\mathcal{P}} a^{p^i} = \prod_{i\in\mathcal{N}} a^{p^i}$ gives rise to d possibly dependent relations of degree $|\mathcal{P}|$. In some cases these relations can be used to speed up $|\mathcal{P}|$-th powering

in $\mathsf{G}_{\Phi_d(p)}$. This will be exploited to get fast squaring in $\mathsf{G}_{\Phi_d(p)}$ for $d = 2, 6$ in the upcoming Sections.

A major ingredient when calculating modulo Γ is writing powers of γ higher than d as linear combinations in Γ. This reduction is performed in two stages. First, all powers higher than n are reduced using $\gamma^n = 1$; next the relation $\Phi_n(\gamma) = 0$ is used to map everything to powers of γ between 1 and $\phi(n)$. Since $d = \phi(n)$, we are done. Note that only additions and subtractions are needed for the reduction.

Lenstra [**104**] only considers pairs (p, n) for which n is prime and for which p generates $\mathbb{Z}_n^*$, because they lead to so-called optimal normal bases. The relevance of such bases for characteristics > 2 is limited, and the 'cheap' reduction they achieve (just $2d - 1$ additions in $\mathbb{F}_p$) is almost met by the somewhat wider class considered above, in which n is a prime power.

4.2.2. Key Generation. Given n and $d = \phi(n)$ and a desired level of security, key generation consists of two phases: sufficiently large primes p and q have to be found with p generating $\mathbb{Z}_n^*$ and q dividing $\Phi_d(p)$, after which a generator of G_q has to be found.

Finding p and q. For small d standard security requirements lead to $\log p > \log q$, cf. Section 1.2.2. In this case the obvious generalization of the method from [**107**] can be used. First, an appropriately sized prime q is selected, where $q|\Phi_d(p)$ may impose a priori restrictions on q (e.g., $q \equiv 1 \bmod 3$ for $d = 6$). Next, a root r of $\Phi_d[x] \in \mathbb{F}_q[x]$ is found and p is determined as $r + \ell q$ for $\ell \in \mathbb{Z}_{\geq 0}$ such that p is a large enough prime that generates $\mathbb{Z}_n^*$.

For larger d (or $e > 1$, cf. Theorem 4.1) one may aim for primes p that fit in a computer word (i.e., $\lg p = 32$ or 64). Although this may be advantageous, $\log p$ becomes substantially smaller than $\log q$. We are not aware of an efficient method to find such p and q. If q is selected first, the probability is negligible that an appropriate p exists such that $q|\Phi_{de}(p)$. If p is selected first, there is only a very slim probability that $\Phi_{de}(p)$ has an appropriate prime factor, and finding it leads to an unattractive integer factorization problem.

Finding a Generator of G_q. This problem is easily solved by selecting $h \in \mathbb{F}_{p^d}$ at random until $g = h^{(p^d-1)/q} \neq 1$, at which point g is the desired generator. A faster method is described in [**106**, credited to H.W. Lenstra, Jr.]. First an element $h \in \mathsf{G}_{\phi_d(p)}$ is constructed directly and next $g = h^{\Phi_d(p)/q}$ is computed. If $g = 1$ another h has to be generated. The specifics follow.

Let $f \in \mathbb{F}_p$ and let γ be a primitive n-th root of unity as in Section 4.2.1. Consider $h_f = (\gamma + f)^{(p^d-1)/\Phi_d(p)} \in \mathsf{G}_{\phi_d(p)}$. Since $\Phi_d(p)$ divides $p^d - 1$ irrespective of p, we can write $(p^d - 1)/\Phi_d(p)$ as $\sum_{i=0}^{d-\phi(d)} f_i p^i$. By separating the positive from the negative coefficients by $r_{p+} = \sum_{i=0, f_i>0}^{d-\phi(d)} f_i p^i$ and $r_{p-} = -\sum_{i=0, f_i<0}^{d-\phi(d)} f_i p^i$ we obtain $(\gamma+f)^{r_{p+}} = h_f(\gamma+f)^{r_{p-}}$, giving rise to a system of d equations linear in the coefficients of h_f. Since the system only depends on p's congruency class modulo n (and not on p itself), solving the system can be done before actually picking p. The resulting h_f corresponding to several different choices for f can be hard coded in the program. In Section 4.5.4 the details for G_{p^2-p+1} with $p \equiv 2 \bmod 9$ are presented.

4.3. Quadratic Extensions

In this section we discuss computations in $\mathbb{F}_{p^2}$ and $\mathsf{G}_{p+1} \subset \mathbb{F}_{p^2}^*$. Fast computations in the full field $\mathbb{F}_{p^2}$ with $p \equiv 2 \bmod 3$ are important for XTR and have been discussed by Lenstra and Verheul [**107**] (but without delaying the reductions, cf. [**47**, Case $p^k = 3$]). We show that the field arithmetic for $p \equiv 3 \bmod 4$ from [**47**, Case $p^k = 4$] can be used for XTR without significant loss of efficiency compared to $p \equiv 2 \bmod 3$. The subgroup G_{p+1} is not relevant for XTR, but it is the subgroup on which LUC is based. We show that it yields some extra computational benefits that are, however, still not competitive with LUC.

We first discuss the field arithmetic for $p \equiv 2 \bmod 3$ in general and then focus on the subgroup. The case $p \equiv 3 \bmod 4$ is dealt with similarly, first the field arithmetic and then the subgroup arithmetic. Suitable exponentiation routines that apply to either case conclude this section.

4.3.1. Field Representation for $p \equiv 2 \bmod 3$.

Field Arithmetic. Let p and q be primes with $p \equiv 2 \bmod 3$ and $q|(p+1)$. Then p generates $\mathbb{Z}_3^*$ and $\Phi_3(x) = x^2 + x + 1 | (x^3 - 1)$ is irreducible in $\mathbb{F}_p$. Let γ denote a root of $\Phi_3(x)$, then $\gamma^n = \gamma^{(n \bmod 3)}$ and in particular $\gamma^p = \gamma^2$. Hence $\Gamma = (\gamma, \gamma^2)$ is an optimal normal basis of $\mathbb{F}_{p^2}$ over $\mathbb{F}_p$. Using Γ instead of $(1, \gamma)$ leads to slightly fewer additions than the basis $(1, \gamma)$ discussed in [**47**, Case $p = 3$]. Lemma 4.2 is easily implied by the formulae from [**107**, Section 2.1] (cf. [**107**, Lemma 2.1.1], and [**47**, Case $p = 3$]). For the sake of completeness we provide the details.

LEMMA 4.2. *Let $a, b, c \in \mathbb{F}_{p^2}$ with $p \equiv 2 \bmod 3$.*

i. Computing a^p is free.
ii. Computing a^2 costs $2M + 2D + 3A_1$.
iii. Computing ab costs $3M + 2D + 2A_1 + 2A_2$.
iv. Computing $ac - bc^p$ costs $4M + 2D + 6A_1 + 2A_2$.

Proof: Let $a \in \mathbb{F}_{p^2}$ be represented by $(a_0, a_1) \in (\mathbb{F}_p)^2$, i.e., $a = \Gamma \cdot (a_0, a_1)^T = a_0\gamma + a_1\gamma^2$ (similarly for b and c). Using Fermat's little theorem we have that $a_0^p = a_0$ etc. From $(a_0\gamma + a_1\gamma^2)^p = a_0^p\gamma^p + a_1^p\gamma^{2p}$ and $\gamma^p = \gamma^2$ we obtain $a^p = a_1\gamma + a_0\gamma^2$, so p-th powering is essentially for free. Squaring a requires computation of $a^2 = (a_1 - 2a_0)a_1\gamma + (a_0 - 2a_1)a_0\gamma^2$, which can be done in the claimed $2M + 2D + 3A_1$. Multiplication of a and b can be done using Karatsuba's technique by computing $ab = ((a_0 - a_1)(b_0 - b_1) - a_0b_0)\gamma + ((a_0 - a_1)(b_0 - b_1) - a_1b_1)\gamma^2$ and then (*iii*) follows by inspection. Computation of $ac - bc^p$, relevant for XTR, boils down to computing $((b_0 - b_1 - a_1)f_0 + (b_1 + a_1 - a_0)f_1)\gamma + ((a_0 - a_1 + b_0)f_0 + (b_1 - b_0 - a_0)f_1)\gamma^2$. This can be done in several ways, all costing $4M + 2D$ and a handful of additions. *Q.E.D.*

Subgroup Arithmetic. Because $x^{p+1} = 1$ for $x \in \mathsf{G}_{p+1}$, we find that inversion in G_{p+1} is equivalent to p-th powering and thus for free. Let $a = a_0\gamma + a_1\gamma^2$ with $a_0, a_1 \in \mathbb{F}_p$, so $a \in \mathbb{F}_{p^2}$. Then $a \in \mathsf{G}_{p+1}$ if and only if $a^{p+1} = a^p \cdot a = 1$, i.e., $(a_1\gamma + a_0\gamma^2)(a_0\gamma + a_1\gamma^2) = 1$. This is equivalent to $a_0^2 - a_0a_1 + a_1^2 = 1$, so that G_{p+1}-membership testing costs $M + S + D + A_1 + A_2$ plus a comparison with one.

This relation can also be exploited to speed up squaring in G_{p+1}, since the value of a_0a_1 follows from a_0^2 and a_1^2 using only a handful of additions. More specifically, $a^2 = (2 - 2a_0^2 - a_1^2)\gamma + (2 - a_0^2 - 2a_1^2)\gamma^2$, which costs $2S + 2D + 2A_1 + 3A_2$.

Free inversion in G_{p+1} also results in an advantage for simultaneous computation of ab and ab^{-1} for $a \in \mathbb{F}_{p^2}$ and $b \in \mathsf{G}_{p+1}$: since there are only four possible combinations a_ib_j, four multiplications suffice.

LEMMA 4.3. *Let G_{p+1} be the order $p+1$ subgroup of $\mathbb{F}_{p^2}^*$ with $p \equiv 2 \bmod 3$ and let $a = a_0\gamma + a_1\gamma^2 \in \mathbb{F}_{p^2}$ with $\Phi_3(\gamma) = 0$.*

i. The element a is in $\mathbb{F}_p$ if and only if $a_0 = a_1$.
ii. The element a is in G_{p+1} if and only if $a_0^2 - a_0a_1 + a_1^2 = 1$. Testing this costs $M + S + D + A_1 + A_2$.
iii. Computing a^{-1} for $a \in \mathsf{G}_{p+1}$ is free.
iv. Computing a^2 for $a \in \mathsf{G}_{p+1}$ costs $2S + 2D + 2A_1 + 3A_2$.
v. Computing ab and ab^{-1} for $b \in \mathsf{G}_{p+1}$ costs $4M + 4D + 6\min(A_1, A_2)$.

4.3.2. Field Representation for $p \equiv 3 \bmod 4$.

Field Arithmetic. Let p and q be primes with $p \equiv 3 \bmod 4$ and $q|(p+1)$. Then p generates $\mathbb{Z}_4^*$ and $\Phi_4(x) = x^2 + 1$ is irreducible in $\mathbb{F}_p$. Let γ denote a root of $\Phi_4(x)$, then $\Gamma = (1, \gamma)$ is a basis of $\mathbb{F}_{p^2}$ over $\mathbb{F}_p$. (Since $\gamma^2 = -1$ the basis (γ, γ^2) looks contrived and leads to slightly more complicated reductions.) This field representation is identical to [**47**, Case $p^k = 4$], although the number of additions in our cost functions is slightly different.

Let $a \in \mathbb{F}_{p^2}$ be represented by $(a_0, a_1) \in (\mathbb{F}_p)^2$, i.e., $a = \Gamma \cdot (a_0, a_1)^T = a_0 + a_1\gamma$. From $\gamma^n = \gamma^{(n \bmod 4)}$ and thus $\gamma^p = \gamma^3 = -\gamma$ it follows that $a^p = a_0^p + a_1^p\gamma^p = a_0 - a_1\gamma$ so that p-th powering costs a modular negation. The cost of multiplication is $3M + 2D + 2A_1 + 3A_2$ since $ab = a_0b_0 - a_1b_1 + ((a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1)\gamma$. The cost of squaring is $2M + 2D + 2A_1$ since $a^2 = (a_0 + a_1)(a_0 - a_1) + 2a_0a_1\gamma$. The cost of computing $ac - bc^p$ for $a, b, c \in \mathbb{F}_{p^2}$ is $4M + 2D + 2A_1 + 2A_2$ since $ac - bc^p = (b_0 + a_0)c_0 + (b_1 - a_1)c_1 + ((a_1 - b_1)c_0 + (a_0 + b_0)c_1)\gamma$. By analogy with Lemma 4.2 we get the following.

LEMMA 4.4. *Let $a, b, c \in \mathbb{F}_{p^2}$ with $p \equiv 3 \bmod 4$.*

i. Computing a^p costs A_1.
ii. Computing a^2 costs $2M + 2D + 2A_1$.
iii. Computing ab costs $3M + 2D + 2A_1 + 3A_2$.
iv. Computing $ac - bc^p$ costs $4M + 2D + 2A_1 + 2A_2$.

Subgroup Arithmetic. As in the $p \equiv 2 \bmod 3$ case, inversion in G_{p+1} is equivalent to p-th powering; it costs A_1. Let $a = a_0 + a_1\gamma$ with $a_0, a_1 \in \mathbb{F}_p$, so $a \in \mathbb{F}_{p^2}$. Then $a \in \mathsf{G}_{p+1}$ if and only if $a^{p+1} = a^p \cdot a = 1$, i.e., $(a_0 - a_1\gamma)(a_0 + a_1\gamma) = 1$ which is equivalent to $a_0^2 + a_1^2 = 1$. So, G_{p+1}-membership testing costs $2S + D + A_2$. It also follows that $a^2 = 2a_0^2 - 1 + ((a_0 + a_1)^2 - 1)\gamma$ for $a \in \mathsf{G}_{p+1}$, which implies that squaring in G_{p+1} can be done faster than in $\mathbb{F}_{p^2}$.

LEMMA 4.5. *Let G_{p+1} be the order $p+1$ subgroup of $\mathbb{F}_{p^2}^*$ with $p \equiv 3 \bmod 4$ and let $a = a_0 + a_1\gamma \in \mathbb{F}_{p^2}$ with $\Phi_4(\gamma) = 0$.*

i. The element a is in $\mathbb{F}_p$ if and only if $a_1 = 0$.
ii. The element a is in G_{p+1} if and only if $a_0^2 + a_1^2 = 1$. Testing this costs $2S + D + A_2$.
iii. If $a \in \mathsf{G}_{p+1}$, then computing a^{-1} costs A_1.
iv. If $a \in \mathsf{G}_{p+1}$, then computing a^2 costs $2S + 2D + A_1$.
v. Computing ab and ab^{-1} for $b \in \mathsf{G}_{p+1}$ costs $4M + 4D + 6\min(A_1, A_2)$.

4.3.3. Subgroup Exponentiation. Under Assumption 1.3 we get for the subgroup G_{p+1} for both $p \equiv 2 \bmod 3$ and $p \equiv 3 \bmod 4$ that $\delta \approx 2S + 2D \approx 1.6$ field multiplications in $\mathbb{F}_p$ and that $\alpha \approx 3M + 2D \approx 2.5$ field multiplications in $\mathbb{F}_p$. In both cases inversion is essentially available for free and group multiplication with a fixed multiplicand offers no speedup, so $\dot{\alpha} = \alpha$. We can now directly apply existing exponentiation routines as described in Chapter 2, Table 2.7.

For a single exponentiation we have to compute a^m, where m has roughly the same bitlength k as q. Using the signed sliding window method with windows of size 4, this requires about $k + 1$ squarings and $7 + k/6$ multiplications in G_q. The resulting number of $\mathbb{F}_p$-multiplications is 19.1 for the precomputation plus 2.0 per exponent bit.

For a double exponentiation we have to compute $a^m b^n$ for m and n of roughly equal size and with m as above. This can be computed using the JSF resulting in k squarings and $k/2$ multiplications in G_q. With G_q-arithmetic as above, this becomes $(2S + 2D)k + (3M + 2D)k/2 \approx 2.85k$ multiplications in $\mathbb{F}_p$. The precomputation of ab and ab^{-1} uses Lemmas 4.3 and 4.5. Combination of these observations leads to the following theorem.

THEOREM 4.6. *Let p and q be primes with $q|p+1$, $p \equiv 2 \bmod 3$ or $p \equiv 3 \bmod 4$, and $\lceil \lg q \rceil = k$. Let a, b be in the order q subgroup G_q of $\mathbb{F}_{p^2}^*$ and $m, n \in \mathbb{Z}_q$. Under Assumption 1.3*

i. computing a^m costs on average $19.1 + 2k$ multiplications in $\mathbb{F}_p$, and
ii. computing $a^m b^n$ costs on average $4 + 2.85k$ multiplications in $\mathbb{F}_p$.

4.4. LUC

This section is included for completeness and comparison. We start with a brief description of the basic principles of LUC, followed by runtime estimates for exponentiation. We conclude with some remarks concerning the history of LUC. Like the two schemes in the previous section, LUC exploits the group $\mathsf{G}_{p+1} \subseteq \mathbb{F}_{p^2}^*$, but with a more concise representation and faster arithmetic. The LUC cryptosystem derives its name from the Lucas sequence.

4.4.1. Description of LUC. Let p and q be primes such that q divides $p + 1$. In the previous section we discussed calculation in G_{p+1} directly. In LUC elements of G_{p+1} are represented by their trace over $\mathbb{F}_p$. Let $\mathrm{Tr}_1 : \mathbb{F}_{p^2} \to \mathbb{F}_p$ be the trace over $\mathbb{F}_p$ defined by $\mathrm{Tr}_1(g) = g + g^p$. For $g \in \mathsf{G}_{p+1}$ we have that $g^p = g^{-1}$ and hence $\mathrm{Tr}_1(g) = g + g^{-1}$. From $\mathrm{Tr}_1(g) = \mathrm{Tr}_1(g^p)$ it follows that in LUC an element and its conjugate are identified with each other.

Since $g \in \mathsf{G}_{p+1}$ implies that $(X-g)(X-g^p) = X^2 - (g+g^p)X + g^p g = X^2 - \mathrm{Tr}(g)X + 1$, the roots of the polynomial $X^2 - \mathrm{Tr}(g)X + 1$ are g and its conjugate g^p. Hence, the trace of an element in G_{p+1} determines the element up to conjugacy. Moreover, given an element $v \in \mathbb{F}_p$, it is the trace of an element $g \in \mathsf{G}_{p+1}$ precisely when $X^2 - vX + 1$ is irreducible over $\mathbb{F}_p$.

Let $v_n \in \mathbb{F}_p$ denote the trace over $\mathbb{F}_p$ of g^n. The following statements are well known.

LEMMA 4.7. *Let $g \in \mathsf{G}_{p+1}$ and $v_n = \mathrm{Tr}_1(g^n)$, then*

i. $v_{-n} = v_{np} = v_n$ *for all* $n \in \mathbb{Z}$.
ii. $v_{n+m} = v_n v_m - v_{n-m}$ *for all* $n, m \in \mathbb{Z}$, *costing* $M + D + A_1$ *to compute.*
iii. $v_{2n} = v_n^2 - 2$ *for all* $n \in \mathbb{Z}$, *costing* $S + D$ *to compute.*

Proof: The first identity follows from $\mathrm{Tr}_1(g) = g + g^{-1}$ for $g \in \mathsf{G}_{p+1}$. The second identity can be verified as follows:

$$\begin{aligned} v_{n+m} &= g^n g^m + g^{np} g^{mp} + (g^n g^{mp} + g^m g^{np}) - (g^n g^{mp} + g^m g^{np}) \\ &= (g^n + g^{np})(g^m + g^{mp}) - (g^n g^{-m} + g^{-mp} g^{np}) \\ &= v_n v_m - v_{n-m} \ . \end{aligned}$$

The final claim is a corollary of the second claim for $n = m$ using $v_0 = \mathrm{Tr}_1(1) = 2$. *Q.E.D.*

4.4.2. Efficient Computation.

Exponentiation. The methods to compute short Lucas chains from Chapter 3, can be directly applied to LUC. From the previous section it is clear that under Assumption 1.3 we have that $\alpha \approx 1$ and $\delta \approx 0.8$ measured in the number of $\mathbb{F}_p$-multiplications (and $\dot{\alpha} = \alpha$ and $\nu = 0$).

COROLLARY 4.8. *Let p and q be primes with $q|(p+1)$, let $g \in \mathbb{F}_{p^2}$ be a generator of G_q, let $v_\kappa = \mathrm{Tr}_1(g^\kappa), v_\lambda = \mathrm{Tr}_1(g^\lambda)$, and $v_{\kappa-\lambda} = \mathrm{Tr}_1(g^{\kappa-\lambda})$ and finally let $n, m \in \mathbb{Z}_q$. Under Assumption 1.3*

i. computing $v_{m\kappa}$ costs on average 1.61 $\mathbb{F}_p$-multiplications (cf. Conjecture 3.35);
ii. computing $v_{m\kappa+n\lambda}$ costs on average 1.75 $\mathbb{F}_p$-multiplications (cf. Conjecture 3.29);
iii. computing $v_{m\kappa}$ and $v_{n\kappa}$ simultaneously costs on average 1.75 $\mathbb{F}_p$-multiplications (cf. Conjecture 3.30);
iv. computing $v_{m\kappa}$ and $v_{(m-1)\kappa}$ simultaneously costs 1.8 $\mathbb{F}_p$-multiplications (based on Lemma 3.10);
v. computing $v_{m\kappa}$ costs on average 0.88 $\mathbb{F}_p$-multiplications if $v_{t\kappa}$ and $v_{(t-1)\kappa}$ are given for some known t with $\lg t \approx \frac{1}{2}\lg m$ (cf. Conjecture 3.32).

Membership Testing. Testing whether an element $v \in \mathbb{F}_p$ is indeed the trace of an element $g \in \mathbb{F}_{p^2}$ of order $q|(p+1)$ consists of two parts. The first part tests whether v is the trace of an element g of order dividing $p+1$. Since this is the case iff $X^2 - vX + 1$ is irreducible in $\mathbb{F}_p$, a single, cheap Jacobi-symbol computation suffices. Now that v is indeed the trace of an element in $\mathsf{G}_{p+1} \subseteq \mathbb{F}_{p^2}$

the exponentiation machinery just described can be used to check whether $v_q = id$ or not. This exponentiation costs about $1.61 \lg q$ $\mathbb{F}_p$-multiplications.

4.4.3. Historical Remarks. The history of LUC is a bit muddled. Müller and Nöbauer [**133**] suggested replacing the exponentiation function in RSA by the evaluation of Dickson polynomials $g_k(1, x)$ where k takes the place of the exponent, x that of the base and g is defined by

$$g_k(a, x) = \sum_{i=0}^{\lfloor \frac{k}{2} \rfloor} \frac{k}{k-i} \binom{k-i}{i} (-a)^i x^{k-2i} \tag{13}$$

working over a commutative unital ring. Müller and Nöbauer propose to use similar rings as in RSA, that is $\mathbb{Z}_n$ where the factorization of n is to be kept secret. Later they use the identity $g_k(a, u + \frac{a}{u}) = u^k + (\frac{a}{u})^k$, where u is a unit in some extension ring, to "efficiently" compute $g_k(1, u)$ by lifting it to the extension ring and afterwards going back again [**134**].

In 1993, Smith and Lennon [**176**] proposed to use the Lucas functions $V_k(x, 1)$ modulo an RSA modulus as alternative to RSA. Since $V_k(x, 1) = g_k(1, x)$, this system is equivalent to the Dickson scheme.

Later, Smith and Skinner [**175**] presented the LUC cryptosystem modulo a prime, instead of a composite, to be used for ElGamal and Diffie-Hellman. At first the authors thought that no subexponential time algorithm existed. However, the LUC cryptosystem turned out to be based on $\mathbb{F}_{p^2}$ since $V_k(v, 1) = v_k$ in our notation [**20, 98, 99**]. At the same time several breaks that apply to careless implementations of RSA and ElGamal were shown to work for their (careless) LUC counterparts [**22, 78, 79, 147, 148**]. As a result of all this bad publicity, LUC was not taken very serious any more.

LUC as described above is presented to match XTR and make as clear as possible what is going on algebraically. Whenever we use LUC, we implicitly mean working over a finite field. The term Dickson scheme can be used to refer to the RSA analogues.

4.5. Sixth Degree Extensions

In this section fast exponentiation routines for the group $\mathsf{G}_{p^2-p+1} \subset \mathbb{F}_{p^6}^*$ with $p \equiv 2 \bmod 9$ are described. Let f be a sixth degree irreducible polynomial over some ground field, with root γ. Consider the extension induced by γ and represented by a polynomial basis consisting of six consecutive powers of γ, such as $(1, \gamma, \ldots, \gamma^5)$ or $(\gamma, \gamma^2, \ldots, \gamma^6)$. The cost of computation in this representation depends on the general question of how many ground field multiplications are needed to multiply two degree five polynomials, and on specific properties of f. Therefore, a short word on the multiplication of fifth degree polynomials in general, before going into details about the field representation and the benefits the cyclotomic subgroup offers. These results are then used in the subsequent exponentiation routines. We conclude this section with an improved key selection method.

4.5.1. Multiplication of Fifth Degree Polynomials. Multiplication of two polynomials of degree five can be done in 18 multiplications plus a handful of additions based on Karatsuba's technique [**9, 47, 85**]. Indeed, let $G(x) = \sum_{i=0}^{5} g_i x^i$ and $H(x) = \sum_{i=0}^{5} h_i x^i$ be two fifth degree polynomials. Write $G = G_0 + G_1 x^3$ and $H = H_0 + H_1 x^3$ where G_0, G_1, H_0, and H_1 are second degree polynomials. Then

$$GH = G_0H_0 + (G_0H_1 + G_1H_0)x^3 + G_1H_1x^6,$$

so that, with $C_0 = G_0H_0$, $C_1 = G_1H_1$, and $C_2 = (G_0 - G_1)(H_0 - H_1)$, it follows that

$$GH = C_0 + (C_0 + C_1 - C_2)x^3 + C_1x^6. \tag{14}$$

Each of the C_i can be computed using 6 multiplications in the ground field. For example, because $G_0 = g_0 + g_1x + g_2x^2$ and $H_0 = h_0 + h_1x + h_2x^2$,

$$C_0 = g_0h_0 + (g_1h_0 + g_0h_1)x + (g_2h_0 + g_1h_1 + g_0h_2)x^2 + (g_2h_1 + g_1h_2)x^3 + (g_2h_2)x^4,$$

so that, with $c_0 = g_0h_0$, $c_1 = g_1h_1$, $c_2 = g_2h_2$, $c_3 = (g_0 - g_1)(h_0 - h_1)$, $c_4 = (g_0 - g_2)(h_0 - h_2)$, and $c_5 = (g_1 - g_2)(h_1 - h_2)$, we have that

$$C_0 = c_0 + (c_0 + c_1 - c_3)x + (c_0 + c_1 + c_2 - c_4)x^2 + (c_1 + c_2 - c_5)x^3 + c_2x^4.$$

With similar expressions for C_1 and C_2 it follows that 18 ground field multiplications (or squarings) suffice to compute the product GH (or the square G^2).

If the g_i and h_i are l-bit numbers, and one is interested in an (unreduced) product with $2l$-bit or slightly larger coefficients, then computing C_0 costs $6M + 6A_1 + 7A_2$ and the cost of computing GH as in (14) is $18M + 24A_1 + 21A_2$.

It remains to reduce GH modulo f, at a cost depending on f. This is discussed in the remainder of this section for ground fields $\mathbb{F}_p$ with $p \equiv 2 \bmod 9$. In that case the resulting coefficients still have to be reduced modulo p at a cost of $6D$ for l-bit p.

4.5.2. Field Representation for $p \equiv 2 \bmod 9$.

Field Arithmetic. Let p be prime with $p \equiv 2 \bmod 9$. Then p generates $\mathbb{Z}_9^*$ and $\Phi_9(x) = x^6 + x^3 + 1$ is irreducible in $\mathbb{F}_p$. Let γ denote a root of $\Phi_9(x)$, then $\Gamma = (\gamma, \gamma^2, \ldots, \gamma^6)$ is a basis for $\mathbb{F}_{p^6}$ over $\mathbb{F}_p$ (in [**47**, Case $p^k = 9$] the similar basis $(1, \gamma, \ldots, \gamma^5)$ is used).

Let $a = \sum_{i=0}^{5} a_i\gamma^{i+1} \in \mathbb{F}_{p^6}$. From $\gamma^n = \gamma^{n \bmod 9}$ and thus $\gamma^p = \gamma^2$ it follows with $\Phi_9(\gamma) = 0$ that $a^p = a_4\gamma + (a_0 - a_3)\gamma^2 + a_5\gamma^3 + a_1\gamma^4 - a_3\gamma^5 + a_2\gamma^6$. Thus, p-th powering costs A_1. In a similar way it follows that p^3-th powering costs $2A_1$. For multiplication in $\mathbb{F}_{p^6}$ the method from Section 4.5.1 is used, with proper adjustment of the powers of x, e.g., $G = G_0x + G_1x^4$. It follows with straightforward bookkeeping that collecting corresponding powers of x in (14) combined with the modular reductions costs $12A_2 + 6D$. (For the basis $(1, \gamma, \ldots, \gamma^5)$ we find that the collecting phase costs $14A_2$, which slightly improves the $18A_2$ reported in [**47**].) With Section 4.5.1 it follows that multiplication can be done for $18M + 6D + 24A_1 + 33A_2$. Doing more elaborate collecting reduces the $33A_2$ to $29A_2$.

Squaring follows by replacing $18M$ by $18S$, but it can be done substantially faster by observing that

$$G^2 = (G_0\gamma + G_1\gamma^4)^2 = (G_0 - G_1)(G_0 + G_1)\gamma^2 + (2G_0 - G_1)G_1\gamma^5,$$

with $G_0, G_1 \in \mathbb{F}_p[\gamma]$ of degree two. Computing this costs $9A_1$ for the preparation of the multiplicands, two polynomial multiplications costing $6M + 6A_1 + 7A_2$ each, $7A_2$ for the collection, and $6D$ for the final reductions. It follows that squaring can be done for $12M + 6D + 21A_1 + 21A_2$. (This is A_2 more than reported in [**47**] for $(1, \gamma, \ldots, \gamma^5)$.)

LEMMA 4.9. *Let $a, b \in \mathbb{F}_{p^6}$ with $p \equiv 2 \bmod 9$.*

i. Computing a^p or a^{p^5} costs A_1.
ii. Computing a^{p^2}, a^{p^3}, or a^{p^4} costs $2A_1$.
iii. Computing a^2 costs $12M + 6D + 21A_1 + 21A_2$.
iv. Computing ab costs $18M + 6D + 24A_1 + 29A_2$.

Subgroup Arithmetic. Let $a = \sum_{i=0}^{5} a_i\gamma^{i+1} \in \mathbb{F}_{p^6}$. Membership of one of the three proper subfields of $\mathbb{F}_{p^6}$ is characterized by one of the equations $a^{p^i} = a$ for $i = 1, 2, 3$. Specifically, $a \in \mathbb{F}_p$ if and only if $a^p = a$ which is equivalent to the system of linear equations $(a_0, a_1, a_2, a_3, a_4, a_5) = (a_4, a_0 - a_3, a_5, a_1, -a_3, a_2)$. The solution $a_0 = a_1 = a_3 = a_4 = 0$ and $a_2 = a_5$ is not surprising since $1 + \gamma^3 + \gamma^6 = 0$, so an element $c \in \mathbb{F}_p$ takes the form $-c\gamma^3 - c\gamma^6$. Similarly, $a \in \mathbb{F}_{p^2}$ if and only if $a^{p^2} = a$, which is equivalent to $a = a_2\gamma^3 + a_5\gamma^6$, and $a \in \mathbb{F}_{p^3}$ if and only if $a^{p^3} = a$ or $a = (a_3 - a_4)\gamma + (-a_3 + a_4)\gamma^2 + a_5\gamma^3 + a_3\gamma^4 + a_4\gamma^5 + a_5\gamma^6$.

More interesting for cryptographic purposes is the subgroup G_{p^2-p+1} of $\mathbb{F}_{p^6}^*$, because that subgroup cannot be embedded in a proper subfield of $\mathbb{F}_{p^6}$. The G_{p^2-p+1}-membership condition $a^{p^2-p+1} = 1$ is equivalent to $a^{p^2}a = a^p$, which can be verified at a cost of, essentially, a single $\mathbb{F}_{p^6}$-multiplication. From $a^{p^3} = a^{-1}$ it follows that inversion in G_{p^2-p+1} costs $2A_1$.

Computing $a^{p^2}a - a^p = \sum_{i=0}^{5} v_i\gamma^{i+1}$ symbolically produces

$$\begin{aligned}
v_0 &= a_1^2 - a_0a_2 - a_4 - a_4^2 + a_3a_5 \ ; \\
v_1 &= -a_0 + a_1a_2 + a_3 - 2a_0a_3 + a_3^2 - a_2a_4 - a_1a_5 \ ; \\
v_2 &= -a_0a_1 + a_3a_4 - a_5 - 2a_2a_5 + a_5^2 \ ; \\
v_3 &= -a_1 - a_2a_3 + 2a_1a_4 - a_4^2 - a_0a_5 + a_3a_5 \ ; \\
v_4 &= a_0^2 + a_1a_2 + a_3 - 2a_0a_3 - a_4a_5 \ ; \\
v_5 &= -a_2 + a_2^2 - a_1a_3 - a_0a_4 + a_3a_4 - 2a_2a_5 \ .
\end{aligned} \tag{15}$$

If $a \in \mathsf{G}_{p^2-p+1}$, then $v_i = 0$ for $0 \le i < 6$ and the resulting six relations can be used to significantly reduce the cost of squaring in G_{p^2-p+1}. Let $V = (v_0, v_1, \ldots, v_5)$ be the vector consisting of the v_i's. Then for any 6×6-matrix M, we have that $a^2 + \Gamma \cdot (M \cdot V^T) = a^2$ if $a \in \mathsf{G}_{p^2-p+1}$, because in that case V is the all-zero vector. Carrying out this computation symbolically, involving the expressions for the v_i's

for a particular choice of M yields the following:
(16)

$$a^2 = a^2 + 2\Gamma \cdot \begin{pmatrix} 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot V^T = \Gamma \cdot \begin{pmatrix} 2a_1 + 3a_4(a_4 - 2a_1) \\ 2a_0 + 3(a_0 + a_3)(a_0 - a_3) \\ -2a_5 + 3a_5(a_5 - 2a_2) \\ 2(a_1 - a_4) + 3a_1(a_1 - 2a_4) \\ 2(a_0 - a_3) + 3a_3(2a_0 - a_3) \\ -2a_2 + 3a_2(a_2 - 2a_5) \end{pmatrix}.$$

Given that we are working over a sixth degree extension, the six multiplications and reductions required for (16) seem optimal. The additions can be taken care of in several ways; a reasonable solution results in $6M + 6D + 9A_1 + 12A_2$.

LEMMA 4.10. *Let G_{p^2-p+1} be the order $p^2 - p + 1$ subgroup of $\mathbb{F}_{p^6}^*$ with $p \equiv 2 \bmod 9$ and let $a = a_0\gamma + a_1\gamma^2 + \cdots + a_5\gamma^6 \in \mathbb{F}_{p^6}$ with $\Phi_9(\gamma) = 0$.*

i. The element a is in $\mathbb{F}_p$ if and only if $a = a_2\gamma^3 + a_2\gamma^6$.
ii. The element a is in $\mathbb{F}_{p^2}$ if and only if $a = a_2\gamma^3 + a_5\gamma^6$.
iii. The element a is in $\mathbb{F}_{p^3}$ if and only if $a = (a_3 - a_4)\gamma + (-a_3 + a_4)\gamma^2 + a_5\gamma^3 + a_3\gamma^4 + a_4\gamma^5 + a_5\gamma^6$.
iv. The element a is in G_{p^2-p+1} if and only if in relations (15) $v_i = 0$ for $0 \le i < 6$. This can be checked at a cost of essentially $18M + 6D$.
v. Computing a^{-1} for $a \in \mathsf{G}_{p^2-p+1}$ costs $2A_1$.
vi. Computing a^2 for $a \in \mathsf{G}_{p^2-p+1}$ costs $6M + 6D + 9A_1 + 12A_2$.

4.5.3. Subgroup Exponentiation. Exponentiation can be performed using the algorithms described in Chapter 2. Under Assumption 1.3 the costs $\alpha \approx 12$ and $\delta \approx 6$, both measured in the number of $\mathbb{F}_p$-multiplications, follow from Lemmas 4.9 and 4.10. Fixed multiplicands do not seem to offer an advantage, so $\dot{\alpha} = \alpha$ and inversion is for free.

For a single exponentiation we have to compute a^m, where m has roughly the same bitlength k as q. For the case $q|(p^2 - p + 1)$ the Frobenius endomorphism will save us some work, since m can quickly be written as $m \equiv m_1 + m_2 p \bmod q$ with m_1 and m_2 of bitlength $k/2$ according to Lemma 2.29. Hence a^m can be rewritten as $a^{m_1}(a^p)^{m_2}$. This double exponentiation can be computed using the Joint Sparse Form, resulting in a cost of about six $\mathbb{F}_p$-multiplications per exponent bit.

A double exponentiation $a^m b^n$, with $\log m \approx \log n$ and m as above, can be rewritten as $a^{m_1}(a^p)^{m_2} b^{n_1}(b^p)^{n_2}$ with $\approx k/2$-bit m_1, m_2, n_1, and n_2. This quadruple exponentiation can be computed using the JSF resulting in a total of about 9 multiplications in $\mathbb{F}_p$ per exponent bit.

Combination of these observations leads to the following theorem.

THEOREM 4.11. *Let p and q be primes with $q|(p^2 - p + 1)$, $p \equiv 2 \bmod 9$, and $\lceil \log_2 q \rceil = k$. Let a, b be in the order q subgroup G_q of $\mathbb{F}_{p^6}^*$ and $m, n \in \mathbb{Z}_q$. Assuming that $M \approx D$,*

i. computing a^m costs on average $12 + 6k$ multiplications in $\mathbb{F}_p$, and
ii. computing $a^m b^n$ costs on average $24 + 9k$ multiplications in $\mathbb{F}_p$, and

4.5.4. Key Selection. We elaborate on the improved key selection mentioned in Section 4.2.2 and similar to [**106**, Algorithm 4.5]. With $f \in \mathbb{F}_p$ and $h_f = (\gamma + f)^{(p^6-1)/\Phi_6(p)}$ it follows that $h_f(\gamma+f)(\gamma+f)^p = (\gamma+f)^{p^3}(\gamma+f)^{p^4}$. Solving this equation for the coefficients of h_f gives

$$h_f = \frac{\Gamma}{f^6 - f^3 + 1} \cdot \begin{pmatrix} -f + f^2 + 3f^3 - f^4 - 2f^5 \\ -f - 2f^2 + 3f^3 + 2f^4 - 2f^5 \\ (1-f^2)^3 \\ f - f^2 + f^4 - f^5 \\ f - f^2 + f^4 - f^5 \\ -f^3(1 - 3f + f^3) \end{pmatrix}. \tag{17}$$

This gives $h_{1/2} = \frac{1}{19}(0, -12, 9, 6, 6, 1)$ and $h_2 = -\frac{3}{19}(6, 2, 3, 2, 2, 8/3)$.

Given either $h \in \mathsf{G}_{\phi_d(p)}$, the element $g = h^{(p^2-p+1)/q}$ generates G_q unless $g = 1$. The probability of failure may be expected to be q^{-1}, independently for each h. This is negligible. Theorem 4.11 does not apply for the exponentiation $h^{(p^2-p+1)/q}$ because the splitting of the exponent requires order q. However, using the signed sliding window method and assuming that p is only slightly larger in size than q the exponentiation takes about $8 \log p + 90$ ground field multiplications.

Our methods work, and result in identical runtimes, as long as p mod 9 generates $\mathbb{Z}_9^*$. Since $\phi(\phi(9)) = 2$, the only other case is $p \equiv 5 \bmod 9$.

4.6. Speeding up XTR

4.6.1. Introduction. The XTR public key system was introduced at Crypto 2000 by Lenstra and Verheul [**107**]. It is based on the same subgroup $\mathsf{G}_{p^2-p+1} \subseteq \mathbb{F}_{p^6}^*$ as the system described in the previous section. From a security point of view XTR is therefore, like LUC, a traditional subgroup discrete logarithm system. Lenstra and Verheul have also written an overview of XTR, incorporating several improvements [**109**].

This section contains several important speedups for XTR, while at the same time simplifying its implementation. In the first place the field arithmetic as described in [**107**] is improved by combining the modular reduction steps as described in Section 4.3. More importantly, the Euclidean algorithms from Section 3.3.5 are used instead of binary methods. These improvements result in an XTR double exponentiation method that is on average more than 60% faster than the double exponentiation from [**107**]. Such exponentiations are used in XTR ElGamal-like signature verifications. Furthermore, they result in two new XTR single exponentiation methods, one that is on average about 60% faster than the method from [**107**] but that requires a one-time precomputation, and a generic one without precomputation that is on average 35% faster than the old method.

Examples where precomputation can typically be used are the 'first' of the two exponentiations (per party) in XTR Diffie-Hellman key agreement, XTR ElGamal-like signature generation, and, to a lesser extent, XTR-ElGamal encryption. The new generic XTR single exponentiation can be used in the 'second' XTR Diffie-Hellman exponentiation and in XTR-ElGamal decryption. As a result the runtime

of XTR signature applications is more than halved, the time required for XTR Diffie-Hellman is almost halved, and XTR-ElGamal encryption and decryption can both be expected to run at least 35% faster (with encryption running more than 60% faster after precomputation).

4.6.2. Description of XTR. Let p and q be primes with q dividing p^2-p+1. For $g \in \mathbb{F}_{p^6}^*$ its trace $\mathrm{Tr}_2(g)$ over $\mathbb{F}_{p^2}$ is defined as the sum of the conjugates over $\mathbb{F}_{p^2}$ of g:

$$\mathrm{Tr}_2(g) = g + g^{p^2} + g^{p^4} \in \mathbb{F}_{p^2} \ .$$

Because the order of g divides p^6-1 the trace over $\mathbb{F}_{p^2}$ of g equals the trace of the conjugates over $\mathbb{F}_{p^2}$ of g:

$$\mathrm{Tr}_2(g) = \mathrm{Tr}_2(g^{p^2}) = \mathrm{Tr}_2(g^{p^4}) \ . \tag{18}$$

In XTR elements of G_{p^2-p+1} are represented by their trace over $\mathbb{F}_{p^2}$. It follows from (18) that XTR makes no distinction between an element of $\langle g \rangle$ and its conjugates over $\mathbb{F}_{p^2}$. If $g \in \mathsf{G}_{p^2-p+1}$ then its order divides p^2-p+1, so that

$$\mathrm{Tr}_2(g) = g + g^{p-1} + g^{-p}$$

since $p^2 \equiv p-1 \bmod (p^2-p+1)$ and $p^4 \equiv -p \bmod (p^2-p+1)$. These relations, together with $\mathrm{Tr}_2(g^{-1}) = \mathrm{Tr}_2(g^p) = \mathrm{Tr}_2(g)^p$ also imply that

$$(X-g)(X-g^{p^2})(X-g^{p^4}) = X^3 - \mathrm{Tr}_2(g)X^2 + \mathrm{Tr}_2(g)^p X - 1 \ .$$

If $g \in \mathsf{G}_{p^2-p+1}$ then the element g, or one of its conjugates, can be retrieved from $c = \mathrm{Tr}_2(g)$ by determining a root of the cubic $X^3 - cX^2 + c^pX - 1$. Lenstra and Verheul also show that any given $c \in \mathbb{F}_{p^2}$ is the trace of some element in G_{p^2-p+1} if the cubic polynomial is irreducible over $\mathbb{F}_{p^2}$.

Throughout this section, c_n denotes $\mathrm{Tr}_2(g^n) \in \mathbb{F}_{p^2}$, for some fixed p and g of order q dividing p^2-p+1 as above. Note that $c_0 = 3$ and $c_1 = c$. The notation c_n nicely corresponds with the notation used for Perrin chains in Chapter 3, since (proof omitted)

$$c_{n+m} = c_n c_m - c_n^p c_{n-m} + c_{n-2m} \ . \tag{19}$$

Efficient computation of c_n given p, q, and c therefore depends of efficient implementation of (19). Lenstra and Verheul propose to use the field arithmetic for $p \equiv 2 \bmod 3$ described in Section 4.3.1 (but without the delayed reductions). From Section 4.3.2 it follows that XTR can be implemented for $p \equiv 3 \bmod 4$ just as easily without loss of efficiency compared to $p \equiv 2 \bmod 3$.

If we want to measure the cost of the elementary XTR operations in the number of underlying $\mathbb{F}_p$-multiplications, we use Lemma 4.2 for $p \equiv 2 \bmod 3$ and Lemma 4.4 for $p \equiv 3 \bmod 4$. These two lemmas are summarized below in the language of Chapter 3.

LEMMA 4.12. *If $p \equiv 2 \bmod 3$ or $p \equiv 3 \bmod 4$ and assuming Assumption 1.3, the costs of XTR operations measured in the number of multiplications in $\mathbb{F}_p$ is $\alpha \approx 3, \delta \approx 2$ and $\tau \approx 4.5$. Moreover $\dot{\alpha}_i = \alpha$ for $i = 1,2,3,4$ and $\nu = 0$.*

Proof:

i. $c_{-n} = c_{np} = c_n^p$. It follows that negations and p-th powers can be computed for free.

ii. $c_{a+b} = c_a c_b - c_b^p c_{a-b} + c_{a-2b}$. It follows that c_{a+b} can be computed at the cost of three multiplications in $\mathbb{F}_p$ if c_a, c_b, c_{a-b}, and c_{a-2b} are given.

iii. If $c_a = \tilde{c}_1$, then $\tilde{c}_b$ denotes the trace of the b-th power g^{ab} of g^a, so that $c_{ab} = \tilde{c}_b$.

iv. $c_{2a} = c_a^2 - 2c_a^p$ takes two multiplications in $\mathbb{F}_p$.

v. $c_{3a} = c_a^3 - 3c_a^{p+1} + 3$ costs four and a half multiplications in $\mathbb{F}_p$, and produces c_{2a} as a side-result.

Q.E.D.

Lenstra and Verheul [**106**–**108**] show how p, q, and c can be found quickly. In particular there is no need to find an explicit representation of $g \in \mathbb{F}_{p^6}$.

4.6.3. Efficient Computation.

Easy exponentiation. Lenstra and Verheul proposed Algorithm 3.14 (cf. [**107**, Algorithm 2.3.7] to perform a single exponentiation. From Lemma 4.12 it follows this will cost 7 $\mathbb{F}_p$-multiplications per exponent bit. (Because of the not fully optimized $\mathbb{F}_{p^2}$ arithmetic, Lenstra and Verheul report 8 $\mathbb{F}_p$-multiplications per exponent bit.)

The double exponentiation method from [**107**] uses matrices. The cost of a double exponentiation are roughly equal to two single exponentiations, that is 14 $\mathbb{F}_p$-multiplications per exponent bit. Both Algorithms 3.20 and 3.37 do away with the matrices, thereby removing the aesthetically least pleasing aspect of XTR. Remark that Algorithm 3.20 does not achieve a noticeable speedup over the double exponentiation from [**107**], since the matrix steps that are no longer needed, though cumbersome, are cheap.

Using the third-degree adaptation of Montgomery's Euclidean algorithm as described in Section 3.3.5, considerably speeds up single and double exponentiation. Plugging in the values for α, δ, and τ from Lemma 4.12 in the estimated runtimes obtained in Chapter 3 gives us the following corollary:

COROLLARY 4.13. *Let p and q be primes with $q|(p^2 - p + 1)$, let $g \in \mathbb{F}_{p^6}$ be a generator of G_q, let $c_\kappa = \mathrm{Tr}_2(g^\kappa), c_\lambda = \mathrm{Tr}_2(g^\lambda)$, and $c_{\kappa-\lambda} = \mathrm{Tr}_2(g^{\kappa-\lambda})$ and finally let $n, m \in \mathbb{Z}_q$.*

i. computing $c_{m\kappa}$ costs on average 5.2 $\mathbb{F}_p$-multiplications (cf. Corollary 3.41);

ii. computing $c_{m\kappa+n\lambda}$ costs on average 6.0 $\mathbb{F}_p$-multiplications (cf. Conjecture 3.38);

iii. computing $c_{m\kappa}$ and $c_{n\kappa}$ simultaneously costs on average 6.0 $\mathbb{F}_p$-multiplications (cf. Conjecture 3.40);

iv. computing $c_{m\kappa}, c_{(m-1)\kappa}$ and $c_{(m-2)\kappa}$ simultaneously costs 7.0 $\mathbb{F}_p$-multiplications (cf. Lemma 3.15);

v. computing $c_{m\kappa}$ costs on average 3.0 $\mathbb{F}_p$-multiplications if given $c_{(t+1)\kappa}, c_{t\kappa}$ and $c_{(t-1)\kappa}$, where t is known and $\lg t \approx \frac{1}{2} \lg m$.

In Table 4.1 the number of multiplications in $\mathbb{F}_p$ required by Algorithm 3.37 when applied to XTR is given, both with and without optional steps. Each set of entries is averaged over the same collection of 2^{20} randomly selected t's, n's,

TABLE 4.1. Empirical performance of Algorithm 3.37 when applied to XTR, with $0 < n, m < t$.

multiplications in $\mathbb{F}_p$

$\lceil \lg t \rceil = T$	including steps X4, X9, and X10			without steps X4, X9, and X10		
	average	standard deviation σ	$\sigma/\sqrt{T}$	average	standard deviation σ	$\sigma/\sqrt{T}$
60	$350.01 = 5.83T$	$20.5 = 0.34T$	2.65	$372.89 = 6.21T$	$30.0 = 0.50T$	3.88
70	$410.42 = 5.86T$	$22.2 = 0.32T$	2.65	$437.41 = 6.25T$	$32.6 = 0.47T$	3.89
80	$470.84 = 5.89T$	$23.7 = 0.30T$	2.65	$501.94 = 6.27T$	$34.8 = 0.44T$	3.90
90	$531.21 = 5.90T$	$25.2 = 0.28T$	2.66	$566.36 = 6.29T$	$37.0 = 0.41T$	3.90
100	$591.63 = 5.92T$	$26.5 = 0.27T$	2.65	$630.85 = 6.31T$	$39.1 = 0.39T$	3.91
110	$652.03 = 5.93T$	$27.8 = 0.25T$	2.65	$695.40 = 6.32T$	$41.1 = 0.37T$	3.92
120	$712.39 = 5.94T$	$29.1 = 0.24T$	2.66	$759.87 = 6.33T$	$43.0 = 0.36T$	3.93
130	$772.78 = 5.94T$	$30.2 = 0.23T$	2.65	$824.31 = 6.34T$	$44.6 = 0.34T$	3.92
140	$833.19 = 5.95T$	$31.5 = 0.22T$	2.66	$888.91 = 6.35T$	$46.4 = 0.33T$	3.92
150	$893.66 = 5.96T$	$32.5 = 0.22T$	2.65	$953.34 = 6.36T$	$48.1 = 0.32T$	3.93
160	$953.98 = 5.96T$	$33.6 = 0.21T$	2.66	$1017.79 = 6.36T$	$49.7 = 0.31T$	3.93
170	$1014.42 = 5.97T$	$34.7 = 0.20T$	2.66	$1082.36 = 6.37T$	$51.3 = 0.30T$	3.93
180	$1074.84 = 5.97T$	$35.7 = 0.20T$	2.66	$1146.88 = 6.37T$	$52.7 = 0.29T$	3.93
190	$1135.19 = 5.97T$	$36.6 = 0.19T$	2.66	$1211.34 = 6.38T$	$54.3 = 0.29T$	3.94
200	$1195.58 = 5.98T$	$37.6 = 0.19T$	2.66	$1275.82 = 6.38T$	$55.7 = 0.28T$	3.94
210	$1256.05 = 5.98T$	$38.5 = 0.18T$	2.66	$1340.23 = 6.38T$	$57.1 = 0.27T$	3.94
220	$1316.42 = 5.98T$	$39.5 = 0.18T$	2.66	$1404.75 = 6.39T$	$58.5 = 0.27T$	3.94
230	$1376.87 = 5.99T$	$40.3 = 0.18T$	2.66	$1469.36 = 6.39T$	$59.7 = 0.26T$	3.94
240	$1437.25 = 5.99T$	$41.2 = 0.17T$	2.66	$1533.89 = 6.39T$	$61.1 = 0.25T$	3.94
250	$1497.61 = 5.99T$	$42.0 = 0.17T$	2.66	$1598.22 = 6.39T$	$62.3 = 0.25T$	3.94
260	$1558.00 = 5.99T$	$42.9 = 0.17T$	2.66	$1662.80 = 6.40T$	$63.7 = 0.24T$	3.95
270	$1618.47 = 5.99T$	$43.8 = 0.16T$	2.66	$1727.31 = 6.40T$	$64.9 = 0.24T$	3.95
280	$1678.74 = 6.00T$	$44.5 = 0.16T$	2.66	$1791.85 = 6.40T$	$66.1 = 0.24T$	3.95
290	$1739.17 = 6.00T$	$45.3 = 0.16T$	2.66	$1856.32 = 6.40T$	$67.2 = 0.23T$	3.94
300	$1799.57 = 6.00T$	$46.1 = 0.15T$	2.66	$1920.88 = 6.40T$	$68.4 = 0.23T$	3.95

and m's, with t of the size specified in Table 4.1 and n and m randomly selected from $\{1, 2, \ldots, t-1\}$. For regular double exponentiation $t \approx q$, but $t \approx \sqrt{q}$ for the application in precomputation. It follows from Table 4.1 that inclusion of the optional steps leads to an overall reduction of more than 6% in the expected number of multiplications in $\mathbb{F}_p$.

For the optional steps it is convenient to keep track of the residue classes of d and e modulo 3. These are easily updated if any of the other steps applies, but require a division by 3 if either one of the optional steps is carried out. It depends on the implementation and the platform whether or not an overall saving is obtained by including the optional steps. In most software implementations it will most likely be worthwhile.

It follows that XTR double exponentiation using Algorithm 3.37 is on average faster than the XTR single exponentiation from [**107**] (given in Algorithm 3.16), and more than twice as fast as the previous methods to compute $c_{m\kappa+b\lambda}$ ([**107**, Algorithm 2.4.8 and Theorem 2.4.9] and Algorithm 3.20). An additional advantage of Algorithm 3.37 is that, like Algorithm 3.20, it does not require matrices. These advantages have considerable practical consequences, not only for the performance of XTR signature verification (Section 4.8), but also for the accessibility and ease of implementation of XTR.

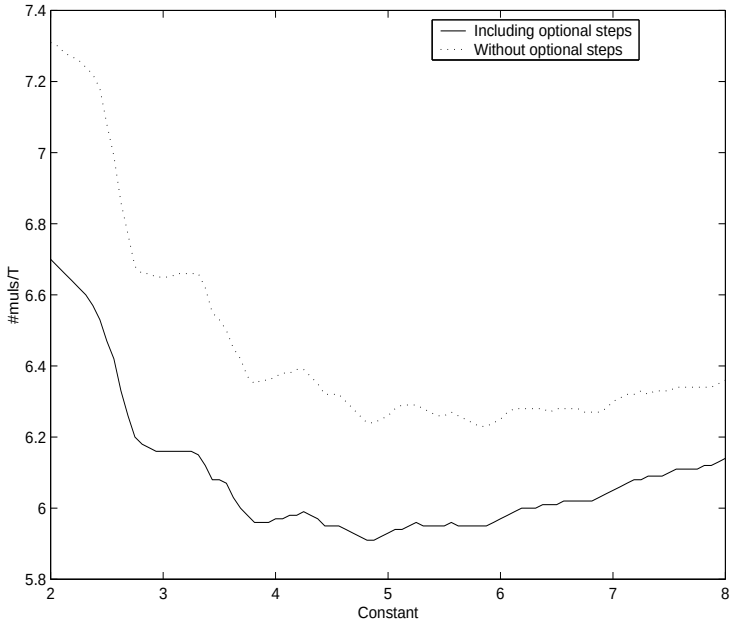


FIGURE 4.1. Dependence of Algorithm 3.37 when applied to XTR on the value of the constant.

In Figure 4.1 the average number of multiplications for $\lceil \log_2 t \rceil = 170$ is given as a function of the value of the constant in rules X1 and X6 given in Table 3.4. The value 4 is close to optimal and convenient for implementation. However, it can be seen from Figure 4.1 that a value close to 4.8 is somewhat better, if one's sole objective is to minimize the number of multiplications in $\mathbb{F}_p$, as opposed to minimizing the overall runtime. The curves in Figure 4.1 were generated for constants ranging from 2 to 8 with stepsize 1/16, per constant averaged over the same collection of 2^{20} randomly selected t's, a's, and b's.

If Algorithm 3.37 is implemented using the slower field arithmetic from [**107**, Lemma 2.1.1] as opposed to the improved arithmetic from Section 4.3, it can on average be expected to require $7.4 \log_2(\max(a, b))$ multiplications in $\mathbb{F}_p$. This is still more than twice as fast as the method from [**107**] (using the slower arithmetic), but more than 20% slower than Corollary 4.13.

Unlike the XTR exponentiation methods from [**107**], different instructions are carried out by Algorithm 3.37 for different input values. This makes Algorithm 3.37 inherently more vulnerable to environmental attacks than the methods from [**107**] (cf. [**107**, Remark 2.3.9]). If the possibility of such attacks is a concern, then utmost care should be taken while implementing Algorithm 3.37.

Further Improved Double Exponentiation. Other small improvements in the running time of Algorithm 3.37 when applied to XTR can be obtained by distinguishing more different cases than implied by Table 3.4. The version presented in Section 3.3.5 represents a good compromise that combines reasonable overhead with decent performance. In practical circumstances the performance of Algorithm 3.37 is on average reasonably close to optimal. Almost 2% can be saved by using Table 4.2 instead. The average cost to compute $c_{m\kappa+n\lambda}$ turns out to be about $5.9\log_2(\max(a,b))$ multiplications in $\mathbb{F}_p$. Omission of X3, X6, and X13 combined with a constant 4 instead of 5.5 in Steps X1 and X9 leads to an almost 1% speedup over Algorithm 3.37 based on Table 3.4.

TABLE 4.2. Improved Rules for Algorithm 3.37

No.	Condition	Substitution (d,e)	Costs
Substitutions if $d \geq e$			
X1	$d \leq 5.5e$	$(e, d-e)$	α
X2	$d \equiv_2 e$	$((d-e)/2, e)$	$\alpha + 2\delta$
X3	$d \leq 6.4e$	$(e, d-e)$	α
X4	$d \equiv_3 e$	$((d-e)/3, e)$	$2\alpha + \tau$
X5	$d \equiv_2 0$	$(d/2, e)$	$\alpha + 2\delta$
X6	$d \leq 7.5e$	$(e, d-e)$	α
X7	$de \equiv_3 2$	$((d-2e)/3, e)$	$2\alpha + \delta + \tau$
X8	$e \equiv_2 0$	$(e/2, d)$	2δ
Substitutions if $e \geq d$			
X9	$e \leq 5.5d$	$(d, e-d)$	α
X10	$e \equiv_2 0$	$(e/2, d)$	2δ
X11	$e \equiv_3 d$	$((e-d)/3, d)$	$2\alpha + \tau$
X12	$de \equiv_3 2$	$(d, (e-2d)/3)$	$2\alpha + \delta + \tau$
X13	$e \leq 7.4d$	$(d, e-d)$	α
X14	$d \equiv_2 1$	$((e-d)/2, d)$	$\alpha + 2\delta$
X15	$e \equiv_3 0$	$(e/3, d)$	$2\alpha + \tau$
X16	$d \equiv_2 0$	$(d/2, e)$	$\alpha + 2\delta$

Fast Precomputation. To apply Corollary 4.13.v the values c_t, c_{t-1}, and c_{t-2} have to be precomputed for some t with $\lg t \approx k/2$. This precomputation costs $3.5k$ $\mathbb{F}_p$-multiplications and requires storage of three elements. The choice $t = p$ leads to a faster precomputation, while only marginally slowing down the preparation of the exponents to be fed to Algorithm 3.37. From the triple (c_{p-2}, c_{p-1}, c_p) two values follow immediately given c, since $c_p = \mathrm{Tr}_2(g^p) = \mathrm{Tr}_2(g)^p = c^p$ and $c_{p-1} = \mathrm{Tr}_2(g^{p-1}) = \mathrm{Tr}_2(g^{p^2}) = c$. According to [**108**, Proposition 5.7], the remaining value c_{p-2} can be computed at the cost of a square-root computation in $\mathbb{F}_p$. Here it is assumed that the public key containing p, q, and c_1 contains an additional single bit of information to resolve the square-root ambiguity. (The statement in [**108**, Proposition 5.7] that this requires a square-root computation in $\mathbb{F}_{p^2}$, as opposed

to $\mathbb{F}_p$, is incorrect. This follows immediately from the proof of [**108**, Proposition 5.7].) Thus, if $p \equiv 3 \bmod 4$ —facilitating square root extraction— recipients of XTR public key data with p and q of the above form can do the precomputation of c_{p-2} at a cost of at most $\approx 1.3 \lg p$ multiplications in $\mathbb{F}_p$, assuming the owner of the key sends the required bit along. The storage overhead (on top of c_1) for $t = p$ is just a single element of $\mathbb{F}_{p^2}$ or even a single bit, as opposed to three elements for $t \approx 2^{\frac{k}{2}}$.

If $p \bmod q \approx \sqrt{q}$, then non-negative a and b of order about $\sqrt{q}$ in Step 1 of Algorithm 3.31 can be found at the cost of a division with remainder. This is, for instance, the case if p and q are chosen as $r^2 + 1$ and $r^2 - r + 1$, respectively, as suggested in [**107**, Section 3.1]. However, usage of such primes p and q is not encouraged in [**107**] because of potential security hazards related to the use of primes p of a 'special form'. As shown in Lemma 2.29 the condition $q|(p^2-p+1)$ is actually sufficient for the existence of a good split, so no special primes are needed.

COROLLARY 4.14. *Given an integer n with $0 < n < q$ and trace values c and c_{p-2}, the trace value c_n can on average be computed in about $3k$ multiplications in $\mathbb{F}_p$ using Algorithm 3.37.*

COROLLARY 4.15. *Given an integer n with $0 < n < q$, a trace values c, and a bit to resolve the ambiguity surrounding c_{p-2}, the trace value c_n can on average be computed in about $3k + 1.3 \lg p$ multiplications in $\mathbb{F}_p$ using Algorithm 3.37.*

The owner of the key must explicitly compute c_{p-2} in order to compute the ambiguity-resolving bit. Thus, the owner cannot take advantage of fast precomputation. This adds a minor cost to the key creation.

The precomputation scheme may also be useful for XTR-ElGamal encryption [**107**, Section 4.2]. In XTR-ElGamal encryption the public key contains two trace values, c_1 and c_k, where k is the secret key. The sender (who does not know k) picks a random integer b, computes c_b based on c_1, computes c_{bk} based on c_k, uses c_{bk} to (symmetrically) encrypt the message, and sends the resulting encryption and c_b to the owner of k. If the sender uses XTR-ElGamal encryption more than once with the same c_1 and c_k, then it is advantageous to use precomputation. In this application *two* precomputations have to be carried out, once for c_1 and once for c_k. The recipient has to compute c_{bk} based on the value c_b received (and its secret k). Because c_b will not occur again, precomputation based on c_b does not make sense for the party performing XTR-ElGamal decryption.

4.6.4. Historical. Carlitz [**39**] investigated sequences of the form $c_n = \alpha^n + \beta^n + \gamma^n$ where α, β, and γ are the roots of the polynomial $1 - ux + vx^2 - x^3$. He showed that $c_{-n}(u, v) = c_n(v, u), c_{m+n} = c_{m-n}c_{-n} - c_m c_n - c_{m-2n}$, and $c_{mn} = c_m(c_n, c_{-n})$. He also noted the similarity with Chebyshev polynomials [**40**].

In 1999 Gong and Harn generalize the LUC cryptosystem, as described in terms of a recurrent relation, to the third degree, reinventing much of the work of Carlitz. The resulting cryptosystem needs to keep track of both positive and negative exponents. As a consequence of this, the compression rate of the GH-cryptosystem is only $\frac{2}{3}$, compared to LUC's $\frac{1}{2}$ or XTR's $\frac{1}{3}$. The exponentiation routines are also

much slower. For key generation, Gong and Harn propose working with an element of order $p^2 + p + 1$. Testing this order typically requires the factorization of $p^2 + p + 1$, which makes key generation [**65**] considerably more troublesome than XTR key generation.

Brouwer et al. [**35**] show that it is possible to represent elements of the subgroup G_{p^2-p+1} in $\mathbb{F}_{p^6}^*$ with only two elements in $\mathbb{F}_p$ instead of the usual six. They use minimal polynomials to achieve this system, which they playfully call LUCKIER. The disadvantage is that for computations the roots of the minimal polynomial have to be determined in order to perform the exponentiation in $\mathbb{F}_{p^6}^*$.

Lenstra and Verheul [**107**] combine the best of both worlds to arrive at XTR, compression and speedup at the same time, without the need to switch back to the extension field at any time. The key generation and related membership testing have been improved by Lenstra and Verheul in later papers [**106, 108**].

4.7. Alternatives

Lenstra and Verheul [**107**] already point out that in principle XTR works for any 6th degree extension. This is worked out in some detail by Lim et al. [**113**], but the main problem, efficient key generation, is not addressed.

Brouwer et al. [**35**] conjecture that for $d = 30$ perhaps an even more efficient representation can be obtained (although in that case too one would face awkward key generation). Bosma et al. [**27**] argue that this conjecture is unlikely to hold and that, if it would hold, the computations would be clumsy. Rubin and Silverberg [**157**] give further evidence against the existence of an efficient representation for $d = 30$.

Generalizing XTR to other congruency classes of p is another possibility to consider. To maintain the efficiency the $\mathbb{F}_{p^2}$-field arithmetic should allow application of the Frobenius endomorphism virtually for free and fast multiplication.

Generalizing the tricks to work in G_{p^2-p+1} directly to other congruency classes requires efficient application of the Frobenius endomorphism to allow splitting the exponent and give free inversion. Using the seventh cyclotomic field this is possible, but is unclear how to speed up the group squaring. Avanzi and Lange consider a quadratic extension of a cubic extension. The Frobenius endomorphism in the quadratic extension is used for free inversion. If the quadratic extension is based on the third or fourth cyclotomic field this gives free inversion (and slightly cheaper squaring as well). Splitting the exponent is not entirely for free, but still worthwhile.

Using other extension degrees is also possible. If the fifth cyclotomic field is used for $\mathsf{G}_{p^2+1} \subseteq \mathbb{F}_{p^4}$ inversions are for free and the exponent can be split, but once more, it is unclear how group squarings can be sped up [**144**].

4.7.1. Quotient Groups. In an anonymous submission [**207**] it was suggested to represent the group $\mathsf{G}_{p+1} \subset \mathbb{F}_{p^2}^*$ by the isomorphic group $\mathbb{F}_{p^2}^*/\mathbb{F}_p^*$. Suppose $p \equiv 2 \bmod 3$, then the same arithmetic as in Section 4.3.1 can be used based on a root γ of $\Phi_3(x)$. Two elements $a_0 + a_1\gamma$ and $b_0 + b_1\gamma$ are in the same congruency class if either both $a_1 = 0$ and $b_1 = 0$ or $a_0/a_1 = b_0/b_1$. As a consequence, elements with $a_1 \neq 0$

can be represented with a single element in $\mathbb{F}_p$, namely a_0/a_1. Compressing a point requires a single field inversion (or more precise, division by a_1). Decompression is for free, simply take $a_0/a_1 + \gamma$ as representative. For the congruency class of elements with $a_1 = 0$ a special symbol must be used.

Arithmetic is based on Lemma 4.2 and the observation that multiplying an element of the form $a_0 + \gamma$ with $b_0 + b_1\gamma$ can be done in $2M + 2D + 3A_1$. Since it can be assumed that the base of the exponent is compressed, this gives a speedup. Computing inverses in the group is basically for free using the Frobenius endomorphism and the order of the group: $(a_0 + a_1\gamma)^{-1} = (a_0 + a_1\gamma)^p = (a_1 + a_0\gamma)$. As a side result, multiplication by the inverse of a compressed element can also be done in $2M + 2D + 3A_1$. This can be summarized by saying that $\alpha = 2.5, \delta = 2, \dot{\alpha} = 2$ and $\nu = 0$, all measured in the number of $\mathbb{F}_p$-multiplications.

Some care has to be taken when applying these figures directly to the runtimes for the algorithms described in Chapter 2. The problem is that the multiplication of two compressed elements does not result in a compressed element, so either extra inversions are needed or some care has to be taken when using $\dot{\alpha}$. For a single exponentiation based on the NAF all is fine; it costs 2.67 $\mathbb{F}_p$-multiplications per exponent bit. A double exponentiation $g^n h^m$ costs 3 $\mathbb{F}_p$-multiplications per exponent bit, if the joint sparse form is used and we ignore the time it takes to compute compressed representations of gh and gh^{-1}. Using the signed sliding window method with $w = 4$ is an alternative, but somewhat involved since not all elements are compressed.

The quotient group approach is especially attractive if precomputation is used. The compression gives both the benefit of a shorter representation and cheaper costs to use the stored elements. For instance, if we use Pippenger's comb with $w = k/10$ and $t = 5$, a single exponentiation based on the 62 precomputed elements will take about $0.59k$ $\mathbb{F}_p$-multiplications. This is more than 30% faster than LUC with precomputation. Whereas extra precomputation for LUC seems ineffective, more precomputation for the quotient group will lead to even faster single exponentiation routines.

4.7.2. Algebraic Tori. Rubin and Silverman [**157**] suggest to use algebraic tori for the compression of elements in G_{p+1} and G_{p^2-p+1}. The same compression rate as LUC respectively XTR is achieved, but without the disadvantages of the trace-methods. In particular, it is possible to decompress to the exact point in G_{p+1} (or G_{p^2-p+1}) and not to one of its conjugates. Rubin and Silverman do not describe the efficiency of the scheme (apart from the compression rate) in any detail. However, for the scheme based on G_{p+1} they derive a formula identical to the one for the quotient group just discussed. For the group based on sextic extensions, it seems the computations have to be performed in G_{p^2-p+1} after 'decompressing'. It is unclear whether this decompression is cheaper than for XTR (where two extra bits can be used to point out the conjugate that is needed).

4.8. Timings

To make sure that the methods introduced in this paper actually work, and to discover their runtime characteristics, all methods based on sextic extension fields were implemented and tested. In this section the results are reported, in such a way that the results can easily and meaningfully be compared to the timings reported in [**107**]. The resulting runtimes are reported in Table 4.3.

For XTR, Algorithm 3.20 was implemented, tested for correctness, and it was confirmed that the speedup over the double exponentiation from [**107**] is negligible. However, implementing Algorithm 3.20 was shown to be significantly easier than it was for the matrix-based method from [**107**]. Thus, Algorithm 3.20 may still turn out to be valuable if Algorithm 3.37 cannot be used. (For instance if one is concerned about side-channel attacks, although this is not very common for double exponentiation.)

XTR routines based on Algorithms 3.37, 3.33, and 3.31 with normal and fast precomputation were implemented as well, and incorporated in cryptographic XTR applications along with the old methods from [**107**].

The traceless method from Section 4.5 was also implemented using the same subgroups $\mathsf{G}_q \subset \mathsf{G}_{p^2-p+1} \subset \mathbb{F}_{p^6}^*$ for 170-bit p and q as XTR which facilitated testing for correctness.

The timings for single exponentiations with precomputation do not include the time needed for the precomputations. These times are given in separate rows.

Each runtime is averaged over 100 random keys and 100 cryptographic applications (on randomly selected data) per key. All times are in milliseconds on a 600 MHz Pentium III NT laptop, and are based on the use of a generic and not particularly fast software package for extended precision integer arithmetic [**103**]. More careful implementation should result in much faster timings. The point of Table 4.3 is however not the absolute speed, but the relative speedup over the methods from [**107**].

The RSA timings are included to allow a meaningful interpretation of the timings: if the RSA signing operation runs x times faster using one's own software and platform, then most likely XTR will also run x times faster compared to the figures in Table 4.3. For each key an odd 32-bit RSA public exponent was randomly selected. 'CRT' stands for 'Chinese Remainder Theorem'. For a theoretical comparison of the runtimes of RSA, XTR, ECC, and various other public key systems at several security levels, refer to [**105**].

4.9. Conclusion

The results for second degree extensions improve previously reported ones, but the resulting exponentiations are less efficient than the LUC exponentiations.

The XTR public key system as published in [**107**] is one of the fastest, most compact, and easiest to implement public key systems. It is shown that it is even faster and easier to implement than originally believed. The matrices from [**107**] can be replaced by the more general iteration from Section 3.3.5. This results in

TABLE 4.3. RSA, old XTR, and new XTR runtimes.

method		key setup	signing	verifying	encrypting	decrypting
1020-bit RSA	with CRT	908 ms	40 ms	5 ms	5 ms	40 ms
	without CRT		123 ms			123 ms
170-bit XTR	old	64 ms	10 ms	21 ms	21 ms	10 ms
	new, no prec.	62 ms	7.3 ms	8.6 ms	15 ms	7.3 ms
	new, with prec.		4.3 ms		8.6 ms	
	precomputation		4.4 ms		8.8 ms	
	fast prec.		1.6 ms		6.0 ms	
170-bit $\mathsf{G}_q \subseteq \mathsf{G}_{p^2-p+1}$	(no prec.)	85 ms	8.9 ms	13 ms	17.8 ms	8.9 ms

60% faster XTR signature applications, substantially faster encryption, decryption, and key agreement applications, and more compact implementations.

The timings also confirm that our new methods for $\mathbb{F}_{p^6}$-subgroup exponentiation are superior to the original XTR and almost competitive with the faster version of XTR described in this chapter. This shows that the main reason to use XTR would no longer be its speed, but mostly its compact —and sometimes inconvenient— representation.

Protocols where traceless methods compare well to LUC and XTR are especially those based on homomorphic ElGamal encryption [**60**] such as Schoenmakers' verifiable secret sharing scheme [**162**]. Another example is Cramer-Shoup encryption [**55**], based on the DDH assumption.

It is also conceivable to use a mixture of trace-based methods and direct computation. Given an element in G_{p+1} or G_{p^2-p+1} the cost of computing the LUC respectively XTR representation is negligible. Going from LUC to G_{p+1} requires a square root computation in $\mathbb{F}_p$, going from XTR to G_{p^2-p+1} can be done by computing the roots of a third degree polynomial over $\mathbb{F}_{p^2}$. In both cases extra information is needed to resolve root ambiguities.

5

Montgomery-Type Representations

This chapter discusses efficient arithmetic on elliptic curves. The emphasis lies on representations where computations are performed on the x-coordinates only. We discuss existing methods and present a new one for curves over binary fields, giving rise to a faster addition routine than previous Montgomery-representations.

As a result a double exponentiation routine is described that requires 8.5 field multiplications per exponent bit, but that does not allow easy y-coordinate recovery. For comparison, we also give a brief update of the survey by Hankerson et al. [**74**] and conclude that, for non-constrained devices, using a Montgomery-representation is slower for both single and double exponentiation than projective methods with y-coordinate.

5.1. Introduction

Since the introduction of elliptic curve cryptography in the mid 1980s, many proposals have been made to speed up the group arithmetic. There are essentially three ways to achieve this: speed up the arithmetic in the underlying field (e.g., binary, prime, optimal extension fields [**9**]), pick a convenient representation of the group elements (e.g., affine, projective, Chudnovsky, 'mixed coordinates' [**48**]), or choose a short addition chain (Chapter 2).

The effects of the three possible choices are certainly not independent as demonstrated by for instance [**9, 48, 86**]. This is in particular the case if the so-called Montgomery representation is used. The Montgomery representation was introduced as part of a speedup of the elliptic curve factoring method [**130**], but has been relatively uncommon in cryptographic applications. It was specifically tailored for curves over large prime fields. (Actually, over large rings $\mathbb{Z}_n^*$, where failure to invert would constitute a factoring success.) For the Montgomery representation the general elliptic curve equation is replaced by

$$E_M : BY^2 = X^3 + AX^2 + X \tag{20}$$

over finite fields of odd characteristic. Because of its intended application in the elliptic curve integer factoring method, the Montgomery representation was specifically designed to speed up the calculation of the x-coordinate of nP, for large integers n and points P on the curve E_M. Montgomery's representation is characterized not so much by the particular form of the curve-equation, but mostly by the facts that to add two points their difference must be known and that the y-coordinate is absent. Furthermore, the order of the elliptic curve group must be divisible by

4 [**140**]. These facts have to be taken into account when one tries to take advantage of the fast Montgomery representation based computation of nP in a cryptographic context. For instance, the divisibility by 4 rules out the so-called NIST curves [**136**].

It is well known that for most cryptographic protocols a y-coordinate is not really needed; for instance, in Diffie-Hellman it adds only a single bit and ECDSA [**77**] can be run with just x-coordinates. Nevertheless, if the y-coordinate of some point P is needed, it can be computed efficiently if P's x-coordinate is known along with the x-coordinate of some other point Q and both the x and y coordinates of $P - Q$. Whether or not these data are available depends on the way P is computed. In the Montgomery representation, and assuming P is the result of a scalar multiplication, P is computed using a second order recurrence in the x-coordinate corresponding to a Lucas chain (Chapter 3), because to add two points their difference must be known.

For elliptic curves over fields of odd characteristic a generalization of Montgomery's result is known that is valid for general Weierstrass curves, including the NIST curves [**34, 76**]. It leads to relatively slow scalar multiplication. Optimization of those formulae leads to Montgomery's results with the same restriction on the curve group order, but slightly more freedom in the curve equation.

For elliptic curves over fields of characteristic two, the traditional Montgomery representation based on the curve equation E_M does not work, because the curve isomorphism requires a division by 2. Adaptation of Montgomery's representation to fields of characteristic two based on the ordinary shortened Weierstrass form for non-supersingular curves is considered in [**3, 115, 192**], resulting in a reasonably fast single exponentiation routine without precomputation.

By introducing an alternative curve equation we further optimize the Montgomery method for curves over binary fields, saving one finite field multiplication for general point addition relative to earlier work. Compared to [**115**] this leads to a speedup of about 15% for double exponentiation. Furthermore, we investigate the consequences for single exponentiation with precomputation, y-coordinate recovery, and two simultaneous exponentiations. The new methods apply to all non-supersingular curves over $\mathbb{F}_{2^l}$, irrespective of the group order.

The Montgomery representation is said to have three possible advantages: it is fast, requires only few registers in memory and can serve as a hedge against timing and power analysis attacks. From our results and comparison with other work, we conclude that in the binary case the Montgomery representation is not as fast as regular methods, for neither single nor double exponentiation. Furthermore, the fastest Montgomery representation approach to either type of exponentiation uses continued fraction based Lucas chains; as a consequence the protection against timing and power analysis attacks is lost. So, despite the fact that our results improve on previous results in this area, we conclude that the use of Montgomery representations for elliptic curves over binary fields can hardly be recommended, unless our results can be improved upon. Only if memory usage or timing and power analysis are of serious concern, the Montgomery representations regain their attractiveness.

In Section 5.2 a brief overview of elliptic curves is given, based on the comprehensive work by Blake et al. [**17**]. In Section 5.3 the Montgomery representations are considered in their full generality. Section 5.4 focuses on binary fields. Existing representations are reviewed and a new one is introduced, that requires one field multiplication fewer for a point addition. A comparison is made with the traditional way of doing elliptic curve arithmetic by means of projective coordinates. In Section 5.5 curves over fields of odd characteristic are considered. These results are mostly known. In Section 5.6 we present our conclusions.

5.2. Elliptic Curves

5.2.1. Curve Definition. An elliptic curve over a finite field $\mathbb{F}$ is the set of points $(X, Y) \in \mathbb{F}^2$ satisfying the long Weierstrass equation

$$E : Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6 \tag{21}$$

together with a point at infinity, denoted $\mathcal{O}$. The coefficients $a_i, i \in \{1, 2, 3, 4, 6\}$, are taken from the field of definition $\mathbb{F}$. The points on an elliptic curve form a finite abelian group under the addition operation also known as the chord-tangent process. The point at infinity, $\mathcal{O}$, serves as group identity and the negation of a point (X, Y) is given by $(X, -Y - a_1X - a_3)$.

In the literature several constants related to the a_i are defined:

$$\begin{aligned}
b_2 &= a_1^2 + 4a_2 \ ; \\
b_4 &= a_1a_3 + 2a_4 \ ; \\
b_6 &= a_3^2 + 4a_6 \ ; \\
b_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \ ; \\
c_4 &= b_2^2 - 24b_4 \ ; \\
c_6 &= -b_2^3 + 36b_2b_4 - 216b_6 \ ; \\
\Delta &= -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6 \ ; \\
j &= c_4^3/\Delta \ .
\end{aligned}$$

Here Δ is called the discriminant and the j stands for j-invariant. It can easily be verified that the b's satisfy

$$4b_8 = b_6b_2 - b_4^2 \ .$$

A curve is singular iff $\Delta = 0$; henceforth we will assume $\Delta \neq 0$. The j-invariant characterizes (group) isomorphism classes over the algebraic closure $\bar{\mathbb{F}}$ of the ground field: Two curves are isomorphic over $\bar{\mathbb{F}}$ if and only if their j-invariants are the same. A curve is supersingular iff $\Delta \neq 0$ and $j = 0$.

We assume we are working in a cyclic subgroup of size q with generator P and $\lg q \approx k$. As customary we use additive notation for the group operation.

5.2.2. Curve Arithmetic. Let P_i for $0 < i \leq 5$ be points on the curve and assume that $P_3 = P_1 + P_2, P_4 = P_1 - P_2$, and $P_5 = 2P_1$. Moreover, we assume that $\mathcal{O}$ is not among the P_i. This allows us to write $P_i = (X_i, Y_i)$ in affine coordinates or $P_i = (x_i, y_i, z_i)$ in projective coordinates, usually with $X_i = x_i/z_i$ and $Y_i = y_i/z_i$. Note that upper case characters are used for affine coordinates, and lower case ones for projective coordinates. The following relations hold (cf. [**17**, Lemma III.2]):

$$X_3 = (\frac{Y_1 - Y_2}{X_1 - X_2})^2 + a_1(\frac{Y_1 - Y_2}{X_1 - X_2}) - X_1 - X_2 - a_2 \tag{22}$$

$$X_4 = (\frac{Y_1 + Y_2 + a_1X_2 + a_3}{X_1 - X_2})^2 + a_1(\frac{Y_1 + Y_2 + a_1X_2 + a_3}{X_1 - X_2}) - X_1 - X_2 - a_2 \tag{23}$$

$$X_5 = \frac{X_1^4 - b_4X_1^2 - 2b_6X_1 - b_8}{4X_1^3 + b_2X_1^2 + 2b_4X_1 + b_6} \ . \tag{24}$$

5.3. The Montgomery Representation

From (24) it follows that it suffices to know the x-coordinate of P to compute the x-coordinate of $2P$. Montgomery [**130**] observed that if the special curve equation (20) is used, the product of the x-coordinates of P_1+P_2 and P_1-P_2 can be expressed using only the x-coordinates of P_1 and P_2. The occurrences of the y-coordinates in (22) and (23) cancel after applying the curve equation. If we consider a curve over an arbitrary finite field and satisfying the long Weierstrass equation (21), we can write the aforementioned product of x-coordinates as

$$X_3X_4 = (-b_8 - b_6(X_1 + X_2) - b_4X_1X_2 + X_1^2X_2^2)/(X_1 - X_2)^2 \ . \tag{25}$$

Interestingly, the sum of the x-coordinates of $P_1 + P_2$ and $P_1 - P_2$ can also be expressed using only the x-coordinates of P_1 and P_2.

$$X_3 + X_4 = (b_6 + b_4(X_1 + X_2) + X_1X_2(b_2 + 2(X_1 + X_2)))/(X_1 - X_2)^2 \ . \tag{26}$$

To denote the difference between the two approaches we speak of the additive Montgomery method if the group addition is based on $X_3 + X_4$ and of the multiplicative Montgomery method if it is based on X_3X_4. The terminology additive and multiplicative method is also used by Izu and Takagi [**76**]. Note that the formula for point doubling is independent of the choice for the multiplicative or additive method. (Although there is some interaction when the formulae are optimized.)

5.3.1. Recovery of the y-Coordinate. The curve equation provides a way to determine the y-coordinate of a given point using a square root computation. This method is relatively expensive and one still needs to address the square root ambiguity. Lopez and Dahab present an alternative [**115**, Lemma 3]. If two points P_1 and P_2 are given by x-coordinate only and their difference P_4 is fully specified, either unknown y-coordinate can be retrieved in a small number of field multiplications. Moreover, there is no square root ambiguity. The method from Lopez and Dahab to recover the y-coordinate is described in more detail below (slightly generalized).

Assuming that $P_4 \neq -P_4$ and given X_1, X_2, X_4 and Y_4, it is possible to determine Y_2 efficiently. From formula (22) it follows how to determine Y_1 if X_1, X_2, X_3

and Y_2 are given, by using the curve equation (21) to get rid of the quadratic term Y_1^2. However, since the values $(P, Q, P-Q)$ have the same additive relation to each other as the values $(P+Q, P, Q)$, the desired result follows from a suitable re-indexing:

(27)
$$Y_1 = \frac{2a_6 - X_1(X_2{-}X_4)^2 + a_4(X_2{+}X_4) + X_2X_4(2a_2{+}X_2{+}X_4) - (a_3{+}a_1X_2)Y_4}{a_3 + a_1X_4 + 2Y_4} \ .$$

Note that $a_3 + a_1X_4 + 2Y_4 = 0$ iff $Y_4 = -Y_4 - a_1X_4 - a_3$ or $P_4 = -P_4$, which we assumed not to be the case. In practical applications P_4 will typically be the generator of a (sub)group for which the DLP is assumed to be hard, so it will certainly have a higher order than two.

5.4. Curves over Binary Fields

5.4.1. Traditional Methods. Before discussing the Montgomery representation for binary fields, a word on more traditional methods. Our point of departure is the overview by Hankerson et al. [**74**].

For non-supersingular curves over fields of binary characteristic the long Weierstrass equation is often replaced by the following short Weierstrass equation

$$E : Y^2 + XY = X^3 + a_2X^2 + a_6 \ . \tag{28}$$

For every curve of the form (21) there is an isomorphic curve of the form (28). Moreover, if the extension degree k is odd, a_2 in (28) can be taken either 0 or 1. Hence multiplications by a_2 may be neglected. A curve of the form (28) has discriminant a_6 and j-invariant $1/a_6$.

Curve Representation. Skewed projective coordinates $(x/z, y/z^2)$ are the most efficient curve representation known [**86, 114**]. Using skewed projective coordinates, a general addition costs 14 field multiplications. However, if one of the points is given in affine coordinates, this drops to 9 field multiplications. Doubling a point costs 4 field multiplications. King [**86**] notes that, if the point resulting from an addition is subsequently doubled, a multiplication can be saved at the cost of a squaring. For a large class of popular exponentiation routines, this effectively reduces the cost of an addition to 8 field multiplications.

Exponentiation (or Scalar Multiplication). Hankerson et al. describe exponentiation routines for three different settings: single exponentiation with a fixed base; single exponentiation with an arbitrary base and double exponentiation with one base fixed and the other arbitrary. This choice seems motivated by the facts that signature generation is a fixed base exponentiation and signature verification is a double exponentiation with one base fixed. However, with the introduction of fast point counting algorithms it becomes more realistic to deviate from standardized curves, which is why we also consider double exponentiations with both bases arbitrary. More details on the exponentiation methods can be found in Chapter 2.

For single exponentiation Hankerson et al. [**74**] propose a signed sliding window (SSW) method with window size 4 (using the Montgomery representation is reported

TABLE 5.1. Expected number of field multiplications for point multiplication given a k-bit exponent, assuming $I \approx 10M$.

Type	Method	Given in [**74**]	Improvement
single	SSW, $w = 4$	$5.8k + 60$	$5.6k + 60$
fixed single	comb, $w = k/8$, $t = 4$	$3.11k + 8$	$2.88k + 9$
semi-fixed double	comb+Montgomery-repr.	$9.11k + 40$	n.a.
semi-fixed double	interl. SSW, $w = 4$	n.a.	$7.24k + 60$
double	interl. SSW, $w = 4$	n.a.	$7.24k + 108$
double	JSF	n.a.	$8k + 36$

to be faster). This results in approximately 1 point doubling and $\frac{1}{5}$ point additions per exponent bit. Moreover, 3 points have to be precomputed, namely $3P, 5P$, and $7P$. This has to be done affinely and costs 8 multiplications and 4 inversions. If the base is fixed, Hankerson et al. propose a fixed base comb of size 4. An exponentiation will cost $\frac{1}{4}$ point doublings and $\frac{15}{64}$ point additions per exponent bit on average. The number of precomputed points is 14. Note that this method does not exploit cheap point negation.

The double exponentiation routine presented assumes one base is fixed and simply consists of two separate single exponentiation routines and multiplying the result (using a fixed base comb and Montgomery respectively). A double exponentiation without any fixed bases is not addressed. However, using the Joint Sparse Form it will require 1 point doubling and $\frac{1}{2}$ point additions per bit of the longest exponent. Interestingly, this method already outperforms the double exponentiation routine with fixed base given by Hankerson et al. Möller [**127**] proposes to interleave two windowed NAF routines thereby saving one set of point doublings. For window size 4 this results in 8 points to be precomputed affinely and 1 point doubling and $\frac{2}{5}$ point additions per exponent bit. Note that King's improvement does not apply in approximately $\frac{1}{25}$ of the additions. Of the 8 points to be precomputed, half can be done in advance if the base is fixed.

Table 5.1 summarizes these results. We give the number of field multiplications for a k-bit exponent, according to [**74**] and with inclusion of the known speedups just described. We use the same ratio as Hankerson et al., namely that one inversion costs the same as 10 multiplications. The extra costs needed at the end to convert back to affine coordinates are also included. For $k = 163$, compare with [**74**, Table 6].

5.4.2. Optimizing Montgomery. The first thing to note when optimizing the Montgomery representation for curves over binary fields is the simplification that arises when working with characteristic 2. Relevant examples are the following three:

$$\begin{aligned} b_8 &= a_1^2 a_6 + a_1 a_3 a_4 + a_2 a_3^2 + a_4^2 \ ; \\ \Delta &= a_1^4 b_8 + a_3^4 + a_1^3 a_3^3 \ ; \\ j &= a_1^{12}/\Delta \ . \end{aligned}$$

Note that a non-singular curve is supersingular iff $a_1 = 0$.

The doubling formula (22) also simplifies considerably, giving

$$X_5 = \frac{X_1^4 + a_1 a_3 X_1^2 + b_8}{(a_1 X_1 + a_3)^2} \ . \tag{29}$$

For affine non-supersingular curves the work is minimized for $a_3 = 0$ and $a_1 = 1$, requiring $2S + I + M + D + 2A_1$, basically an inversion and a multiplication (or one division). Projectively we get

$$\frac{x_5}{z_5} = \frac{x_1^4 + a_1 a_3 x_1^2 z_1^2 + b_8 z_1^4}{a_1 x_1^2 z_1^2 + a_3 z_1^4} \ . \tag{30}$$

which costs at most $5M+5S+3A_1$. If $a_3 = 0$ and either $a_1 = 1$ or $b_8 = a_1^2 a_6 + a_4^2 = 1$, this reduces to $2M + 3S + A_1$.

The recovery of the y-coordinates is also considerably simplified. For both shortened Weierstrass as our proposed representation $a_3 = 0$ and $a_4 = 0$. If we furthermore assume projective coordinates are used, then Lopez and Dahab [**115**] show that (27) simplifies to

(31a)

$$X_1 = \frac{a_1 z_4 x_4 z_2 x_1}{a_1 z_4 x_4 z_2 z_1} \ ;$$

(31b)

$$Y_1 = y_4 + \frac{(x_2 z_4 + x_4 z_2)((x_1 z_4 + x_4 z_1)(x_2 z_4 + x_4 z_2) + z_1 z_2 x_4^2 + a_1 z_1 z_2 z_4 y_4)}{a_1 z_1 z_2 x_4 z_4} \ .$$

Simultaneous recovery of X_1 and Y_1 will therefore cost at most 15 multiplications and 1 inversion. If $z_4 = 1$, this reduces to 11 multiplications and 1 inversion. A further reduction to 10 multiplications and 1 inversion is achieved if $a_1 = 1$, which is the case for the shortened Weierstrass form.

We will now review known methods of computations in the elliptic curve groups over binary fields without using y-coordinates. Interestingly, these methods are all based on a relationship for $X_3 + X_4$. Next, we analyse methods based on a relationship for $X_3 X_4$.

5.4.3. Additive Montgomery Formulae. Agnew et al. [**3**] mention computing with x-coordinates only for non-supersingular curves over large extension fields of characteristic two. By using the curve equation (21) it is possible to rewrite (22) as

$$X_3 = \frac{Y_1(a_1 X_2 + a_3) + Y_2(a_1 X_1 + a_3) + (X_1 + X_2)(a_4 + X_1 X_2)}{(X_1 + X_2)^2} \ . \tag{32}$$

For X_4 a similar formula can be obtained by replacing Y_2 in (32) with $Y_2 + a_1 X_2 + a_3$, the y-coordinate of $-P_2$. This shows that

$$X_3 + X_4 = \frac{(a_1 X_1 + a_3)(a_1 X_2 + a_3)}{(X_1 + X_2)^2} \ . \tag{33}$$

For the shortened Weierstrass form the above is simplified by setting $a_1 = 1$ and $a_3 = 0$. The projective version presented in [**3**] based on these formulae is incorrect.

In 1993 the Montgomery representation is also mentioned in an article by Menezes and Vanstone [**123**]. Here the affine version for supersingular curves over fields of binary characteristic is given. In this case $a_1 = 0$ and $a_3 = 1$, so (33) simplifies to

$$X_3 + X_4 = \frac{1}{(X_1 + X_2)^2} \ . \tag{34}$$

Since squarings can be neglected, an addition costs a single inversion. This affine version is reported to present an improvement in storage requirements, but at a considerable expense of speed.

In 1999 Lopez and Dahab [**115**] and Vanstone et al. [**192**] independently propose correct projective versions for non-supersingular curves. Both proposals are based on the short Weierstrass form. In both works the following formulae are given, easily verified by (33) and (29):

$$\begin{aligned} \frac{x_3}{z_3} &= \frac{x_4(x_1z_2 + x_2z_1)^2 + z_4(x_1z_2)(x_2z_1)}{z_4(x_1z_2 + x_2z_1)^2} \ ; \\ \frac{x_5}{z_5} &= \frac{x_1^4 + a_6z_1^4}{x_1^2z_1^2} \ . \end{aligned} \tag{35}$$

In both works emphasis is put on an addition with a fixed difference P_4, which allows setting z_4 in (35) to 1. Such a fixed difference addition takes 4 field multiplications and 1 squaring (for some reason Vanstone et al. report 2 squarings). It seems that for an ordinary addition one needs 6 field multiplications and 1 squaring. A point doubling takes 2 multiplications and, if $a_6^{1/4}$ is precomputed, 3 squarings (and here Lopez and Dahab use 5 squarings, which is more than needed even without precomputation of $a_6^{1/4}$). It is also noted that having a small $a_6^{1/4}$ can reduce the cost of multiplication with $a_6^{1/4}$ considerably, and therefore of a point doubling. A similar argument holds for fixed difference addition if x_4 is small.

In both works affine versions are also presented, but not worked out in full since the inversions seem to deter. It is said that a point addition takes 2 multiplications, a squaring and an inversion, whereas a doubling takes one multiplication fewer.

5.4.4. Multiplicative Montgomery Formulae. Originally, Montgomery derived his formula for curves over large prime characteristic by multiplying X_3 and X_4, not by considering their sum. Not surprisingly, a multiplicative version of Montgomery's trick also proves possible for curves over binary characteristic. For a curve, that is non-singular and has coefficient $a_3 = 0$, one obtains:

$$\begin{aligned} \frac{x_3x_4}{z_3z_4} &= \frac{x_1^2x_2^2 + b_8z_1^2z_2^2}{(x_1z_2 + x_2z_1)^2} \ ; \\ \frac{x_5}{z_5} &= \frac{x_1^4 + b_8z_1^4}{(a_1x_1z_1)^2} \ . \end{aligned}$$

In the short Weierstrass form a_1 will be 1, but b_8 can be any field element. Hence, computing x_3 and z_3 will take 6 field multiplications by computing $(x_1z_2 + x_2z_1)$ as $(x_1 + z_1)(x_2 + z_2) - x_1x_2 - z_1z_2$. The number of multiplications is one less if P_4 is fixed and $z_4 = 1$. Setting $b_8 = 1$ will reduce these costs to 5 respectively 4 field multiplications, while at the same time keeping the costs for a point doubling the same. As an alternative, one could consider using the representation (x, z, xz). This requires the same number of multiplications though, so should therefore not be recommended.

If $a_3 = 0$, then $b_8 = a_1^2a_6 + a_4^2$. Hence, $b_8 = 1$ can be achieved by setting $a_4 = 0$ and $a_6 = 1/a_1^2$. We therefore propose working on elliptic curves of the form

$$E : Y^2 + a_1XY = X^3 + a_2X^2 + 1/a_1^2 \ , \tag{36}$$

where a_1 and a_2 are in $\mathbb{F}_{2^l}$. To ensure we do not exclude any interesting curves we present the following lemma.

LEMMA 5.1. *Any non-supersingular curve over $\mathbb{F}_{2^l}$ is isomorphic to a curve over $\mathbb{F}_{2^l}$ of the form (36).*

Proof: Recall that all non-supersingular curves have a representation of the form (28), having j-invariant $1/a_6$. A curve of the form (36) has j-invariant equal to a_1^8. Squaring is a permutation on $\mathbb{F}_{2^l}$ whence all elements have a unique eighth root in $\mathbb{F}_{2^l}$. Set $a_1 = a_6^{-1/8}$ and let $s \in \mathbb{F}_{2^l}$. Consider the admissible change of variables given by

$$\begin{aligned} x &= X/a_1^2, \\ y &= sX/a_1^2 + Y/a_1^3 \ . \end{aligned}$$

This gives an isomorphism over $\mathbb{F}_{2^l}$ of a curve given by (28) and a curved defined by

$$1/a_1^2 + a_1^2(a_2 + s + s^2)x^2 + x^3 = a_1xy + y^2 \ . \tag{37}$$

It is easily verified that (37) is of the form (36). *Q.E.D.*

5.4.5. Exponentiation, also known as Scalar Multiplication. Table 5.2 contains an overview of the various Lucas chain methods described in Chapter 3 when applied to the Montgomery representation. Once again, α stands for a point addition and δ for a point doubling. The notation $\dot{\alpha}_3$ is used for a point addition with a fixed difference, since these are cheaper. *Old* refers to [**115**] and [**192**]'s additive version, *New* refers to the multiplicative version presented in Section 5.4.4. The column Y denotes whether easy y-coordinate recovery using (31) is possible or not.

5.5. Prime Fields

5.5.1. Traditional Methods. An elliptic curve over a prime field $\mathbb{F}_p, p > 3$ can be represented using the short Weierstrass form

$$Y^2 = X^3 + a_4X + a_6 \ . \tag{38}$$

TABLE 5.2. Overview of asymptotic costs of scalar multiplication based on Montgomery representation in $\mathbb{F}_{2^l}$-multiplications per exponent bit.

	Y	General	Old	New
Group arithmetic				
Doubling		δ	2	2
Addition		α	6	5
Mixed Addition		$\dot{\alpha}_3$	4	4
Single exponentiation				
Binary	Yes	$\dot{\alpha}_3 + \delta$	6	6
Montgomery	No	$1.5\alpha + 0.25\delta$	9.5	8
Precomp. Montgomery	No	$0.75\alpha + 0.25\delta$	5	4.3
Double exponentiation				
Akishita	Yes	$2\frac{1}{4}\dot{\alpha}_3 + \frac{3}{4}\delta$	10.5	10
Montgomery	No	$1.5\alpha + 0.5\delta$	10	8.5
Twofold exponentiation				
Montgomery	No	$1.5\alpha + 0.5\delta$	10	8.5

TABLE 5.3. Operation counts for elliptic curve point addition and doubling. A = affine, P = standard projective, J = Jacobian, and C = Chudnovsky.

Doubling, δ		General Addition, α		Mixed Coordinates, $\dot{\alpha}$	
$2A \to A$	$I + 2M + 2S$	$A + A \to A$	$I + 2M + 2S$	$J + A \to J$	$8M + 3S$
$2P \to P$	$7M + 3S$	$P + P \to P$	$12M + 2S$	$J + C \to J$	$11M + 3S$
$2J \to J$	$4M + 4S$	$J + J \to J$	$12M + 4S$	$C + A \to C$	$8M + 3S$
$2C \to C$	$5M + 4S$	$C + C \to C$	$11M + 3S$		

Such a curve has j-invariant $6912a_4^3/(4a_4^3 + 27a_6^2)$. For a large class of curves it is possible to use $a_4 = -3$ (for example the NIST-curves).

Brown et al. [**36**] give an overview of efficient arithmetic on elliptic curves over prime fields. If the group arithmetic is performed affinely, a point addition costs $2M + S + I + 5A_1 + 3D$ and a point doubling $2M + 2S + I + 3A_1 + 4D$. (This is a rare case where $\delta > \alpha$). The high costs of inversions are the reason to examine projective coordinates.

The standard projective coordinates are (x, y, z) subject to $X = \frac{x}{z}$ and $Y = \frac{y}{z}$. Generally faster are Jacobian coordinates (x, y, z) satisfying $X = \frac{x}{z^2}$ and $Y = \frac{y}{z^3}$. Keeping track of z^2 and z^3 can save some work, giving rise to Chudnovsky Jacobian coordinates. Cohen et al. [**48**] consider adding points with different coordinate representations. This technique is known as mixed coordinate representation. The main application is performing the precomputation affinely and the rest projectively. Table 5.3 is a reproduction of [**36**, Table 5]. The modular reductions and modular additions are not counted, moreover curves with $a_4 = -3$ are assumed.

Exponentiation. The exponentiation routines described in Chapter 2 can be used to perform a scalar multiplication. Table 5.4 contains an overview of the costs for the most common exponentiations. based on Assumption 1.3. We would like to point out that the application of Assumption 1.3 has the effect of seemingly halving the number of operations required compared to the existing literature (for instance Brown et al., count $M = 1$ and $S = 0.85$). For comparison with the cryptosystems described in the previous chapter this deviation is reasonable. The moduli used in elliptic curve cryptography allow fast reduction, whereas the use of such moduli is not recommended for the cryptosystems discussed in Chapter 4. Especially for the cryptosystems based on $\mathbb{F}_{p^6}$ the comparison is meaningful, since moduli of comparable length can be used.

Quite often the projective point at the end of the computation is brought back into affine format. This costs an inversion and some multiplications. If an inversion costs as much as 80 multiplications and 160-bit curves are used, this effectively adds half a multipication per exponent bit (which is significant). We do not count these inversions for three reasons: all projective methods require one; in some protocols the projective point actually suffices; and the relative cost of an inversion compared to a multiplication greatly differs depending on who you listen to.

For an ordinary single exponentiation Brown et al. propose a signed sliding window method with window size $w = 4$ using the Chudnovsky-Jacobian representation for the precomputation and the Jacobian representation for the main computations. The costs for this method given a random k-bit exponent are $79M + 25S$ for the precomputation and on average $\approx (4 + \frac{11}{6})kM + (4 + \frac{3}{6})kS$ for the processing of the exponent. Under Assumption 1.3 this sums up to $47 + 4.3k$ measured in $\mathbb{F}_p$-multiplications. Using window size $w = 3$ leads to an average runtime of $21.4+4.5k$. For the sake of comparison, the (binary) NAF based on Jacobian coordinates (and an affinely represented base), takes $4.8k$ $\mathbb{F}_p$-multiplications on average for a single exponentiation.

If inverses are not too expensive, it can be advantageous to mix even more coordinates. First use Chudnovsky-Jacobion coordinates for the precomputation. Use Montgomery's simultaneous inversion trick [**45**, Algorithm 10.3.4] to obtain the affine representations for these points. This costs roughly $6M + S$ per point plus one inversion. For instance, using a window size $w = 4$ will lead to a runtime of $\approx 70.1 + 4.0k$ $\mathbb{F}_p$-multiplications plus one inversion (window size $w = 3$ gives $\approx 31.3 + 4.2k$). If $k = 192$ and the cost of an inversion satisfies $15M + 2S = 8.1 < I < 25$ we see that $31.3 + 4.2 \cdot 192 + I < 47 + 4.3 \cdot 192$, so the mixture of three representations outperforms the mixture based on only two representations. A similar use of Montgomery's simultaneous inversion trick is proposed by Okeya and Sakurai [**141**].

For fixed-base single exponentiation a comb method can be used. Brown et al. suggest a comb method based on $w = \frac{k}{8}$ and $t = 4$. In this case 30 points need to be precomputed (and stored affinely). Any subsequent exponentiation takes $\approx \frac{15\dot{\alpha}+8\delta}{64}k - \dot{\alpha} - \delta$ for a k-bit exponent, where $\dot{\alpha}$ represents the cost of adding a Jacobian

TABLE 5.4. Expected number of field multiplications for point multiplication given a k-bit exponent.

Type	Method	Implied by [**36**]
single	signed sliding window, $w = 4$	$4.3k + 47$
fixed single	fixed base comb, $w = k/8, t = 4$	$1.54k - 7.8$
semi-fixed double	comb+window	$5.9k + 47$
semi-fixed double	interleaved signed sliding window with $w = 4$ and $w = 6$	$4.9k + 47$
double	JSF (windowed)	$5k + 49$

and an affine point and δ that of a Jacobian point doubling. Assumption 1.3 leads to the simplification $1.54k - 7.8$, measured in the number of $\mathbb{F}_p$-multiplications.

Brown et al. only consider a double exponentiation with one of the bases fixed. In this case the exponentiations are performed separately, one using the comb method and one using the window method. The total costs of this method are $(1.54k - 7.8) + (4.3k + 47) + 7.2 \approx 5.9k + 47$. The windowed JSF will in general be faster and has the advantage that no base needs to be fixed. The precomputation can mostly be based on mixed Chudnovsky and affine coordinates; the main computation uses mixed Jacobian and Chudnovsky coordinates. Interleaving two signed sliding window methods seems slightly faster if one of the bases is fixed and large windows for this base have been affinely precomputed.

5.5.2. Optimizing Montgomery. Montgomery [**130**] considers curves given by the somewhat unusual curve equation (20). He shows that a point doubling can be computed by

$$\frac{x_5}{z_5} = \frac{(x_1 + z_1)^2(x_1 - z_1)^2}{4x_1z_1((x_1 - z_1)^2 + ((A + 2)/4)(4x_1z_1))} , \tag{39}$$

which costs $3M + 2S + 5D + 4A_1$ exploiting that $4x_1z_1 = (x_1 + z_1)^2 - (x_1 - z_1)^2$ and assuming $(A + 2)/4$ is precomputed. Point addition is based on

$$\frac{x_3x_4}{z_3z_4} = \frac{(x_1x_2 - z_1z_2)^2}{(x_1z_2 - z_1x_2)^2} . \tag{40}$$

Using a Karatsuba-like technique, this can be rewritten as

$$\frac{x_3}{z_3} = \frac{z_4((x_1 - z_1)(x_2 + z_2) + (x_1 + z_1)(x_2 - z_2))^2}{x_4((x_1 - z_1)(x_2 + z_2) - (x_1 + z_1)(x_2 - z_2))^2} ,$$

from which it follows that a point addition costs $4M + 2S + 6D + 6A_1$. Montgomery notes that by storing $x_i + z_i$ and $x_i - z_i$ the number of additions reduces to 4 (without affecting the number of additions needed for a point doubling). If the point P_4 is given affinely, that is $z_4 = 1$, the costs reduce to $3M + 2S + 6D + 6A_1$.

Not all curves have a Montgomery-representation. Okeya et al. [**140**] give necessary and sufficient conditions for a curve in the shortened Weierstrass form (38) to have a Montgomery-representation. The equation $X^3 + a_4X + a_6 = 0$ should have at

TABLE 5.5. Overview of average costs per bit of scalar multiplication based on Montgomery representation.

	Y	General	Montgomery	Weierstrass
Group arithmetic				
Doubling		δ	2.1	3.1
Addition		α	2.6	4.6
Mixed Addition		$\dot{\alpha}_3$	2.1	4.1
Single exponentiation				
Binary	Yes	$\dot{\alpha}_3 + \delta$	4.2	7.2
Montgomery	No	$1.5\alpha + 0.25\delta$	4.4	7.7
Precomp. Montgomery	No	$0.75\alpha + 0.25\delta$	2.5	4.2
Double exponentiation				
Akishita	Yes	$2\frac{1}{4}\dot{\alpha}_3 + \frac{3}{4}\delta$	6.3	11.6
Montgomery	No	$1.5\alpha + 0.5\delta$	4.9	8.4
Twofold exponentiation				
Montgomery	No	$1.5\alpha + 0.5\delta$	4.9	8.4

least one root in $\mathbb{F}_p$, corresponding to the point $(X_0, 0)$ on the curve, and $3X_0 + a_4$ should be a quadratic residue modulo p. The point $(X_0, 0)$ has order 2 which implies that Weierstrass curves without a point of order 2 cannot be represented by (20). On a Weierstrass curve the x-coordinate of a point of order 2 satisfies

$$b_6 + 2b_4X_1 + b_2X_1^2 + 4X_1^3 = 0 \ .$$

The x-coordinate of a point of order 4 is a root of the polynomial (in x)

$$(-b_4^3+b_2b_4b_6-4b_6^2)+(4b_2b_8-4b_4b_6)X_1+40b_8X_1^2+40b_6X_1^3+20b_4X_1^4+4b_2X_1^5+8X_1^6 \ .$$

The point doubling formula (24) is very similar to Montgomery's formula (39) if $b_6 = 0$, $b_4 = 2$, and $b_8 = -1$, namely

$$\frac{x_5}{z_5} = \frac{(x_1^2 - z_1^2)^2}{x_1z_1(4x_1^2 + b_2x_1z_1 + 4z_1^2)} \ .$$

In this case the addition formula is exactly identical to Montgomery's. Although representations with $b_6 = 0$, $b_4 = 2$, and $b_8 = -1$ might at first look more general than Montgomery's, this is not the case.

One way to achieve $b_6 = 0$, $b_4 = 2$, and $b_8 = -1$ is by using a curve with $a_1 = a_3 = a_6 = 0$ and $a_4 = 1$, i.e., a curve with equation $Y^2 = X^3 + a_2X^2 + a_4X$.

5.5.3. Using the Short Weierstrass Form. It follows from the introduction that computing with only x-coordinates can be done for any curve given in Weierstrass form. Brier and Joye [**34**] and Izu and Takagi [**76**] discuss fast x-coordinate only arithmetic based on the short Weierstrass form (38). To ease comparison with the traditional representations, we will assume $a_4 = -3$, which saves some multiplications (one per group operation to be precise).

5.5.4. Exponentiation. In Table 5.5 an overview is given of the various Lucas chain methods described in Chapter 3 when applied to the Montgomery representation. Once again, α stands for a point addition and δ for a point doubling. The notation $\dot{\alpha}_3$ is used for a point addition with a fixed difference, since these are cheaper. The column Y denotes whether easy y-coordinate recovery using (31) is possible or not. The column Montgomery refers to Montgomery's original representation (or the equivalent one described in the section above) and the column Weierstrass to applying the Montgomery-technique directly to a curve in short Weierstrass form.

5.6. Conclusion

Although elliptic curves over binary fields and those over prime fields can be quite different in certain respects, there are also some striking similarities.

In both cases it turned out that if traditional coordinates are used and a double exponentiation has to be performed with one base fixed, it is advantageous to combine the double exponentiation (instead of performing two separate exponentiations). It is unclear exactly how much gain can be obtained from the fact that one base is fixed.

If Montgomery-like coordinates are used, the advantage of a fixed difference for the binary Lucas algorithm was bigger than the shorter chain length provided by the Euclidean algorithms. For curves over prime fields the difference in speed is quite small though, as already remarked by Montgomery [**130**].

In some respects different conclusions need to be drawn for the binary curves and the prime curves.

For curves defined over a binary field, comparing Table 5.1 with Table 5.2 shows that the Montgomery representation is considerably slower than traditional methods, both for single and for double exponentiation. However, for single exponentiation using the Montgomery form can still have two advantages. First of all, the uniformity of the steps in the binary algorithm provides a hedge against timing and power analysis. Using ordinary projective coordinates in conjunction with the Lucas binary algorithm is substantially slower. Secondly, the Montgomery method requires less memory during a computation.

For double exponentiation timing and power analysis are seldom of any concern, but if they were, the fast double exponentiation routines by Akishita and Montgomery would not provide a hedge. Of course one could run two single exponentiations, recover the y-coordinates and add the result. As for memory requirements, here the Montgomery representation clearly stands out. During the computation three points have to be stored, each consisting of two $\mathbb{F}_{2^l}$-elements. Moreover two elements in $\mathbb{Z}_q$ need to be stored (d and e in Algorithm 3.25). All in all $8l$ bits. On the other hand, Solinas' method precomputes four points of two $\mathbb{F}_{2^l}$-elements each (this includes the two bases). During computation, one point of three $\mathbb{F}_{2^l}$-elements is used. The exponents need to be recoded and stored as well. In total, this costs $13l$ bits. The method based on interleaving two windowed NAFs requires even more memory.

For curves over a prime field, Tables 5.4 and 5.5 show that the expected runtimes for the Montgomery-representation and the traditional technique are very close to each other. So close in fact that changing the relative costs of field squarings, multiplications and inversions might very well give different winners. For a single exponentiation routine the Montgomery form has the additional benefits of higher resistance against side channel attacks and lower memory requirements. A serious disadvantage is that not all curves can be brought into Montgomery form. If the Montgomery-technique is applied to the short Weierstrass form all curves can be used, but the resulting exponentiation routines are significantly slower.

With the current status in solving the DLP over elliptic curves and solving it over finite fields, cryptosystems based on G_{p^2-p+1} in the previous chapter and elliptic curves can be based on primes of roughly the same length. This allows a meaningful comparison (including XTR). Elliptic curves seem slightly faster, but the margin is so small that a different $S : M$ ratio and the additional cost for an elliptic curve to bring the point back to affine representation, could turn the tables.

6

Optimizing a Black Box Secret Sharing Scheme

A black box secret sharing scheme for a threshold access structure is a linear secret sharing scheme that works over any finite abelian group. In this chapter we describe how to efficiently implement Cramer and Fehr's black box secret sharing scheme. Several tweaks are used to speed up the scheme. Some of these tricks have applications for other threshold systems as well.

6.1. Introduction

In his glorious but bloody past the famous pirate Blackbeard has accumulated great wealth. These days are all gone now, Blackbeard is turning grey and he decides to hide his treasure on some uninhabited island. Only to his two most trusted lieutenants he reveals the location of the loot. But, being pirates, he does not trust either of them with the complete location. Blackbeard considers giving one lieutenant the latitude and the other the longitude of the treasure island. He realizes that the number of islands on a given parallel of latitude is fairly limited and will probably enable a single lieutenant to find the treasure without too much effort. How can Blackbeard share the secret location of the treasure among his lieutenants in such a way that any single lieutenant learns essentially nothing more, than that it is hidden on an island?

The problem above is an example of secret sharing. A dealer wants to share a secret s among n participants, such that only certain qualified coalitions of them can reconstruct the entire secret, but any non-qualified coalition should not learn anything new about the secret. The structure of which sets are qualified and which are not is commonly referred to as the access structure.

A special but simple access structure is used for threshold secret sharing. Given a threshold t, the secret is distributed among the n participants in such a way that any $t+1$ participants can reconstruct the secret, but any t participants remain clueless. (Here we adhere to the notation used by Cramer and Fehr [**52**], based on multi-party computation.) Since at least one participant is needed for reconstruction and at most n, it holds that $0 \leq t < n$. The case $t = 0$ allows any single participant to reconstruct the secret. It can be dealt with by giving each player a copy of the secret. If $t = n-1$, all participants are required for the reconstruction. The sharing can be done additively by giving $n-1$ participants random shares and the last participant the sum of all shares and the secret (this approach does require that the secret can be embedded in some finite abelian group, but this will not pose any

problems). We will henceforth assume that $0 < t < n - 1$, thereby excluding the trivial cases just mentioned.

6.1.1. Shamir's Secret Sharing. Shamir's secret sharing [**166**] is a well-known method for threshold secret sharing over a finite field $\mathbb{F}$. Given parameters n and t defining the access structure and a secret $s \in \mathbb{F}$, the dealer picks uniformly at random a polynomial $g \in \mathbb{F}[x]$ subject to $g(0) = s$. Given n different evaluation points $\alpha_i \in \mathbb{F}^*$ known to everyone, the dealer hands out share $s_i = g(\alpha_i)$ to participant i for $i = 1, \ldots, n$. The vector of shares $(s_1, \ldots, s_n)^T$ is also called the share vector and denoted $\mathbf{s}$. Together $t+1$ players know $t+1$ points on a polynomial of degree t. This allows them to determine the polynomial using Lagrange interpolation,

$$g(x) = \sum_{i \in A} s_i \prod_{j \in A, j \neq i} \frac{x - \alpha_j}{\alpha_i - \alpha_j} \ . \tag{41}$$

The secret is safe for any coalition of at most t players since for every possible value of $g(0)$ there is exactly one polynomial consistent with the shares in the hands of a coalition of t players. (Without loss of generality, a qualified subset will always consist of exactly $t + 1$ players and a non-qualified subset of exactly t players.)

Shamir's secret sharing is a special case of linear secret sharing. In a linear secret sharing scheme the secret and the shares are elements in a group G (denoted additively). If s and s' are secrets and $\mathbf{s}$ and $\mathbf{s}'$ the corresponding vectors of shares, then the vector $\mathbf{s} + \mathbf{s}'$ should be a share vector for the secret $s + s'$. A linear secret sharing scheme can be interpreted in the following way. Given the group and the access structure a $d \times e$ matrix M is defined. A secret s is shared by picking a random vector $\mathbf{g}$ in G^e with $g_0 = s$ and computing the share vector $\mathbf{s} = M\mathbf{g}$. The d shares are then distributed among the participants in a predetermined way (so $d \geq n$). Reconstruction is also linear: the participants in a qualified subset A compute the inner product of their shares and a reconstruction vector $\boldsymbol{\lambda}(A)$ to form the secret s.

6.1.2. Black Box Secret Sharing. In Shamir's secret sharing, the matrix M is (a part of) a Vandermonde matrix with entries in the field. In general, the elements in M should come from a ring R such that the group G is an R-module. All finite abelian groups are $\mathbb{Z}$-modules, so a generic choice for R is the set of integers $\mathbb{Z}$. The only problem that occurs is that for the reconstruction based on Lagrange interpolation the elements $\alpha_i - \alpha_j$ need to be inverted in the ring R. If the group order of G is known, the ring R can be chosen 'modulo' this group order.

Desmedt and Frankel [**57**] consider black box secret sharing. That is, given an arbitrary finite abelian group G, they describe how to share a secret $s \in \mathsf{G}$ without relying on the structure of G (other than that it is finite abelian). This requires that the matrix M and all the reconstruction vectors $\boldsymbol{\lambda}(A)$ are independent of the group.

Definition 6.1 (Black Box Secret Sharing). Let n and $t < n$ be nonnegative integers and let $M \in \mathbb{Z}^{d \times e}$. Then M is a black-box secret sharing scheme for n and t with expansion factor d/n if the following holds. Let G be an arbitrary finite abelian group. For an arbitrarily distributed $s \in \mathsf{G}$ let $\mathbf{g} = (g_1, \ldots, g_e)^T \in \mathsf{G}^e$ be drawn uniformly at random subject to $g_1 = s$ only. Define the share vector as $\mathbf{s} = M\mathbf{g}$

(each entry in $\mathbf{s}$ is handed out to exactly one participant). Let $A \subseteq \{1, \ldots, n\}$ be some subset of participants, let $M_A \in \mathbb{Z}^{d_A \times e}$ denote the restriction of M to the rows jointly owned by members of A, and let $\mathbf{s}_A = M_A\mathbf{g}$. Let $\boldsymbol{\lambda}(A) \in \mathbb{Z}^{d_A}$ be defined (independent of G) for all subsets $A \subseteq \{1, \ldots, n\}$ of cardinality at least $t+1$.

i. (Completeness) If A contains at least $t+1$ participants, then the inner product $(\mathbf{s}_A, \boldsymbol{\lambda}(A)) = s$ with probability 1.

ii. (Privacy) If there are at most t participants in A, then $\mathbf{s}_A$ contains no Shannon information on s.

The expansion factor is a measure for the number of group elements each participant gets. For the trivial cases $t = 0$ and $t = n-1$ the expansion factor is 1; for Shamir's secret sharing over a finite field the expansion factor is also 1. Note that the expansion factor is unrelated to the 'size' of the secret. The fact that the secret should be (encoded as) a group element limits this size. For larger secrets the secret will have to be encoded using several group elements, that can be shared independent of each other.

Black box secret sharing schemes were originally introduced with threshold RSA applications in mind. Several players share the private key d of an RSA system. The public key consists of the product of two large primes n, known as the RSA-modulus, and an integer e satisfying $de \equiv 1 \bmod \phi(n)$. The factorination of n is unknown (the values d, e, and n have nothing to do with those used in Definition 6.1). If the players want to sign a message m they need to 'reconstruct' $m^d \bmod n$ without knowing $\phi(n)$, the order of the group $\mathbb{Z}_n^*$. Although this specific problem, known as threshold RSA signatures, was later solved without using black box secret sharing schemes by Shoup [**173**], black box secret sharing remains the generic solution. It also has applications for secure general multi-party computation based on black box rings [**54**].

Desmedt and Frankel provide a solution for black box secret sharing with expansion factor $O(n)$. They achieve this by defining a ring $R = \mathbb{Z}[X]/(f)$ and regard $\mathsf{G}^{\deg f}$ as an R-module. The polynomial f is chosen in such a way, that there exist n evaluation points in the ring such that the points themselves are all units, but also all their differences are units in the ring. If Shamir's secret sharing is performed based on this module and these evaluation points, all the denominators occurring in the Lagrange coefficients are units in R (and hence the division can take place in the ring). The maximal cardinality of a subset of R such that all differences are units is called the Lenstra constant of the ring R. Finding a polynomial f of low degree but for which the ring $\mathbb{Z}[X]/(f)$ has a high Lenstra constant is an open problem in number theory. However, a possible choice for f is the p-th cyclotomic polynomial where p is the smallest prime larger than n. Since G itself is not an R-module, G^{p-1} is used instead. This gives rise to the expansion factor $p-1$, which is $O(n)$.

6.1.3. Optimal Black Box Secret Sharing. Cramer and Fehr [**52**] prove that the expansion factor for a black box secret sharing scheme is at least $\lfloor \lg n \rfloor + 1$ for all t and n with $0 < t < n-1$. Central in their work is the relationship between black box secret sharing schemes and span programs, leading to the following theorem.

THEOREM 6.2 (Cramer and Fehr). *Let n and $t < n$ be nonnegative integers and let $M \in \mathbb{Z}^{d\times e}$. Define $\varepsilon = (1, 0, \ldots, 0) \in \mathbb{Z}^e$. Then M is a black-box secret sharing scheme for n and t if and only if for $A \subseteq \{1, \ldots, n\}$ the following holds.*
(Completeness) If $|A| > t$ then $\varepsilon \in \text{im}\,(M_A^T)$.
(Privacy) If $|A| \leq t$ then there exists a $\boldsymbol{\kappa} = (\kappa_1, \ldots, \kappa_e)^t \in \ker(M_A)$ with $\kappa_1 = 1$.

Cramer and Fehr also describe a clever scheme that achieves expansion factor $\lfloor \lg n \rfloor + 2$. We will refer to this scheme as the CF scheme. In a nutshell, it shares the secret $t$wice using weak secret sharing schemes. A weak secret sharing scheme allows the reconstruction of a multiple of the secret, which is not quite the same as reconstructing s. Suppose $\delta_\alpha s$ and $\delta_\beta s$ are reconstructed from the two weak secret sharing schemes. By ensuring that δ_α and δ_β are coprime, standard Euclidean techniques can be used to recover the real secret s: let a and b be such that $a\delta_\alpha + b\delta_\beta = 1$, then $a(\delta_\alpha s) + b(\delta_\alpha s) = s$.

The weak secret sharing schemes used by Cramer and Fehr are versions of Shamir's secret sharing scheme over extension rings of $\mathbb{Z}$. This is similar to Desmedt and Frankel's scheme, but the important difference is that using two independent weak schemes, circumvents the dependency on the Lenstra constant and greatly reduces the extension degree of the rings that are used. More precisely, Cramer and Fehr show that the integers can be used for one of the rings and that the other ring can have a defining polynomial of degree $\lfloor \lg n \rfloor + 1$. The overall expansion factor is $\lfloor \lg n \rfloor + 2$, as desired. Henceforth we will use the subscript α to indicate the integer sharing and the subscript β to indicate the sharing over the extension ring. Cramer and Fehr thus proved the following theorem for the expansion factor for a black box secret sharing scheme.

THEOREM 6.3 (Cramer and Fehr). *Let n and $0 < t < n - 1$ be given, then the minimal expansion factor m' of a black box secret sharing for n participants with threshold t satisfies*

$$\lfloor \lg n \rfloor + 1 \leq m' \leq \lfloor \lg n \rfloor + 2 \ .$$

Secret sharing schemes can have additional desirable properties such as share completeness and uniformity.

A threshold scheme satisfies the share-completeness property if any t participants can reconstruct the shares of the remaining $n - t$ players when also given the secret s, in addition to their own shares. This property can be useful in protocol design, if it is necessary to simulate the view of an adversary in a security proof. A closely related example exploiting this property is Shoup's RSA threshold signature scheme [**173**]. Cramer and Fehr [**53**] give a slightly relaxed but more formal definition of share-completeness. The CF scheme satisfies the relaxed version of the share-completeness property.

Uniformity of a secret sharing scheme refers to what extent the shares behave as uniformly random chosen elements. Clearly, if all elements were to be chosen uniformly at random, no secret can ever be recovered. For a threshold scheme, the best that can happen is the uniformity of any subset of t shares, that is, the distribution of any subset of t shares is identical to the uniform distribution over

G^{d_A} (here d_A is the number of group elements that make up the t shares). In the CF scheme, not even single shares are guaranteed to be uniformly distributed.

6.1.4. Topics of This Chapter. In this chapter we describe an efficient implementation of the CF scheme. Although Cramer and Fehr show that the number of operations is bounded by a polynomial in the number of participants, they do not optimize this number of operations. Implementing the scheme will require both calls to the black box to perform group operations (mainly to add and subtract, but also to compare and sample group elements) and operations outside the realm of the black box group. We ignore the subtlety of partially black box groups. For instance if the dealer of the secret knows the order of the group, but the participants do not. The group will always be regarded as a black box.

Our main goal is to minimize the number of calls to the black box. (Our secondary goal would be to keep the work outside the black box to a minimum.) The number of calls to the black box will be measured in ops (we use "ops" as a synonym for "calls to the black box"). We will allow any polynomial time (pre)computation that does not use black box calls. This model is strongly related to that underlying the definition of addition chains. Indeed, once the matrix M is determined, minimizing the number of ops boils down to finding a shortest addition-subtraction chain for M^T, since computing $M\mathbf{g}$ corresponds to computing $\mathbf{g}^{M^T}$ in the notation of Chapter 2. The distinction between different group operations made in that chapter is not useful here, since a black box does not discriminate between them.

In Section 6.2 the weak black box version of Shamir's secret sharing scheme on which the CF scheme is based, is described. However, instead of putting the secret in the constant term of the polynomial we propose putting the secret in the leading coefficient of the polynomial. We call this the swapping trick. The main benefit of the swapping trick is that privacy comes for free, strongly reducing the size of the multiplicands δ_α and δ_β. Other benefits of the swapping trick are a smaller expansion factor if n is a power of two, less stringent conditions on the polynomial f that defines the extension for the second sharing and slightly more uniformity among the shares.

Section 6.3 concentrates on optimizing the sharing over the integers, whereas Section 6.4 does the same for the sharing over the extension ring $\mathbb{Z}[X]/(f)$. In both cases the most important aspect is trying to minimize δ_α respectively δ_β. In Section 6.4 we also discuss the implications the choice of f has for the efficiency of the scheme. Efficient ways to combine $\delta_\alpha s$ and $\delta_\beta s$ are discussed in Section 6.5.

In Section 6.6 we cheat a little bit. Given the theoretical constraints on f given by Lemma 6.5 and the desirable properties derived in Section 6.4, we use an exhaustive search to find polynomials f for $n \leq 41$ (and the runtime for this search is superpolynomial in n). Although there is no guarantee that the polynomials suggested are in any sense optimal, we believe they offer good performance.

Although we do compare the new methods with those originally suggested by Cramer and Fehr, this comparison is not entirely fair, for the simple reason that Cramer and Fehr did not attempt to optimize the runtime of their scheme. To illustrate the techniques in this chapter, the example $n = 5$ and $t = 2$ will be worked

out in detail throughout the presentation. This example shows that, even though the speedup itself is asymptotic in nature, the benefits already become evident for small numbers of participants. Moreover, experiments show that for $n > 64$ the computations outside the black box already begin to get noticeable to the point of being annoying. Of use for our comparison was an implementation of the original CF scheme in Mathematica by Krook [**96**].

Also of relevance is Chapter 7 of King's thesis [**87**], who describes an efficient implementation of the black box secret sharing scheme by Desmedt and Frankel. King concentrates on the application of the black box scheme for a threshold RSA signature scheme and gives a comparison with Shoup's threshold RSA version. This comparison requires that the operations outside the black box are also taken into account, since they can easily take up most of the time. This makes a comparison with the present work harder.

6.2. Weak Black Box Secret Sharing

6.2.1. Reformulating Shamir Secret Sharing. With some adaptations Shamir's secret sharing scheme can be used for weak black box secret sharing. As already described, if $s \in \mathsf{G}$ is the secret to be distributed, then the dealer will pick a polynomial in $\mathsf{G}[x]$ of degree $t+1$ at random, but with constant term s. Since G is a $\mathbb{Z}$-module, the polynomial can be regarded as a function $g : \mathbb{Z} \to \mathsf{G}$. Given n publicly known distinct elements α_i in $\mathbb{Z}$ —a popular choice is $\alpha_i = i$— the dealer hands out share $g(\alpha_i)$ to participant i for $i = 1, \ldots, n$.

The scheme described above is not a black box secret sharing scheme though. It satisfies neither the completeness nor the privacy condition from Definition 6.1. This can however be fixed by suitable multiplication of the secret by a constant, giving rise to a weak black box secret sharing scheme. Before going into details, it is useful to reformulate Shamir's secret sharing scheme in terms of Definition 6.1 in order to apply Theorem 6.2.

Given a secret s, the dealer chooses $\mathbf{g} = (g_0, \ldots, g_t)$ in G^{t+1} uniformly at random with $g_0 = s$. This vector can be identified with the polynomial $g(x) = \sum_{i=0}^{t} g_i x^i$. The share vector $\mathbf{s} = (s_1, \ldots, s_n)^T$ is computed as $M\mathbf{g}$, where $M \in \mathbb{Z}^{n \times (t+1)}$ is the Vandermonde matrix

$$M = \begin{pmatrix} 1 & \alpha_1 & {\alpha_1}^2 & \ldots & {\alpha_1}^t \\ 1 & \alpha_2 & {\alpha_2}^2 & \ldots & {\alpha_2}^t \\ 1 & \alpha_3 & {\alpha_3}^2 & \ldots & {\alpha_3}^t \\ \vdots & & & & \\ 1 & \alpha_n & {\alpha_n}^2 & \ldots & {\alpha_n}^t \end{pmatrix} .$$

We will now describe how the scheme can be adapted to satisfy the completeness and privacy properties.

6.2.2. Completeness. The completeness property relates to the ability of qualified subsets to reconstruct the secret. Reconstruction of the polynomial (and hence of the secret) based on Lagrange interpolation runs into trouble because of the divisions

by $\alpha_i - \alpha_j$. This problem can be taken care of by multiplying both sides of (41) with some $\delta_\alpha \in \mathbb{Z}$ such that all the divisions can take place in $\mathbb{Z}$. As a consequence, $\delta_\alpha s$ is recovered instead of the real secret s. A possible, generic, choice for δ_α is $\prod_{i,j\in A, i\neq j}(\alpha_i - \alpha_j)$. Better ways of determining δ_α are described in Sections 6.3.2 and 6.4.3.

6.2.3. Privacy. According to Theorem 6.2 a black box secret sharing scheme satisfies the privacy property iff it is not possible to find a vector $\boldsymbol{\kappa}$ with $\kappa_1 = 1$ subject to $M_A\boldsymbol{\kappa} = 0$ for $|A| = t$. We therefore determine the kernel of M_A.

LEMMA 6.4. *Let $\alpha_1, \ldots, \alpha_t$ be distinct elements in $\mathbb{Z}$. Define $a_0, \ldots, a_t$ by $\prod_{i=1}^{t}(x - \alpha_i) = \sum_{i=0}^{t} a_i x^i$ (as polynomials) and $\mathbf{a} = (a_0, \ldots, a_t)^T$. Define*

$$M_A = \begin{pmatrix} 1 & \alpha_1 & {\alpha_1}^2 & \ldots & {\alpha_1}^t \\ 1 & \alpha_2 & {\alpha_2}^2 & \ldots & {\alpha_2}^t \\ \vdots & & & & \\ 1 & \alpha_t & {\alpha_t}^2 & \ldots & {\alpha_t}^t \end{pmatrix}$$

then $\ker M_A = \langle \mathbf{a} \rangle$.

Proof: Any element $\mathbf{r} = (r_0, \ldots, r_t)^T \in \mathbb{Z}^{t+1}$ can be regarded as a polynomial $r(x) = \sum_{i=0}^{t} r_i x^i$. An element $\mathbf{r} \in \mathbb{Z}^{t+1}$ is in the kernel of M_A iff $r(\alpha_i) = 0$ for $i = 1, \ldots, t$. By construction, this is the case for the polynomial a (the fact that $\mathbf{a} \in \ker(M_A)$ is already sufficient for the secret sharing).

Suppose $\mathbf{a}' \in \ker M_A$. Regarded as polynomials a' has degree at most t and a has degree exactly t. Exploiting the fact that a is monic, we see $r = a' \mod a$ has degree smaller than t. Since a nonzero polynomial of degree strictly smaller than t cannot have t distinct zeroes, but, $r(\alpha_i) = 0$ for all i, we know that $r = 0$, implying that a' is a polynomial multiple of a. Since the degree of a is t and that of a' at most t, we also have that $\mathbf{a}'$ is a scalar multiple of $\mathbf{a}$. *Q.E.D.*

Cramer and Fehr fix the property by sharing a multiple Δs of the secret where $\Delta = \prod_{0\leq j<i\leq n} \alpha_i - \alpha_j$ with $\alpha_0 = 0$. The value Δ is based on the determinant of M_A; it is a multiple of $a_0 = \prod_{0<i\leq n} \alpha_i$. Sharing (a multiple of) $a_0 s$ is secure. The proof is basically a reproduction of the (omitted) proof of Theorem 6.2. Let $\mathbf{s}$ be the share vector resulting from sharing $a_0 s$. Given another secret s', compute $\mathbf{s}' = \mathbf{s} + M(a(s' - s))$. The share vector corresponds to the secret s', which is unrelated to s, but $\mathbf{s}'$ restricted to the view of the adversary equals $\mathbf{s}_A$, since $M_A a = 0$.

With a little trick, it is actually possible to discard the multiplication of the secret completely. The observation that $a_t = 1$ suggests to use the leading coefficient of the polynomial g to hide the secret. Indeed, since $a_t = 1$, privacy is guaranteed immediately without the need to share a multiple of the secret. This trick is called the swapping trick.

6.2.4. Side Effects. Putting the secret in g_t saves a costly multiplication during the dealing phase. This can also be felt in the reconstruction later on when the two sharings are combined. An extra advantage is that the multiple of the secret

that is recovered no longer needs to be a multiple of n if n is prime, which relaxes the condition on the polynomial f used to make the extension ring $R = \mathbb{Z}[X]/(f)$. Moreover, the evaluation point 0 is available for which evaluation comes for free. The extra evaluation also gives a small additional benefit if n is a power of 2 (more will become clear in Section 6.2.5).

Reconstruction proceeds slightly differently. We are only interested in g_t, so it suffices to compute the leading coefficient of $\sum_{i\in A} s_i \prod_{j\in A, j\neq i} \frac{x-\alpha_j}{\alpha_i-\alpha_j}$ by computing $\sum_{i\in A} s_i(\delta_A \lambda_i)$ where $\lambda_i = \prod_{j\in A, j\neq i} \frac{1}{\alpha_i-\alpha_j}$ (we will use λ_i throughout, with A implicitly understood).

Cramer and Fehr also discuss share completion and uniformity.

The share completion property still holds, since given the secret and t shares, it is possible to first peel of the highest order of g using the secret and then use the shares to reconstruct the remaining polynomial of degree $t-1$. Note that the reconstruction will introduce a factor δ to the polynomial, just as in the ordinary, qualified set reconstruction described below (in specific, this means that the g_t instead of g_0 trick is of no avail in Shoup's threshold signature, since his Δ stems from the share completion property and not from the security property).

Uniformity is improved by using g_t instead of g_0. Multiplication of the secret with Δ in the g_0 case means that the share i will always be a multiple of i of some group element. Hence, unless the group order is coprime to $n!$, this cannot give a uniform distribution. (In Shoup's threshold RSA signature scheme, the group order is known to be the product of two large primes; consequently Δ is invertible modulo the group order and is not really needed at all during the sharing). If g_t is used, all the individual shares look uniformly random, but it can be verified that $g(i) - g(0)$ is a multiple of i again, so the shares are not pairwise independent.

6.2.5. Conditions on the Extension. In this section we will derive necessary and sufficient conditions on the polynomial f for the existence of $\beta_i \in R$ such that δ_β can be chosen coprime to δ_α. For the moment we will assume that $\delta_\beta \in \mathbb{Z}$.

If $\delta_\alpha = (n-1)!$ this implies that δ_β should not have any (prime) divisors smaller than n. Note that it is not possible to get rid of any of these primes, either by using a different δ_α or even different $\alpha_i \in \mathbb{Z}$, as can be shown by the pigeon hole principle. For the reconstruction, it is required that $\alpha_i - \alpha_j$ divides δ_α for all $0 < i < j \leq n$. Clearly, a prime p divides δ_α iff $\alpha_i \equiv \alpha_j \mod p$. Since there are n different values of α_i for all primes $p < n$ collisions will occur and $p|\delta_\alpha$.

LEMMA 6.5 (Cramer, Fehr, H.W. Lenstra, Jr.). *Let f be a monic irreducible polynomial in $\mathbb{Z}[x]$. Then there exist $\beta_i \in \mathbb{Z}[x]/(f), i = 1, \ldots, n$ and $\delta_\beta \in \mathbb{Z}$ subject to $\gcd(\delta_\beta, (n-1)!) = 1$ and $(\prod_{0<i<j\leq n} \beta_i - \beta_j)$ divides δ_β in the ring $\mathbb{Z}[x]/(f)$ iff for all primes $p < n$ the smallest irreducible factor of f modulo p has degree d satisfying $p^d \geq n$.*

Proof: Suppose the claimed δ_β and β_i exist. Since we can divide δ_β by $\beta_i - \beta_j$ in the ring $\mathbb{Z}[X]/(f)$ for all $i \neq j$ the factor δ_β will be in the ideal generated by $\beta_i - \beta_j$. The intersection of this ideal with the integers is a subgroup generated by d_{ij}. A

necessary and sufficient condition is that $p \not| d_{ij}$ for primes $p < n$. If we focus on any specific prime $p < n$, we can consider all elements modulo p. Suppose that f' is an irreducible factor of f modulo p, then we can write $d_{ij} \equiv k(\beta_i - \beta_j) + k'f' \bmod p$. The condition $p \not| d_{ij}$ is equivalent to invertibility of $(\beta_i - \beta_j)$ modulo f' and p. If the degree of f' is d, computing modulo both f' and p corresponds to working in $\mathbb{F}_{p^d}$. In a finite field the difference of two elements is invertible iff the two elements are unequal. There are p^d elements in $\mathbb{F}_{p^d}$, so there can be at most p^d different points β_i.

This proves necessity of the condition on the smallest irreducible factors of f modulo the primes smaller than n. Sufficiency follows by picking points β_i modulo (f', p) for all irreducible factors f' of f modulo p for all primes $p < n$. The CRT implies that β_i also exist modulo f simultaneously satisfying all the modulo (f', p) congruencies. *Q.E.D.*

Filling in $p = 2$ in $p^d \geq n$ gives the minimum value $m = \lceil \lg n \rceil$. Cramer and Fehr originally only consider f irreducible modulo all the primes $p \leq n$. The relaxation to polynomials f whose 'smallest' irreducible factor still has sufficiently large degree is credited to H.W. Lenstra, Jr [**53**].

Two minor side effects of putting the secret in the leading coefficient are also incorporated in Lemma 6.5. In the original CF scheme, the lack of interpolation in 0 resulted in the condition that, for all primes $p \leq n$ the least degree d should satisfy $p^d > n$. The difference is that Cramer and Fehr need $m = \lfloor \lg n \rfloor + 1$ which differs if n is a power of two and if n is prime, an extra constraint is laid to f.

If f happens to be irreducible modulo all primes $p < n$, it is possible to set β_i to the unique $(0, 1)$-polynomial that evaluates to $i - 1$ in the point 2. In this case $\beta_i - \beta_j$ will be a $(-1, 0, 1)$-polynomial of degree lower than m.

If f factors modulo some prime $p < n$, this choice is no longer guaranteed to work. One of the factors of f might also be a factor modulo p of some $\beta_i - \beta_j$. One way to find evaluation points in this setting is based on the Chinese Remainder Theorem and the observation that for each $0 \leq i < n$, for each prime p and for each irreducible factor f' of f (modulo p), there is a unique polynomial modulo p that evaluates to i in p. In Section 6.4.3 some of the computational ramifications of this approach are discussed (and they seem pretty severe).

Example 6.6. For $n = 5$ the required extension degree is $\lceil \lg 5 \rceil = 3$. An example of an irreducible polynomial of degree 3 in $\mathbb{F}_2[X]$ is $X^3 + X^2 + 1$. The polynomial $X^3 + 2X^2 + 1$ is irreducible modulo 3. Applying the Chinese Remainder Theorem yields $f(X) = 1 + 5X^2 + X^3$. This polynomial also illustrates one of the additional benefits of putting the secret in the most significant coefficient during the weak Shamir secret sharing. If the constant term would have been used, the integer sharing would return a multiple of the secret divisible by 5, requiring f to be irreducible modulo 5. Since $f(-1) \equiv 0 \bmod 5$ this is clearly not the case.

Another polynomial we will use in future examples is $X^3 - X - 1$. This polyniomal is irreducible both in $\mathbb{F}_2[X]$ and $\mathbb{F}_3[X]$, but has some computational advantages compared to $X^3 + 5X^2 + 1$.

6.3. Weak Secret Sharing over the Integers

6.3.1. Optimizing the Sharing. The dealer has to perform a multi-point polynomial evaluation. For single point polynomial evaluation over rings Horner's rule is well known. It rewrites the evaluation as

$$g(i) = g_0 + i(g_1 + i(g_2 + \dots i(g_{t-1} + ig_t)\dots) \ ,$$

taking t additions and t multiplications by i. When Horner's rule is applied to evaluate $g(i)$ in the current black box setting, the number of required ops is $t(l(i)+1)$, where l is the function given in Definition 2.2. If Horner's rule is applied on all n interpolation points separately, the costs are $O(tn \log n)$ since $\sum_{i=1}^{n} l(i) = O(n \log n)$ and the binary method (Algorithm 2.7) achieves this asymptotic bound, so there is no need to use optimal addition chains.

In the present case, minimizing the number of black box calls for multi-point polynomial evaluation requires finding a minimal addition-subtraction chain for the matrix

$$M_\alpha = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & -1 & 1 & \dots & (-1)^t \\ 1 & 2 & 4 & \dots & 2^t \\ 1 & -2 & 4 & \dots & (-2)^t \\ \vdots & & & & \\ 1 & n/2 & (n/2)^2 & \dots & (n/2)^t \end{pmatrix}$$

(here n is even, the case n is odd is similar).

Pippenger's bound (page 24) gives the upper bound $t^2 \lg n + nt^2 \lg n/(\lg n + 2 \lg t + \lg \lg n)$ on the number of ops required. The lower order terms are neglected and since $t < n$, it is clear that this runs in $O(nt^2)$. For fixed t and a growing number of participants this is an improvement over Horner's rule. As special cases $t = 1$ and $t = 2$ can be mentioned.

For $t = 1$ at most $\approx 2n$ ops are necessary, since the $(t + 1)$-fold simultaneous double exponentiation can be rewritten as a twofold simultaneous $(t+1)$-tuple multi-exponentiation. These two multi-exponentiations can be performed separately and, using duality once more, correspond to $(t + 1)$-fold simultaneous exponentiations. One of these is for free (containing only ones), the other costs n ops counting inversions. The use of duality costs about n ops, so the total cost is indeed $\approx 2n$ ops.

For $t = 2$ at most $\approx 4n$ ops are needed. In this case three multi-exponentiations have to be performed. Two of these are identical to the case $t = 1$, the third corresponds (after duality) to computing all the squares up to $n/2$. This can be done surprisingly fast, for instance by exploiting the fact that a large number of squares can be written as the sum of two previous squares (e.g., $3^2 + 4^2 = 5^2$). The total cost for computing all these squares is at most n ops (using a simplified version of [**90**, 4.6.3, Exercise 38]). The total cost is then $\approx 4n$ ops, taking into account the extra costs due to duality.

Traditional multi-point polynomial evaluation over rings is considerably cheaper than this. However, in rings multiplications by constants are allowed (for free) and the roots of the polynomial that is being evaluated can be used. In the present case the coefficients of the polynomial are elements of a black box group, making it improper to base the evaluation algorithm on specifics of these elements. The concept of root is for instance not clearly defined over such a group. To summarize, we want our method to be largely independent of the polynomial in $\mathsf{G}[x]$ that is being evaluated.

We settle for the statement that the sharing of the M_α matrix asymptotically can be done in $O(tn \log n)$ ops. If the secret would be hidden in the constant term of the polynomial the dealer has to multiply the secret in order to achieve privacy. The bitlength of the multiplicand is representative for the costs involved. If the original proposal by Cramer and Fehr based on the determinant of M_α is used, this seemingly innocent multiplication requires $O(n^2 \log n)$ ops, which might very well be more expensive than the rest of the integer sharing. If the smaller value $n!$ is used to fix the privacy property, the costs of the extra multiplication are $O(n \log n)$ ops, which is bearable.

6.3.2. Optimizing the Reconstruction. During the reconstruction the participants need to compute $\sum_{i \in A} (\delta_\alpha \lambda_i) s_i$, where s_i is an element of the group G and $\delta_\alpha \lambda_i$ is an integer. The size of δ_α directly reflects on the time it takes to do the reconstruction. The value given by Cramer and Fehr is far from minimal. For simplicity we will assume that the evaluation points are $0, \ldots, n-1$ but it is clear that the same arguments hold if $\alpha_i = i - \lfloor \frac{n}{2} \rfloor$ for instance.

If the set of participants is known, it is quite easy to determine the minimal δ_A necessary. Each participant $i \in A$ needs to be able to divide by $d_{A,i} = \prod_{j \in A, j \neq i} (i - j)$, so δ_α needs to be a multiple of this value. The minimal δ_α for a specific qualified subset A is the least common multiple of all these $d_{A,i}$ where $i \in A$. If a more uniform δ_α is desired that works for all qualified subsets of a given size, one could attempt to compute the least common multiple of all $d_{A,i}$, ranging over all subsets A of cardinality $t+1$ and $i \in A$. Unfortunately, the number of such sets A is not polynomial in n if for instance t is linear in n.

There is an easy alternative though. If we focus on a single participant, then we see that $d_{A,i}$ is a divisor of $(\prod_{0 \leq j < i} i - j)(\prod_{i < j \leq n-1} j - i) = i!(n-1-i)!$, which itself is a divisor of $(n-1)!$. Hence $\delta_\alpha = (n-1)!$ suffices, regardless of the threshold. This is a known argument, also used by Shoup [**173**].

A slightly more advanced method gives the minimal value of δ_α given the threshold. The trick is to look at the prime factorization of δ_α. We know that δ_α can be written as $\prod_{p < n, p \text{ prime}} p^{e_p}$ for suitable e_p. After determining for each prime $p < n$ the minimal order e_p in δ_α the minimal value of δ_α follows naturally. To determine e_p for a given prime, we will concentrate on the participant with the share 0 and assume he is part of a qualified subset A of size $t+1$, so $d_{A,0} = \prod_{0 \neq j \in A} j$. Determining the minimal order of δ_α for a given prime boils down to determining the

maximum order of $d_{A,i}$ for this prime where all sets A of cardinality $t+1$ that contain i have to be considered. This can be achieved by simply taking the t elements in the set $\{1,\ldots,n-1\}$ with maximal order. These t elements can be determined by actually computing the order for all elements in $\{1,\ldots,n-1\}$ and then sorting them, but alternatively δ_α can be computed more directly using Algorithm 6.7. It can be proven that the focus on the participant with share 0 does not result in a loss of generality.

Two special cases are worth mentioning, since they are quite common in cryptographic protocol design. These are the cases $t = \lfloor \frac{n}{2} \rfloor$ and $t = \lfloor \frac{n}{3} \rfloor$. If a simple majority is required, or $t = \lfloor \frac{n}{2} \rfloor$, there are for any given prime p at most $\lfloor \frac{n-1}{p} \rfloor \leq t$ values that contribute, so $\delta_\alpha = (n-1)!$ is not only sufficient but also necessary (unless δ_α is based on a single subset). If $t = \lfloor \frac{n}{3} \rfloor$ for all primes $p > 2$ once more $\lfloor \frac{n-1}{p} \rfloor \leq t$, so only for $p = 2$ some gain can be expected.

ALGORITHM 6.7 (Determining the Minimal δ_α).
On input two integers n and t, $0 < t < n-1$, this algorithm determines the minimal δ_α.

1. [Initialize] Set $p \leftarrow 2$ and $d \leftarrow 1$.
2. [Finished?] If $p \geq n$ terminate with output d.
3. [Initialize subroutine] Set $e \leftarrow 0, t' \leftarrow 0$ and $s \leftarrow \lfloor \log_p(n-1) \rfloor$.
4. [Finished subroutine?] If $t' \geq t$ set $d \leftarrow dp^e$, set p to be the next prime and go back to step 2.
5. [Decrease s] Set $e \leftarrow e + s\min(t-t', \lfloor \frac{n-1}{p^s} \rfloor - t')$ and $t' \leftarrow t' + \lfloor \frac{n-1}{p^s} \rfloor$. Decrease s by one and go back to the previous step.

EXAMPLE 6.8. For $n = 5$ and $t = 2$ we have to consider $p = 2$ and $p = 3$. If $p = 2$ then $\lfloor \log_2 4 \rfloor = 2$, so initially $s = 2$. There is only one positive multiple of 4 smaller or equal to 4, contributing 2 to e. There are two even numbers smaller or equal to 4, but one of these is 4 which we already counted. This gives a total of $e = 3$ for $p = 2$. If $p = 3$ then $\lfloor \log_3 4 \rfloor = 1$, so initially $s = 1$. Since there is only one positive multiple of 3 smaller or equal to 4, we know that $e = 1$ for $p = 3$. Combining the result gives $\delta_\alpha = 2^3 3^1 = 24$. This result for δ_α happens to be equal to $(n-1)!$, which is not very surprising since $t = \lfloor \frac{n}{2} \rfloor$.

To put the value 24 in a little perspective, if the participants joining in the reconstruction own the shares $-2, 0$, and 1, the least common multiple of $6, 2$, and 3 suffices. Cramer and Fehr's upper bound would yield $\delta_\alpha = 34560$.

For the actual reconstruction there are two main scenarios. Either the participants all hand in their shares and some central authority computes the inner product. The scenario corresponds to a single multi-exponentiation with $t+1$ bases and exponents. In the second scenario each participant computes and publishes $(\delta_\alpha \lambda_i) s_i$, corresponding to a single exponentiation. The combination of the modified shares takes t black box calls by a central authority. If δ_α is independent of A, it is likely that $\gcd_{i \in A}(\delta_\alpha \lambda_i) = \delta_\alpha / \operatorname{lcm}_{i \in A} d_{A,i}$ is unequal to 1. In this case it could be advantageous to allow the individual players to use $\delta'_\alpha = \operatorname{lcm}_{i \in A} d_{A,i}$ for

the computation of $(\delta'_\alpha \lambda_i)s_i$ and let the central authority take care of the factor $\delta_\alpha/\delta'_\alpha$ that remains (after adding the modified shares).

In both scenarios the number of ops is mainly determined by the maximum bitlength of $\delta_\alpha \lambda_i$. Since $\lambda_i \leq 1$ for all i, this is clearly upper bounded by the bitlength of δ_α. Using $\delta_\alpha = (n-1)!$ gives a bitlength of $O(n \log n)$. For comparison, the bitlength of $1/\lambda_i$ is at least $O(t \log t)$ and at most $O(t \log n)$. For the determination of the necessary addition chains, the factorization of the 'exponents' can be used to find very good chains.

EXAMPLE 6.9. Suppose that for $n = 5$ and $t = 2$ the qualified subset wanting to reconstruct the secret owns the shares $-2, 0$, and 1. From Example 6.8 we know that $\delta_\alpha = 24$ and, for this specific qualified subset, that $\delta'_\alpha = 6$. The three relevant Lagrange coefficients are $\lambda_{-2} = 1/6, \lambda_0 = -1/2$, and $\lambda_1 = 1/3$. Central reconstruction based on $\delta'_\alpha = 6$ corresponds to a multi-exponent with exponent $(1, -3, 2)^T$, which takes 4 ops. In the case of decentralized reconstruction, player '-2' does not need to do anything, player '0' needs 2 ops and player '2' 1 op. The combination takes 2 extra ops, so the total number of ops is slightly larger than in the centralized situation. Taking care of the discrepancy between δ_α and δ'_α takes 2 ops.

6.4. Weak Secret Sharing over an Extension Ring

In this section we discuss efficient implementation of the weak black box secret sharing over an extension ring $R = \mathbb{Z}[X]/(f)$. This scheme uses the fact that sufficiently many copies of G can be regarded as an R-module. Before going into the details of optimizing the dealing and the reconstruction we examine the computational aspects of this module, including which choices of f would lead to efficiency improvements.

6.4.1. The Module Product. Let $f \in \mathbb{Z}[X]$ be the monic irreducible polynomial of degree m, say $f = X^m + f_{m-1}X^{m-1} + \cdots + f_1 X + f_0$, that is used to define the integral extension ring $R = \mathbb{Z}[X]/(f)$. The group G^m, consisting of m copies of G under pointwise addition, can be regarded as an R-module. The obvious way to denote an element $\mathbf{g} \in \mathsf{G}^m$ is as a vector $\mathbf{g} = (g_0, \ldots, g_{m-1})$. The element $\mathbf{g}$ can alternatively be represented by the polynomial $g(X) = g_{m-1}X^{m-1} + \cdots + g_1 X_1 + g_0$ (with $g_i \in \mathsf{G}$ for $i = 0, \ldots, m-1$). An element $r \in R$ can be represented by the polynomial $r = r_0 + \cdots + r_{m-1}X^{m-1}$ or by the vector $\mathbf{r} = (r_0, \ldots, r_{m-1})$ where $r_i \in \mathbb{Z}$ for $i = 0, \ldots, m-1$. In some cases it makes more sense to use a polynomial of degree higher than $m-1$ to represent r. Throughout this section the link between a vector $\mathbf{g}$ and the polynomial g will be used without constant reminder.

For $\mathbf{g} \in \mathsf{G}^m$ and $r \in R$ the module product $\mathbf{y} = r \bullet \mathbf{g} \in \mathsf{G}^m$ is defined such that $y = rg \bmod f$. (Alternatively, simply say that the group $G = \mathsf{G}[X]/(f)$ is well defined under addition, abelian and is also a $\mathbb{Z}[X]$-module, so in specific an R-module. Isomorphy of $\mathsf{G}[X]/(f)$ with G^m does the rest.) For any r the module multiplication is in fact a linear transformation from G^m to G^m, so we can associate a matrix $[r] \in \mathbb{Z}^{m \times m}$ with $r \in R$ such that $r \bullet \mathbf{g} = \mathbf{g}[r]$ for all $\mathbf{g}$. The use of $\mathbf{g}[r]$

and not the perhaps more natural $[r]\mathbf{g}$ is to bring the notation in line with that of addition chains from Chapter 2.

The companion matrix of the irreducible monic polynomial f is defined to be the following $\mathbb{Z}^{m\times m}$ matrix:

$$F^T = \begin{pmatrix} 0 & 0 & \dots & 0 & -f_0 \\ 1 & 0 & \dots & 0 & -f_1 \\ 0 & 1 & \dots & 0 & -f_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & -f_{m-1} \end{pmatrix} .$$

The matrix F bears great resemblance with the companion matrix belonging to a linear recurrence (Section 1.3). This is no surprise; for all $k \in \mathbb{Z}^{\geq 0}$ it holds that $(X^k, X^{k+1}, \dots, X^{k+m-1}) = (X^0, X^1, \dots, X^{m-1})(F^T)^k$. Also, with $\mathbf{g} \in \mathsf{G}^m$, the reduction of the polynomial $x^i g(x)$ will correspond to the vector $\mathbf{g}F^i$. This allows us to be more specific about the matrix $[r] \in \mathbb{Z}^{m\times m}$ associated to $r \in R$ and satisfying $r \bullet \mathbf{g} = \mathbf{g}[r]$ for all $\mathbf{g}$. Indeed,

$$\begin{aligned} r \bullet \mathbf{g} &= r_0 g + r_1 X g + r_2 X^2 g + \dots \\ &= r_0 \mathbf{g}F^0 + r_1 \mathbf{g}F^1 + r_2 \mathbf{g}F^2 + \dots \\ &= \mathbf{g}(\sum_{i=0} r_i F^i) \\ &= \mathbf{g}[r] \ . \end{aligned}$$

From this we see that $[r] = \sum_{i=0} r_i F^i$, where the maximum degree of r is deliberately not specified.

Optimizing the number of ops for a module multiplication is equivalent to finding a shortest addition-subtraction chain for the matrix $[r]$. In practice we will have to settle for a reasonably short chain. Computing such a chain can of course be done with the general techniques for computing short addition chains as described in Chapter 2.

Given r, simply determine $[r]$ and use an efficient addition-subtraction chain algorithm to determine $\mathbf{g}[r]$. Pippenger's bound (Theorem 2.5) gives an accurate indication of the asymptotics of the length of such a chain when the matrix $[r]$ grows bigger (the entries or the dimension). In order to apply the bound, an upper bound on the entries of $[r]$ is needed. Asymptotically, a good estimate for an upper bound seems to be $\lambda_f^{d_r}||r||$, where d_r denotes the degree of r, $||r||$ denotes the maximum absolute values of the coefficients of r, and λ_f is the largest absolute value of the set of eigenvalues. The latter is based on the fact that the characteristic polynomial of F is f. The eigenvalues of F are therefore the roots of f (over $\mathbb{C}$). Since f is irreducible over $\mathbb{Z}$, all eigenvalues are different, so F is diagonizable and the size of the entries in F^i is asymptotically dominated by the i-th power of the eigenvalue which is largest in absolute value. (Using the well known fact that if $F = Q_f \Lambda_f {Q_f}^{-1}$, then $F^i = Q_f \Lambda_f^i {Q_f}^{-1}$.)

Using the bound on the entries of $[r]$ and the knowledge that $[r]$ is an m by m matrix, Pippenger's bound implies that a module multiplication can be performed at a cost of about

$$(m + \frac{m^2}{2 \lg m \lg(\lambda_f^{d_r} ||r||)}) \lg(\lambda_f^{d_r} ||r||) \ .$$

In the special case that only the first coordinate of $[r]\mathbf{g}$ is needed, this reduces to

$$(1 + \frac{m}{\lg m \lg(\lambda_f^{d_r} ||r||)}) \lg(\lambda_f^{d_r} ||r||) \ .$$

EXAMPLE 6.10. Suppose the module is based on $f(X) = X^3 + 5X^2 + 1$ and $(13X^2 + 20X + 17) \bullet \mathbf{g}$ needs to be computed for some $\mathbf{g}$. We first determine the companion matrix of f, namely

$$F^T = \begin{pmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & -5 \end{pmatrix} \ .$$

For the given $r(X) = 13X^2 + 20X + 17$ the matrix of interest is

$$\begin{aligned} [r] &= 17 \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + 20 \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & -5 \end{pmatrix} + 13 \begin{pmatrix} 0 & 0 & 1 \\ -1 & 0 & -5 \\ 5 & -1 & 25 \end{pmatrix} \\ &= \begin{pmatrix} 17 & 20 & 13 \\ -13 & 17 & -45 \\ 45 & -13 & 242 \end{pmatrix} \ . \end{aligned}$$

There is an addition-subtraction chain of length 46 to compute $[r]$.

The polynomial $f(X) = X^3 + 5X^2 + 1$ has a root of absolute value ≈ 5. The polynomial $\tilde{f}(X) = X^3 - X - 1$ is irreducible modulo 2 and 3 but the absolute value of its root does not exceed 1.5, so one expects less costly module multiplications using this polynomial. Indeed, using this polynomial to define the ring, multiplication by $r(X) = 17 + 20X + 13X^2$ boils down to computing $\mathbf{g}[r]$ where

$$[r] = \begin{pmatrix} 17 & 20 & 13 \\ 13 & 17 & 33 \\ 33 & 13 & 50 \end{pmatrix} \ ,$$

which can be computed in 34 ops. If only the first element of $r \bullet \mathbf{g}$ is required, it suffices to compute $\mathbf{g}(17, 13, 33)^T$, which can be done in 11 ops.

Polynomial Multiplication. Pippenger's algorithm underlying his upper bound does not directly exploit any special structure $[r]$ might have. However, in this case $[r]$ certainly has some structure. One could for instance multiply the polynomials $g(x)$ and $r(x)$ first and reduce the result modulo f afterwards. This method is based on the decomposition $[r] = [\tilde{r}][f]$, where the matrix $[\tilde{r}]$ represents the polynomial multiplication and $[f]$ the subsequent reduction.

In the literature, polynomial multiplication is usually counted in the number of ring multiplications. The number of multiplications needed to multiply two polynomials of degree $m-1$ is denoted by $\mu(m)$. It is well known that $m \leq \mu(m) \leq m^2$. Using Karatsuba's technique leads to $\mu(m) = O(m^{\lg 3})$, but at the cost of enlarging the multiplicands in $\mathbb{Z}$ by a factor of at most m. Here the multiplications are those defined by regarding G as $\mathbb{Z}$-module which is a slightly different setting. The number of group operations of G is then minimized by picking an appropriate addition-subtraction chain corresponding to the freshly formed multiplicands. This gives us either $O(m^2 \log ||r||)$ or the faster $O(m^{\lg 3}(\log ||r|| + \log m))$ black box group operations based on Pippengers upper bound. The number of operations in $\mathbb{Z}$ is negligible (some additions). Note also that for certain regular polynomials improvements are possible.

An alternative way is by directly determining an addition(-subtraction) chain for the matrix $[\tilde{r}]$. This matrix is the following m by $(2m-1)$ matrix:

$$[\tilde{r}] = \begin{pmatrix} r_0 & r_1 & \cdots & r_{m-1} & 0 & \cdots & 0 \\ 0 & r_0 & \cdots & r_{m-2} & r_{m-1} & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & r_0 & r_1 & \cdots & r_{m-1} \end{pmatrix}. \tag{42}$$

After determining the polynomial rg, the result still needs to be reduced modulo f. This can be done using the matrix style introduced above, exploiting that $(X^{m-2}, X^{m-1}, \ldots, X^{2m-2}) = (X^0, X^1, \ldots, X^{m-1})F^{m-1}$. More precisely, the matrix $[f]$ is the $(2m-1)$ by m matrix that consists of an $m \times m$ identity matrix in the top rows and the transpose of the matrix F^{m-1} in the bottom rows (there is one row overlap). It helps if the largest entry of F^{m-1} is still small. This is related to the degree of $f(X) - X^m$. If this degree is large, the reduction will cascade, giving rise to a big F^{m-1}. If the degree is at most $\lfloor \frac{m-1}{2} \rfloor$, the largest entry of F^{m-1} is the same as that of F. For $m > 2$ it is always possible to find a polynomial f with coefficients at most $\prod_{p<n, p \text{ prime}} p$ and with the degree of $f(X) - X^m$ at most $\lfloor \frac{m-1}{2} \rfloor$. As a result, the reduction will cost $O(m^2 n)$ ops.

Example 6.11. We will consider the same module multiplication as in Example 6.10 with the only difference that we swap the roles of f and $\tilde{f}$, so we have $f(X) = X^3 - X - 1$ and $\tilde{f}(X) = X^3 + 5X^2 + 1$. We begin with the polynomial multiplication based on $r(X) = 13X^2 + 20X + 17$. The corresponding matrix is

$$[\tilde{r}] = \begin{pmatrix} 17 & 20 & 30 & 0 & 0 \\ 0 & 17 & 20 & 30 & 0 \\ 0 & 0 & 17 & 20 & 30 \end{pmatrix}.$$

There is an addition chain of length 27 for this matrix.

The reduction based on f requires finding an addition chain for the matrix

$$[f] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} .$$

For this matrix it is possible to find an optimal chain of length 4 by hand.

If the other polynomial, $\tilde{f}$ is used for the definition of the extension ring, the relevant reduction is based on the matrix

$$[\tilde{f}] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & -5 \\ 5 & -1 & 25 \end{pmatrix} .$$

This time a little more effort is needed for finding an addition-subtraction chain by hand, but a chain of length 10 can be found.

In both cases we see that this indirect method gives better chains than the direct method.

Summary. We can distinguish between matrix multiplication on one side and polynomial multiplication on the other to implement the module product $r \bullet \mathbf{g}$. An important difference between the direct matrix version and the polynomial version is that in the latter case the reduction matrix $[f]$ is fixed, so it is more worthwhile to spend time optimizing it (first by choosing some 'nice' f, later by finding good addition-subtraction chains for $[f]$). Also, the complexity of r and of f are separated. Note that in the case of an inner product the standard trick of delayed reduction can be applied.

Also of interest is the special case where $r = r_1 \cdot r_2$, for instance with r_1 and r_2 of degree smaller than m, since one can compute $r \bullet \mathbf{g}$ then either as $(r_1 \cdot r_2) \bullet \mathbf{g}$ or as $r_1 \bullet (r_2 \bullet \mathbf{g})$. If the first approach is used, the matrix method seems to be better, provided that f has only small eigenvalues.

6.4.2. Optimizing the Sharing. For the sharing over the extension ring the dealer has to pick a random polynomial of degree t with coefficients in G^m. The leading coefficient should correspond to the secret $s \in \mathsf{G}$, for instance by using $(s, 0, \ldots, 0) \in \mathsf{G}^m$ as leading coefficient. The polynomial is then evaluated in the points β_i for $i = 1, \ldots, n$.

The obvious approach for this evaluation is by repeated application of Horner's rule (over the module). For each participant this requires t module additions and t module multiplications with β_i. The total number of module multiplications is therefore $O(t)$. If the β_i are the $(0,1)$-polynomials and a 'nice' polynomial f is used, the polynomial approach for the module product yields that the sharing of the B-matrix will cost $O(tnm^2)$ ops.

6.4.3. Optimizing the Reconstruction. Given a qualified subset A, the first job during reconstruction is determining some $\delta_\beta \in R$ such that for all $i \in A$ the ring element δ_β is a multiple of $\prod_{j \in A, i \neq j} \beta_i - \beta_j$. The generic solution given by Cramer and Fehr is (a small multiple of) $\delta_\beta = \prod_{0<i<j\leq n} \beta_i - \beta_j$. Although this solution works for all sets A of any cardinality and regardless of the specifics of f and the β_i, there is a price to be paid. The product ranges over $n(n-1)/2$ factors most of which of degree only slightly smaller than m, resulting in an overall degree of $O(mn^2)$ for δ_β.

Cramer and Fehr also suggest the special choice of polynomials β_i being the $(0,1)$-polynomial evaluating to i in 2 for $i = 1, \ldots, n$ (or rather evaluating to $i-1$ since evaluation in 0 is otherwise not used). The difference of two $(0,1)$-polynomials is a $(-1,0,1)$-polynomial. There are $3^m = n^{\lg 3}$ such polynomials of degree smaller than m. The product of all these polynomials also suffices as δ_β for all sets A of any cardinality, but obviously only if the special β_i are used. The new δ_β has degree $O(mn^{\lg 3})$ and reconstruction takes $O(m^2 n^{1+\lg 3})$ ops.

Further practical improvement is possible by considering the irreducible factors of δ_β (regarded as polynomial over $\mathbb{Z}[X]$). The only factors that can occur are the factors of the $(-1,0,1)$-polynomials, so the set of possible factors is polynomially bounded (in n). For each irreducible polynomial the maximum power still necessarily dividing δ_β has to be determined. This power is computed by determining the maximum power required for the reconstruction for each participant and taking the maximum over all these powers. For a given participant the $n-1$ differences can easily be computed and for each of these differences how often they are divisible by the irreducible polynomial in question. The sum of the t largest values gives the maximum for the specific participant.

Unless n is a power of two, there is some freedom here which we will not exploit. The number of $(0,1)$-polynomials of degree smaller than m is larger than the number of evaluation points needed and picking different sets arguably leads to different runtimes (in fact, a similar argument holds for the integer sharing if n is not a prime). It is hard to see how this would affect the runtime significantly and how to cleverly make use of this relaxation. Note that different sets of evaluation points might also lead to slightly different conditions on the polynomial f that defines the extension degree.

Example 6.12. If $n = 5$ the degree of the extension is 3, so second degree polynomials are used for evaluation. Let us use the set $\{0, 1, X, 1+X, X^2\}$. The irreducible polynomials we need to consider are $X, X+1, X-1$ and X^2-X-1. For the polynomial X the table below shows that it appears at most three times. Using similar tables for the other irreducible polynomials leads to $\delta_\beta = X^3(X+1)(X-1)^2(X^2-X-1)$.

To put this into perspective, the product of all relevant $(-1,0,1)$-polynomials gives $\delta_\beta = XX^2(X^2-1)(X-1)(X+1)(X^2-X-1)$, which is a factor $X+1$ bigger. The method by Cramer and Fehr would yield a δ_β of degree 14, which is already almost twice as much as the degree 8 of the improved δ_β.

	0	1	X	$1+X$	X^2	$\sum(t=2)$
0	–	0	1	0	2	3
1	0	–	0	1	0	1
X	1	0	–	0	1	2
$1+X$	0	1	0	–	0	1
X^2	2	0	1	0	–	3

For reasons that will become clear in Example 6.15 we will use $\delta_\beta = X^3(X+1)(X-1)^2(X^2-X-1)(X^2-X+1)$ in our next examples.

6.5. Combining the Two Sharings

Suppose the players have already reconstructed $r_\alpha = \delta_\alpha s$ and $r_\beta = \delta_\beta s$ with both δ_α and δ_β positive integers. In order to obtain the secret s, values a and b have to be found such that $a\delta_\alpha + b\delta_\beta = 1$. The extended Euclidean algorithm provides unique a and b satisfying $|a| < \delta_\beta$ and $|b| < \delta_\alpha$. Given these a and b the players can then recover the secret as $s = ar_\alpha + br_\beta$.

Although δ_α will always be an integer, there is no a priori reason why δ_β should. It is an element in $\mathbb{Z}[X]/(f)$ for sure, but the representation of δ_β in $\mathbb{Z}[X]$ that is used can be a (non-constant) polynomial, as shown by Example 6.12. In this case $\mathbf{r}_\beta = \delta_\beta \bullet (s, 0, \ldots 0)$ is recovered and there are several ways to combine r_α and $\mathbf{r}_\beta$: one can find a representation of δ_β that is an integer, or one can directly construct a and b in the extension ring R such that $a\delta_\alpha + b\delta_\beta \equiv 1 \mod f$. Cramer and Fehr suggest the second method, which we will describe in more detail first. We were unable to make the first method competitive.

There might be an intriguing third method based on the observation that $\mathbf{r}_\beta = (s, 0, \ldots, 0)[\delta_\beta] = (\delta_0 s, \ldots, \delta_{t-1} s)$ for certain integers δ_i. If these δ_i are coprime, it is possible to base the reconstruction on standard Euclidean techniques without thinking about the ring $\mathbb{Z}[X]/(f)$ any more. This would make the integer sharing redundant for the reconstruction, leading to the optimal expansion rate $\lfloor \lg n \rfloor + 1$.

6.5.1. Direct Method. Let $\delta_\beta, \delta_\alpha$, and f be given for some n. By construction, δ_α factors into primes smaller than n, say $\delta_\alpha = \prod_{p<n, p \text{ prime}} p^{e_p}$. To determine a and b that satisfy $a\delta_\alpha + b\delta_\beta \equiv 1 \mod f$ we first determine for all $p < n$ a polynomial b_p satisfying $b_p\delta_\beta \equiv 1 \bmod (p^{e_p}, f)$. Application of the Chinese Remainder Theorem yields b from which a can be computed by $a = (b\beta - 1 \mod f)/\delta_\alpha$. Determining b_p can be done by computing $b_p \bmod (p, f)$ first. If f is irreducible, this is equivalent to finding the multiplicative inverse of δ_β in $\mathbb{F}_{p^m}$ (where $m = \deg f$). If f factors modulo p the computation can be performed modulo the irreducible factors of f first after which the Chinese Remainder Theorem yields the desired $b_p \bmod (p, f)$.

A necessary and sufficient condition for the existence of b_p that follows from the derivation above, is that δ_β should be invertible modulo the irreducible factors of f modulo p for all primes $p < n$. Since δ_β is basically the product of the differences $\beta_i - \beta_j$ this immediately translates to invertibility of $\beta_i - \beta_j$ (modulo ...), a result that was also derived in the proof of Lemma 6.5.

The resulting polynomials a and b both have degree at most $m-1$. The Chinese Remainder Theorem implies that the coefficients of b can be chosen between 0 and δ_α. The coefficients of a have roughly the same size as the coefficients of δ_β after reduction modulo f.

EXAMPLE 6.13. We continue our $n = 5$ and $t = 2$ example. In the Examples 6.8, 6.12, and 6.10, we came across $\delta_\alpha = 2^3 \cdot 3$, $\delta_\beta = 2X^2 - 3X$ and $f = X^3 - X - 1$ (where δ_β has already been reduced modulo f). We begin by determining b_2. The inverse of δ_β modulo 2 and f is the inverse of X. From $X^3 - X - 1 \equiv 0 \bmod f$ this inverse can be seen to be $X^2 + 1$ modulo 2. From $(X^2+1)\delta_\beta \equiv 4X^2 - 4X - 3 = 1 + 4(X^2 - X - 1) \bmod f$ it follows that $b_p \equiv X^2 + 1 \bmod 4$ as well, but modulo 8 something more has to be done. However, $(1+4(X^2-X-1))(1-4(X^2-X-1) \equiv 1 \bmod 16$, so we can take $b_2 = (X^2+1)(1+4(X^2-X-1)) \equiv 5X^2+4X+1 \bmod (f, 8)$. Determining $b_3 = X^2 + 2X + 2$ requires only one step. The Chinese Remainder Theorem can be applied on each of the coefficients individually, giving $b = 13X^2 + 20X + 17$. A small computation shows that $a = X$.

Recapitulating, the players have reconstructed $r_\alpha = \delta_\alpha s \in \mathsf{G}$ and $\mathbf{r}_\beta = \delta_\beta \bullet (s, 0, \ldots, 0) \in \mathsf{G}^m$ and computed a and b in the extension ring R such that $a\delta_\alpha + b\delta_\beta \equiv 1 \bmod f$. The secret s can be obtained by computing $a \bullet (r_\alpha, 0, \ldots, 0) + b \bullet \mathbf{r}_\beta = (s, 0, \ldots, 0)$. Clearly there is no need to compute all the zeros that occur in the vector $(s, 0, \ldots, 0)$. Writing $(r_\alpha, 0, \ldots, 0)[a] + \mathbf{r}_\beta[b]$ for the module operations reveals that only the first columns of $[a]$ and $[b]$ are of interest. In fact, of $[a]$ only the first entry of the first column, corresponding to the constant coefficient of the polynomial a, is needed because the other entries will only contribute multiples of $0 \in \mathsf{G}$.

EXAMPLE 6.14. Continuing Example 6.13, we first determine the matrices $[a]$ and $[b]$ based on $a = X$, $b = 13X^2 + 20X + 17$, and $f = X^3 - X - 1$. From Example 6.10 we already know $[b]$ and $[a] = F$ in this specific case.

$$[a] = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}, \qquad [b] = \begin{pmatrix} 17 & 20 & 13 \\ 13 & 17 & 33 \\ 33 & 13 & 50 \end{pmatrix}.$$

It follows that $(r_\alpha, 0, \ldots, 0)[a] = (0, r_\alpha, 0)$. Only the first coefficient is interesting, but it is always zero! In other words, the integer sharing is not used in the reconstruction and is redundant. The reconstruction of the secret is based on the inner product of $\mathbf{r}_\beta$ and the first column of $[b]$. This can be computed in 11 ops as we already saw.

6.5.2. Using the Resultant. The second approach mentioned to compute a and b satisfying $a\delta_\alpha + b\delta_\beta \equiv 1 \bmod f$ is by first determining b' satisfying that $b'\delta_\beta \bmod f$ is an integer (or constant polynomial) after which standard Euclidean techniques over $\mathbb{Z}$ can be used to determine a and b.

The polynomial b' satisfies $b'\delta_\beta + kf = m$ for some $k \in \mathbb{Z}[X]$ and $m \in \mathbb{Z}$. If δ_β and f were defined over a finite field, the extended Euclidean algorithm could be used to compute b' and k (with $m = 1$, since δ_β and f are relatively prime). The Euclidean algorithm is not very well suited for polynomials over $\mathbb{Z}$, since the

coefficients of intermediate polynomials can (and typically will) become very big. If the subresultant algorithm is used to compute the greatest common divisor of two polynomials over $\mathbb{Z}$, all intermediate results are relatively small (for more details we refer to [**45, 90, 199**]). An extended version of the subresultant algorithm returns polynomials b' and k as well such that $b'\delta_\beta + kf$ is an integral multiple of the greatest common divisor of δ_β and f (in our case this gcd equals 1 of course).

It is almost inevitable that $b'\delta_\beta + kf$ is a multiple of the greatest common divisor. As small illustration suppose that $\delta_\beta = 2$ and $f = X$. Their greatest common divisor is 1 but clearly whenever $b2 + kX$ is an integer, it will be an even integer. For coprime δ_β and f, the extended subresultant algorithm actually returns b' and k such that $b'\delta_\beta + kf = \mathrm{Res}(\delta_\beta, f)$, where $\mathrm{Res}(\delta_\beta, f)$ is the resultant of the two polynomials δ_β and f. Unfortunately, the resultant of δ_β and f will be quite big, which makes this method unattractive.

EXAMPLE 6.15. For $n = 5$ and $t = 2$ the resultant of $\delta_\beta = 2X^2 - 3X$ and $f = X^3 - X - 1$ is -7 and $(1 + 4X + 5X^2)\delta_\beta + (7 - 10X)f = -7$. The numbers are still quite low, but this is atypical for this method.

The fact that the resultant is -7 means that δ_β is not invertible in R, which makes the redundancy of the integer sharing all the more surprising. However, if the minimal δ_β would have been used (we put in an extra factor $X^2 - X + 1$ in Example 6.12) the resultant is 1 (so the set β_i is exceptional). Nevertheless this example is significant, since it shows that full reconstruction of the secret is possible based solely on the extension ring, even though not all Lagrangian denominators are invertible.

6.5.3. Trade-Offs. If δ_α and δ_β are independent of the qualified set, the dealer could compute a and b and multiply these into the secret, so that reconstruction immediately yields $a\delta_\beta s$ and $(b\delta_\beta) \bullet (s, 0, \ldots, 0)$.

There is some trade-off here, since in the first case the participants need only to reconstruct and multiply with a and b, where in the second case the dealer can multiply with a and b, so the participants need only to reconstruct and multiply with the cofactors of the lcm's and the actual share product.

6.6. Picking the Extension

One of the things that is important is an estimate on the coefficients of the polynomials f that satisfy Lemma 6.5. Presumably, the smaller the coefficients, the less black box calls in the resulting secret sharing scheme. We stress however that giving an indication for the 'optimal' f and finding it are two entirely different things.

One method always works, namely picking random irreducible polynomials modulo p for all the primes $p < n$ and use the Chinese Remainder Theorem to get a polynomial over the integer. The coefficients of this polynomial are all smaller than $\prod_{p<n} p$, which corresponds to a bitlength linear in n (which is acceptable). It is in fact possible to have $f - X^m$ of reasonably low degree which is benificial as shown in Section 6.4.1.

TABLE 6.1. Suggested polynomials

n	f	λ_f	Reduction costs
3–4	$X^2 - X - 1$	1.61	1
5	$X^3 - X - 1$	1.32	2
6–7	$X^3 - X - 7$	2.09	12
8	$X^3 + 11X - 1$	3.32	15
9–16	$X^4 - 19X - 1$	2.69	25
17	$X^5 - 17X^2 - 2X - 1$	2.62	57
18–19	$X^5 - 3X^2 + 14X + 19$	2.08	60
20–31	$X^5 + 3X^2 + 68X - 41$	2.93	69
32	$X^5 - 93X^2 + 38X + 1$	4.60	111
33–37	$X^6 - 90X^2 - 121X - 103$	3.40	159
38–41	$X^6 - 170X^2 - 13X + 43$	3.62	148
42–47	$X^6 - 174X^2 + 289X - 229$	4.02	169

To get an impression how much improvement there is possible, we assume irreducible polynomials behave sufficiently random. We begin by recalling that the probability that a randomly chosen (monic) polynomial of degree m is irreducible modulo p is about $1/p$. Hence, the probability that a random polynomial over $\mathbb{Z}$ is irreducible modulo all primes $p < n$ is about $\prod_{p<n} \frac{1}{p}$. It is therefore reasonable to assume we need to look at about $\prod_{p<n} p$ polynomials before we find an irreducible one. The number of polynomials with all coefficients smaller then B is B^m. In order for $B^m > \prod_{p<n} p$ we require that $B = O(n/m)$. Since $m \approx \lg n$, we cannot hope to find polynomials that are that much better than random CRT based polynomials.

Using Lenstra's relaxation gives us different probabilities. For instance, for the larger primes the condition $p^d \geq n$ will simply mean that f has no linear factors. This poses the natural question what the probability is that a randomly chosen (monic) polynomial of degree m has no linear factors modulo p (or no roots). The generalization to higher order factors (of degree at most $d-1$) also presents itself.

For small numbers of participant we performed a search for what we believe are nice polynomials. The results are summarized in Table 6.1. The runtime of the search was not polynomial time in the number of participants, so it could be considered as cheating. However, the polynomials only have to be picked once and there are no security concerns if everyone uses the same polynomials. The absolute value of the 'largest' eigenvalue of the polynomial and an upper bound on the length of an addition-subtraction chain for $[f]$ are also given.

Bibliography

[1] Adams and Shanks. Strong primality tests that are not sufficient. *Mathematics of Computation*, 39(159):255–300, 1982.

[2] G. Agnew, R. Mullin, and S. Vanstone. Fast exponentiation in $GF(2^n)$. In C. G. Günther, editor, *Advances in Cryptography—Eurocrypt'88*, volume 330 of *Lecture Notes in Computer Science*, pages 251–255. Springer-Verlag, 1988.

[3] G. Agnew, R. Mullin, and S. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, 1993.

[4] A. Akhavi and B. Vallée. Average bit-complexity of Euclidean algorithms. In U. Montanari, J. D. Rolim, and E. Welzl, editors, *ICALP'00*, volume 1853 of *Lecture Notes in Computer Science*, pages 374–387. Springer-Verlag, 2000.

[5] T. Akishita. Fast simultaneous scalar multiplication on elliptic curve with Montgomery form. In Vaudenay and Youssef [**193**], pages 255–268.

[6] R. M. Avanzi. On multi-exponentiation in cryptography. Technical Report 154, IACR's ePrint Archive, 2002.

[7] J. B. Kaliski. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995.

[8] E. Bach and J. Shallit. *Algorithmic Number Theory*. MIT Press, 1996.

[9] D. Bailey and C. Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.

[10] P. D. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In A. Odlyzko, editor, *Advances in Cryptography—Crypto'86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer-Verlag, 1987.

[11] P. Beelen, 2001. Personal communication.

[12] R. Bellman. Problems and solutions: (5125) addition chains of vectors. *American Mathematical Monthly*, 70:765, 1963.

[13] F. Bergeron, J. Berstel, and S. Brlek. Efficient computation of addition chains. *Journal de Théorie des Nombres de Bordeaux*, 6:21–38, 1994.

[14] F. Bergeron, J. Berstel, S. Brlek, and C. Duboc. Addition chains using continued fractions. *J. Algorithms*, 10:403–412, 1989.

[15] D. J. Bernstein. Multidigit multiplication for mathematicians. To appear in Advances of Applied Mathematics, 2000.

[16] D. J. Bernstein. Pippenger's exponentiation algorithm. Draft, 2002.

[17] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.

[18] G. R. Blakley. A computer algorithm for calculating the product AB modulo M. *IEEE Transactions on Computers*, 32(5):497–500, 1983.

[19] D. Bleichenbacher. *Efficiency and Security of Cryptosystems based on Number Theory*. PhD thesis, ETH Zürich, 1996.

[20] D. Bleichenbacher, W. Bosma, and A. K. Lenstra. Some remarks on Lucas-based cryptosystems. In D. Coppersmith, editor, *Advances in Cryptography—Crypto'95*, volume 936 of *Lecture Notes in Computer Science*, pages 306–316. Springer-Verlag, 1995.

[21] D. Bleichenbacher and A. Flammenkamp. An efficient algorithm for computing shortest addition chains. year unknown.

[22] D. Bleichenbacher, M. Joye, and J.-J. Quisquater. A new and optimal chosen-message attack on RSA-type cryptosystems. In Y. Han, T. Okamoto, and S. Qing, editors, *ICICS'97*, volume 1334 of *Lecture Notes in Computer Science*, pages 302–313. Springer-Verlag, 1997.

[23] I. E. Bocharova and B. D. Kudryashov. Fast exponentation in cryptography. In G. Cohen, M. Giustie, and T. Mora, editors, *AAECC-11*, volume 948 of *Lecture Notes in Computer Science*, pages 146–157. Springer-Verlag, 1995.

[24] D. Boneh. The decision Diffie-Hellman problem. In Buhler [**37**], pages 48–63.

[25] J. Bos and M. Coster. Addition chain heuristics. In Brassard [**30**], pages 400–407.

[26] J. N. E. Bos. *Practical Privacy*. PhD thesis, Technische Universiteit Eindhoven, 1992.

[27] W. Bosma, J. Hutton, and E. R. Verheul. Looking beyond XTR. In Y. Zheng, editor, *Advances in Cryptography—Asiacrypt'02*, volume 2501 of *Lecture Notes in Computer Science*, pages 46–63. Springer-Verlag, 2002.

[28] A. Bosselaers, R. Govaerts, and J. Vandewalle. Comparison of three modular reduction functions. In D. Stinson, editor, *Advances in Cryptography—Crypto'93*, volume 773 of *Lecture Notes in Computer Science*, pages 175–186. Springer-Verlag, 1993.

[29] C. Boyd, editor. *Advances in Cryptography—Asiacrypt'01*, volume 2248 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[30] G. Brassard, editor. *Advances in Cryptography—Crypto'89*, volume 435 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[31] A. Brauer. On addition chains. *Bulletin of the American Mathemetical Society*, 45:736–739, 1939.

[32] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation (extended abstract). In R. A. Rueppel, editor, *Advances in Cryptography—Eurocrypt'92*, volume 658 of *Lecture Notes in Computer Science*, pages 200–207. Springer-Verlag, 1992. Newer version in [**33**].

[33] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation: algorithms and lower bounds. Updated and expanded version of [**32**], 1995.

[34] É. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. In Naccache and Paillier [**135**], pages 335–345.

[35] A. Brouwer, R. Pellikaan, and E. Verheul. Doing more with fewer bits. In K. Y. Lam, E. Okamoto, and C. Xing, editors, *Advances in Cryptography—Asiacrypt'99*, volume 1716 of *Lecture Notes in Computer Science*, pages 321–332. Springer-Verlag, 1999.

[36] M. Brown, D. Hankerson, J. López, and A. Menezes. Software implementation of the nist elliptic curves over prime fields. In D. Naccache, editor, *CT-RSA'01*, volume 2020 of *Lecture Notes in Computer Science*, pages 250–265. Springer-Verlag, 2001.

[37] J. P. Buhler, editor. *Algorithmic Number Theory, 3rd International Symposium, ANTS-III*, volume 1423 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[38] J. Burton S. Kaliski, Ç. Koç, and C. Paar, editors. *Cryptographic Hardware and Embedded Systems 2002*, volume 2523 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[39] L. Carlitz. Recurrences of the third order and related combinatorial identities. *Fibonacci Quarterly*, 16(1):11–18, 1978.

[40] L. Carlitz. Some combinatorial identities of Bernstein. *SIAM J. Math. Anal*, 9(1):65–75, 1978.

[41] C.-Y. Chen and C.-C. Chang. A fast modular multiplication algorithm for calculating the product ab modulo n. *Information Processing Letters*, 72:77–81, 1999.

[42] C.-Y. Chen and T.-C. Liu. A fast modular multiplication method based on the Lempel-Ziv binary tree. *Computer Communications*, 22:871–874, 1999.

[43] C. W. Chiou and T. C. Yang. Iterative modular multiplication algorithm without magnitude comparison. *IEE Electronics Letters*, 30(24):2017–2018, 1994.

[44] G. Cohen, A. Lobstein, D. Naccache, and G. Zémor. How to improve an exponentiation black-box. In K. Nyberg, editor, *Advances in Cryptography—Eurocrypt'98*, volume 1403 of *Lecture Notes in Computer Science*, pages 211–220. Springer-Verlag, 1998.

[45] H. Cohen. *A Course in Computational Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer-Verlag, 1996.

[46] H. Cohen. Analysis of the flexible window powering algorithm. Submitted for publication, available from `http://www.math.u-bordeaux.fr/~cohen`, 2001.

[47] H. Cohen and A. K. Lenstra. Supplement to implementation of a new primality test. *Mathematics of Computation*, 48(177): S1–S4, 1987.

[48] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In K. Ohta and D. Pei, editors, *Advances in Cryptography—Asiacrypt'98*, volume 1514 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, 1998.

[49] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Transactions on Information Theory*, 30(4):587–594, 1984.

[50] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.*, 9(3):251–280, 1990.

[51] M. Coster. The algorithm of Brun and addition chains, 2000. Manuscript.

[52] R. Cramer and S. Fehr. Optimal black-box secret sharing over arbitrary abelian groups. In M. Yung, editor, *Advances in Cryptography—Crypto'02*, volume 2442 of *Lecture Notes in Computer Science*, pages 272–287. Springer-Verlag, 2002.

[53] R. Cramer and S. Fehr. Optimal black-box secret sharing over arbitrary abelian groups. Manuscript, 2003.

[54] R. Cramer, S. Fehr, Y. Ishai, and . Kushilevitz. Efficient multi-party computation over rings. Manuscript, Feb. 2002.

[55] R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In H. Krawczyk, editor, *Advances in Cryptography—Crypto'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25. Springer-Verlag, 1998.

[56] P. J. N. de Rooij. Efficient exponentiation using precomputation and vector addition chains. In A. D. Santis, editor, *Advances in Cryptography—Eurocrypt'94*, volume 950 of *Lecture Notes in Computer Science*, pages 389–399. Springer-Verlag, 1994.

[57] Y. Desmedt and Y. Frankel. Threshold cryptosystem. In Brassard [**30**], pages 307–315.

[58] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[59] P. Downey, B. Leong, and R. Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 10:638–646, 1981.

[60] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[61] R. J. Fateman. Lookup tables, recurrences and complexity. In G. H. Gonnet, editor, *Proceedings of the ACM-SIGSAM 1989 international symposium on Symbolic and algebraic computation*, pages 68–73. ACM Press, 1989.

[62] C. M. Fiduccia. On obtaining upper bounds on the complexity of matrix multiplication. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 31–40, 1972.

[63] C. M. Fiduccia. An efficient formula for linear recurrences. *SIAM Journal on Computing*, 14(1):106–112, 1985.

[64] R. Gallant, R. Lambert, and S. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In J. Kilian, editor, *Advances in Cryptography—Crypto'01*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200. Springer-Verlag, 2001.

[65] K. J. Giuliani and G. Gong. Generating large instances of the gong-harn cryptosystem. Technical report, CACR (University of Waterloo) preprint series, 2002.

[66] S. Goldwasser and M. Bellare. Lecture notes on cryptography, 1999.

[67] D. Gollman, Y. Han, and C. Mitchell. Redundant integer representations and fast exponentiation. *Designs, Codes and Cryptography*, 7:135–151, 1996.

[68] G. Gong and L. Harn. Public-key cryptosystems based on cubic finite field extensions. *IEEE Transactions on Information Theory*, 45(7):2601–2605, 1999.

[69] G. Gong, L. Harn, and H. Wu. The GH public-key cryptosystems. In Vaudenay and Youssef [**193**], pages 284–300.

[70] D. M. Gordon. Discrete logarithms in GF(p) using the number field sieve. *SIAM J. Discrete Math.*, 6(1):124–138, 1993.

[71] D. M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, 1998.

[72] R. L. Graham, A. C. Yao, and F. F. Yao. Addition chains with multiplicative cost. *Discrete Mathematics*, 23:115–119, 1978.

[73] D. Gries and G. Levin. Computing Fibonacci numbers (and similarly defined functions) in log time. *Information Processing Letters*, 11(2):68–69, 1980.

[74] D. Hankerson, J. López Hernandez, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. In Ç. Koç and C. Paar, editors, *CHES'00*, volume 1965 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, 2000.

[75] P. Horster, H. Petersen, and M. Michels. Meta-ElGamal signature schemes. In *CCS'94*, pages 96–107. ACM Press, 1994.

[76] T. Izu and T. Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In Naccache and Paillier [**135**], pages 280–296.

[77] D. Johnson and A. Menezes. The elliptic curve digital signature algorithm (ECDSA). Technical report, CACR (University of Waterloo) preprint series, 1999. Technical Report CORR 99-31.

[78] M. Joye. *Security Analysis of RSA-type Cryptosystems*. PhD thesis, Université Catholique de Louvain, 1997.

[79] M. Joye and J.-J. Quisquater. Protocol failures for RSA-like functions using lucas sequences and elliptic curves. In M. Lomas, editor, *Security Protocols*, volume 1189 of *Lecture Notes in Computer Science*, pages 93–100. Springer-Verlag, 1996.

[80] M. Joye and S.-M. Yen. Optimal left-to-right binary signed-digit recoding. *IEEE Transactions on Computers*, 49(7):740–748, 2000.

[81] M. Joye and S.-M. Yen. The Montgomery powering ladder. In Burton S. Kaliski et al. [**38**], pages 291–302.

[82] J. K. R. Sloan. Comments on "a computer algorithm for calculating the product ab modulo m". *IEEE Transactions on Computers*, 34(3):290–292, 1985.

[83] M. Kaminski. An algorithm for polynomial multiplication that does not depend on the ring constants. *J. Algorithms*, 9(1):137–147, 1988.

[84] M. Kaminski, D. G. Kirkpatrick, and N. H. Bshouty. Addition requirements for matrix and transposed matrix products. *Journal of Algorithms*, 9(3):354–364, 1988.

[85] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[86] B. King. An improved implementation of elliptic curves over GF(2^n) when using projective point arithmetic. In Vaudenay and Youssef [**193**], pages 134–150.

[87] B. S. King. *Some Results in Linear Secret Sharing*. PhD thesis, University of Wisconsin-Milwaukee, 2000.

[88] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison Wesley, 1 edition, 1969.

[89] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison Wesley, 3 edition, 1997.

[90] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison Wesley, 3 edition, 1997.

[91] D. E. Knuth, editor. *Selected papers on analysis of algorithms*. CSLI Publications, Stanford, 2000.

[92] D. E. Knuth and C. H. Papadimitriou. Duality in addition chains. *Bulletin of the European Association for Theoretical Computer Science*, 13:2–4, 1981. Reprinted in [**91**, Chapter 31].

[93] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[94] N. Koblitz. Hyperelliptic curve cryptosystems. *Journal of Cryptology*, 1:139–150, 1989.

[95] W. Koepf. Efficient computation of Chebyshev polynomials. In M. Wester, editor, *Computer Algebra Systems: A Practical Guide*, pages 79–99. John Wiley, 1999.

[96] C. Krook. Optimal black-box secret sharing over arbitrary abelian groups. Available on-line, http://www.student.tue.nl/u/c.krook, 2002.

[97] M. Kutz. Lower bounds for Lucas chains. 3rd revised edition. Available from the author, 2002.

[98] C. Laih, W. Tai, and F. Tu. On the security of LUC function. *Information Processing Letters*, 53:243–247, 1995.

[99] C.-S. Laih, F.-K. Tu, and W.-C. Tai. Remarks on LUC public key system. *IEE Electronics Letters*, 30(2):123–124, 1994.

[100] T. Lange. *Efficient Arithmetic on Hyperelliptic Curves*. PhD thesis, Universität-Gesamthochschule Essen, 2001.

[101] D. H. Lehmer. Computer technology applied to the theory of numbers. In W. J. LeVeque, editor, *Studies in Number Theory*, volume 6 of *MAA Studies in Mathematics*, pages 117–151. Math. Assoc. Amer. (distributed by Prentice-Hall, Englewood Cliffs, N.J.), 1969.

[102] A. Lempel, G. Seroussi, and S. Winograd. On the complexity of multiplication in finite fields. *Theoretical Computer Science*, 22:285–296, 1983.

[103] A. K. Lenstra. The long integer package freelip. Available from www.ecstr.com.

[104] A. K. Lenstra. Using cyclotomic polynomials to construct efficient discrete logarithm cryptosystems over finite fields. In V. Varadharajan, J. Pieprzyk, and Y. Mu, editors, *ACISP'97*, volume 1270 of *Lecture Notes in Computer Science*, pages 127–138. Springer-Verlag, 1997.

[105] A. K. Lenstra. Unbelievable security: matching AES security using public key systems. In Boyd [**29**], pages 67–86.

[106] A. K. Lenstra and E. R. Verheul. Key improvements to XTR. In T. Okamoto, editor, *Advances in Cryptography—Asiacrypt'00*, volume 1976 of *Lecture Notes in Computer Science*, pages 220–233. Springer-Verlag, 2000.

[107] A. K. Lenstra and E. R. Verheul. The XTR public key system. In M. Bellare, editor, *Advances in Cryptography—Crypto'00*, volume 1880 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2000.

[108] A. K. Lenstra and E. R. Verheul. Fast irreducibility and subgroup membership testing in XTR. In K. Kim, editor, *PKC'01*, volume 1992 of *Lecture Notes in Computer Science*, pages 73–86. Springer-Verlag, 2001.

[109] A. K. Lenstra and E. R. Verheul. An overview of the XTR public key system. In K. Alster, J. Urbanowicz, and H. C. Williams, editors, *The proceedings of the Public-Key Cryptography and Computational Number Theory Conference*, pages 151–180. Verlages Walter de Gruyter, 2001.

[110] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

[111] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, 1994.

[112] C. H. Lim and P. J. Lee. More flexible exponentiation with precomputation. In Y. Desmedt, editor, *Advances in Cryptography—Crypto'94*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107. Springer-Verlag, 1994.

[113] S. Lim, S. Kim, I. Yie, J. Kim, and H. Lee. XTR extended to GF(p^{6m}). In Vaudenay and Youssef [**193**], pages 301–312.

[114] J. López and R. Dahab. Improved arithmetic for elliptic curve arithmetic in GF(2^m). In S. Tavares and H. Meijer, editors, *SAC'98*, volume 1556 of *Lecture Notes in Computer Science*, pages 201–212. Springer-Verlag, 1998.

[115] J. López and R. Dahab. Fast multiplication on elliptic curves over GF(2^m) without precomputation. In Ç. K. Koç and C. Paar, editors, *CHES'99*, volume 1717 of *Lecture Notes in Computer Science*, pages 316–327. Springer-Verlag, 1999.

[116] E. Lucas. *Le calcul des nombres entiers, le calcul des nombres rationnels, la divisibilité arithmétique*, volume 1 of *Th/'eorie des Nombres*. Gauthier-Villars et Fils, Paris, 1891.

[117] O. B. Lupanov. On rectifier and contact rectifier circuits. *Dokl. Akad. Nauk SSSR (N.S.)*, 111:1171–1174, 1956.

[118] A. J. Martin and M. Rem. A presentation of the Fibonacci algorithm. *Information Processing Letters*, 19(2):67–68, 1984.

[119] D. P. McCarthy. The optimal algorithm to evaluate x^n using elementary multiplication methods. *Mathematics of Computation*, 31(137):251–256, 1977.

[120] K. S. McCurley. The discrete logarithm problem. In C. Pomerance, editor, *Cryptology and Computational Number Theory*, volume 42 of *Proceedings of Symposia in Applied Mathematics*, pages 49–74. American Mathematical Society, 1990.

[121] A. Menezes, P. van Oorschot, and S. Vanstone. *CRC-Handbook of Applied Cryptography*. CRC Press, 1996.

[122] A. Menezes and S. Vanstone. ECSTR (XTR): Elliptic curve singular trace representation. Rump Session of Crypto 2000, 2000.

[123] A. J. Menezes and S. A. Vanstone. Elliptic curve cryptosystems and their implementation. *Journal of Cryptology*, 6(4):209–224, 1993.

[124] M. Michels. *Kryptologische Aspekte digitaler Signaturen und elektronischer Wahlen*. PhD thesis, Technischen Universität Chemnitz-Zwickau, Aachen (Germany), 1996.

[125] J. Miller and D. S. Brown. An algorithm for evaluation of remote terms in a linear recurrence sequence. *The Computer Journal*, 9:188–190, 1966.

[126] V. Miller. Use of elliptic curves in cryptography. In H. Williams, editor, *Advances in Cryptography—Crypto'85*, volume 218 of *Lecture Notes in Computer Science*, pages 417–425. Springer-Verlag, 1986.

[127] B. Möller. Algorithms for multi-exponentiation. In Vaudenay and Youssef [**193**], pages 165–180.

[128] P. L. Montgomery. Evaluating recurrences of form $X_{m+n} = f(X_m, X_n, X_{m-n})$ via Lucas chains. Revised (1992) version from ftp.cwi.nl: /pub/pmontgom/Lucas.ps.gz, 1983.

[129] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[130] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(170):243–264, 1987.

[131] P. L. Montgomery, Aug. 2000. Private communication: *expon2.txt, Dual elliptic curve exponentiation*.

[132] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *RAIRO Inform. Theor. Appl.*, pages 531–543, 1990.

[133] W. Müller and R. Nöbauer. Some remarks on public-key cryptosystems. *Studia Sci. Math. Hungar.*, 16:71–76, 1981.

[134] W. B. Müller and R. Nöbauer. Cryptanalysis of the Dickson-scheme. In F. Pichler, editor, *Advances in Cryptography—Eurocrypt'85*, volume 219 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[135] D. Naccache and P. Paillier, editors. *Public Key Cryptography*, volume 2274 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[136] National Institute of Standards and Technology. *Digital Signature Standard*, 2000. FIPS Publication 186-2.

[137] N. Nedjah and L. de Macedo Mourelle. Minimal addition chain for efficient modular exponentiation using genetic algorithms. In T. Hendtlass and M. Ali, editors, *IEA/AIE 2002*, volume 2358 of *LNAI*, pages 88–89. Springer-Verlag, 2002.

[138] L. J. O'Connor. On string replacement exponentiation. *Designs, Codes and Cryptography*, 23:173–183, 2001.

[139] A. M. Odlyzko. Discrete logarithms: The past and the future. *Designs, Codes and Cryptography*, 19:129–145, 2000.

[140] K. Okeya, H. Kurumatani, and K. Sakurai. Elliptic curves with the Montgomery-form and their cryptographic applications. In H. Imai and Y. Zheng, editors, *PKC'00*, volume 1751 of *Lecture Notes in Computer Science*, pages 238–257. Springer-Verlag, 2000.

[141] K. Okeya and K. Sakurai. Fast multi-scalar multiplication methods on elliptic curves with precomputation strategy using Montgomery trick. In Burton S. Kaliski et al. [**38**], pages 564–578.

[142] J. Olivos. On vectorial addition chains. *Journal of Algorithms*, 2:13–21, 1981.

[143] A. Ostrowski. On two problems in abstract algebra connected with Horner's rule. In *Studies in mathematics and mechanics presented to Richard von Mises*, pages 40–48. Academic Press Inc., New York, 1954.

[144] Y.-H. Park, S. Jeong, and J. Lim. Fast exponentiation in subgroups of finite fields. *IEE Electronics Letters*, 38(13):629–630, 2002.

[145] R. Perrin. Query 1484. *L'Intermédiaire des Mathématiciens*, 6:76, 1899.

[146] B. Pfitzmann, editor. *Advances in Cryptography—Eurocrypt'01*, volume 2045 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[147] R. Pinch. Extending the Hastad attack to LUC. *IEE Electronics Letters*, 31(21):1827–1828, 1995.

[148] R. Pinch. Extending the Wiener attack to RSA-type cryptosystems. *IEE Electronics Letters*, 31(20):1736–1738, 1995.

[149] N. Pippenger. On the evaluation of powers and related problems. In *Proceedings of 17th Annual IEEE Symposium on Foundations of Computer Science (FOCS'76)*, pages 258–263. IEEE Computer Society, 1976.

[150] N. Pippenger. The minimum number of edges in graphs with prescribed paths. *Mathematical Systems Theory*, 12(4):325–346, 1979.

[151] N. Pippenger. On the evaluation of powers and monomials. *SIAM Journal on Computing*, 9(2):230–250, 1980.

[152] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. *IEEE Transactions on Information Theory*, 24:106–110, 1978.

[153] J. Pollard. Monte Carlo methods for index computation ($\mathrm{mod}\, p$). *Mathematics of Computation*, 32(143):918–924, 1978.

[154] J. Pollard. Kangaroos, monopoly and discrete logarithms. *Journal of Cryptology*, 13(4):437–447, 2000.

[155] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. of the ACM*, 21(2):120–126, 1978.

[156] S. Rönn. On the logarithmic evaluation of recurrence relations. *Information Processing Letters*, 40(4):197–199, 1991.

[157] K. Rubin and A. Silverberg. Torus-based cryptography. Technical Report 39, IACR's ePrint Archive, 2003.

[158] A.-R. Sadeghi and M. Steiner. Assumptions related to discrete logarithms: Why subtleties make a real difference. In Pfitzmann [**146**], pages 244–261. Full version online.

[159] O. Schirokauer, D. Weber, and T. F. Denny. Discrete logarithms: the effectiveness of the index calculus method. In H. Cohen, editor, *ANTS II*, volume 1122 of *Lecture Notes in Computer Science*, pages 337–361. Springer-Verlag, 1996.

[160] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., 2 edition, 1996.

[161] C. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4:161–174, 1991.

[162] B. Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In M. Wiener, editor, *Advances in Cryptography—Crypto'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 148–164. Springer-Verlag, 1999.

[163] B. Schoenmakers, Aug. 2000. Personal communication.

[164] A. Scholz. Aufgabe 253. *Jahresbericht der deutschen Mathematiker-Vereinigung*, 47:41–42, 1937.

[165] I. Semba. Systematic method for determining the number of multiplications required to compute x^m, where m is a positive integer. *J. Information Proc.*, 6:31–33, 1983.

[166] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[167] D. Shanks. Class number, a theory of factorization, and genera. In *1969 Number Theory Institute (Proc. Sympos. Pure Math., Vol. XX, State Univ. New York, Stony Brook, N.Y., 1969)*, pages 415–440, Providence, R.I., 1971. Amer. Math. Soc.

[168] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423; 623–656, 1948. Also appears in [**170**].

[169] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656–715, 1949. Also appears in [**170**]. The material originally appeared in a confidential report 'A Mathematical Theory of Cryptography', dated Sept. 1, 1945.

[170] C. E. Shannon. *Claude Elwood Shannon.* IEEE Press, New York, 1993. Collected papers, Edited by N. J. A. Sloane and Aaron D. Wyner.

[171] J. Shortt. An iterative program to calculate Fibonacci numbers in $O(\log n)$ arithmetic operations. *Information Processing Letters*, 7(6):299–303, 1978.

[172] V. Shoup. Lower bounds for discrete logarithms and related problems. In W. Fumy, editor, *Advances in Cryptography—Eurocrypt'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266. Springer-Verlag, 1997.

[173] V. Shoup. Practical threshold signatures. In B. Preneel, editor, *Advances in Cryptography—Eurocrypt'00*, volume 1807 of *Lecture Notes in Computer Science*, pages 207–220. Springer-Verlag, 2000.

[174] F. Sica, M. Ciet, and J.-J. Quisquater. Analysis of the Gallant-Lambert-Vanstone method based on efficient endomorphisms: Elliptic and hyperelliptic curves. In K. Nyberg and H. Heys, editors, *SAC'02*, volume 2595 of *Lecture Notes in Computer Science*, pages 21–36. Springer-Verlag, 2003.

[175] P. Smith and C. Skinner. A public-key cryptosystem and a digital signature system based on the Lucas function analogue to discrete logarithms. In J. Pieprzyk and R. Safavi-Naini, editors, *Advances in Cryptography—Asiacrypt'94*, volume 917 of *Lecture Notes in Computer Science*, pages 357–364. Springer-Verlag, 1995.

[176] P. J. Smith and M. J. J. Lennon. LUC: A new public key system. In E. G. Douglas, editor, *Proceedings of the Ninth IFIP International Symposium on Computer Security*, pages 97–111. Elsevier Science Publications, 1993.

[177] J. A. Solinas. An improved algorithm for arithmetic on a family of elliptic curves. In J. Burt Kaliski and S. Burton, editors, *Advances in Cryptography—Crypto'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 357–371. Springer-Verlag, 1997.

[178] J. A. Solinas. Low-weight binary representations for pairs of integers. Technical report, CACR (University of Waterloo) preprint series, 2001.

[179] M. Stam. On Montgomery-like representations for elliptic curves over $GF(2^k)$. In Y. Desmedt, editor, *PKC'03*, volume 2567 of *Lecture Notes in Computer Science*, pages 240–253. Springer-Verlag, 2002.

[180] M. Stam and A. K. Lenstra. Speeding up XTR. In Boyd [**29**], pages 125–143.

[181] M. Stam and A. K. Lenstra. Efficient subgroup exponentiation in quadratic and sixth degree extensions. In Burton S. Kaliski et al. [**38**], pages 318–332.

[182] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[183] V. Strassen. Berechnung und programm. i. *Acta Informatica*, 1:320–335, 1972.

[184] V. Strassen. Some results in algebraic complexity theory. In *Proceedings of the International Congress of Mathematicians, Vancouver, 1974*, pages 497–501. Canadian Mathematical Congress, 1975.

[185] E. G. Straus. Problems and solutions: (5125) addition chains of vectors. *American Mathematical Monthly*, 71:806–808, 1964.

[186] E. G. Thurber. Efficient generation of minimal length addition chains. *SIAM Journal on Computing*, 28(4):1247–1263, 1999.

[187] Y. Tsuruoka. Computing short Lucas chains for elliptic curve cryptosystems. *IEICE Trans. Fundamentals*, E84–A(5):1227–1233, 2001.

[188] B. P. Tunstall. *Synthesis of Noiseless Compression Codes*. PhD thesis, Georgia Institute of Technology, 1968.

[189] F. J. Urbanek. An $O(\log n)$ algorithm for computing the nth element of the solution of a difference equation. *Information Processing Letters*, 11(2):66–67, 1980.

[190] B. Vallée. Dynamics of the binary Euclidean algorithm: functional analysis and operators. *Algorithmica*, 22:660–685, 1998.

[191] J. H. van Lint. *Introduction to Coding Theory*, volume 86 of *Graduate Texts in Mathematics*. Springer-Verlag, 3 edition, 1999.

[192] S. A. Vanstone, R. C. Mullin, A. Antipa, and R. Gallant. Accelerated finite field operations on an elliptic curve. Technical Report WO 99/49386, Patent Cooperation Treaty, 1999.

[193] S. Vaudenay and A. Youssef, editors. *Selected Areas in Cryptography 2001*, volume 2259 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[194] E. R. Verheul. Evidence that XTR is more secure than supersingular elliptic curve cryptosystems. In Pfitzmann [**146**], pages 195–210.

[195] J. von zur Gathen. Efficient and optimal exponentiation in finite fields. *Comput. Complexity*, 1:360–394, 1991.

[196] J. von zur Gathen and M. Nöcker. Exponentiation in finite fields: theory and practice. In T. Mora and H. Mattson, editors, *AAECC-12*, volume 1255 of *Lecture Notes in Computer Science*, pages 88–133. Springer-Verlag, 1997.

[197] C.-T. Wang, C.-C. Chang, and C.-H. Lin. A method for computing lucas sequences. *Computers and Mathematics with Applications*, 38:187–196, 1999.

[198] E. D. Win, S. Mister, B. Preneel, and M. Wiener. On the performance of signature schemes based on elliptic curves. In Buhler [**37**], pages 252–266.

[199] F. Winkler. *Polynomial Algorithms in Computer Algebra*. Springer-Verlag, 1996.

[200] Y. Yacobi. Fast exponentiation using data compression. *SIAM Journal on Computing*, 28(2):700–703, 1998.

[201] A. C. Yao and D. E. Knuth. Analysis of the subtractive algorithm for greatest common divisors. *Proc. Nat. Acad. Sci. U.S.A.*, 72(12):4720–4722, 1975.

[202] A. C.-C. Yao. On the evaluation of powers. *SIAM Journal on Computing*, 5(1):100–103, 1976.

[203] S. Yen and C. Laih. Fast algorithms for LUC digital signature computation. *IEE Proceedings, Computers and Digital Techniques*, 142(2):165–169, 1995.

[204] S. Yen, C. Laih, and A. Lenstra. Multi-exponentiation. *IEE Proceedings, Computers and Digital Techniques*, 141(6):325–326, 1994.

[205] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[206] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[207] Improved discrete logarithm systems with key-size reduction, 2000. Anonymous Submission.

Samenvatting

Cryptologie is letterlijk de leer der geheimen. Oorspronkelijk is het inderdaad zo dat cryptografen zich voornamelijk bezig hielden met het versleutelen van geheime informatie. Cryptanalisten probeerden die geheimen vervolgens weer te ontfutselen. Tegenwoordig worden cryptografische systemen ook voor hele andere doeleinden gebruikt, zoals het zetten van een digitale handtekening (een functioneel en soms ook wettelijk equivalent van de handgeschreven variant), maar ook het veilig uitvoeren van verkiezingen over het Internet.

Centraal in al deze moderne toepassingen zijn functies die makkelijk uit te rekenen zijn voor de gebruiker, maar die moeilijk te inverteren zijn voor een boosdoener. Het is bijvoorbeeld eenvoudig twee priemgetallen met elkaar te vermenigvuldigen, maar het is onbekend of het omgekeerde, factoriseren, ook snel kan. Een groot aantal cryptografische systemen is gebaseerd op groepen waarin het discrete logaritme probleem algemeen wordt verondersteld moeilijk te zijn. Er is onderzoek gedaan naar het versnellen van machtsverheffen in een aantal dergelijke groepen. Hierdoor winnen de cryptografische systemen eveneens aan snelheid.

Het versnellen van een machtsverheffing kan ruwweg op twee manieren. Enerzijds door sneller te vermenigvuldigen en anderzijds door minder te vermenigvuldigen.

In hoofdstuk 2 is gekeken naar de theorie en praktijk van optelketens. Deze kunnen gebruikt worden om het aantal groepsvermenigvuldigingen laag te houden. Een speciaal type optelketens dat gebruikt kan worden voor recurrente betrekkingen staat centraal in 3. Een algoritme van Montgomery voor tweede orde betrekkingen is aangepast voor derde orde betrekkingen.

In hoofdstuk 4 wordt gekeken naar ondergroepen van priemorde delende $p+1$ in $\mathbb{F}_{p^2}^*$ en van priemorde delende p^2-p+1 in $\mathbb{F}_{p^6}^*$. Voor beide gevallen worden versnellingen voor de groepsaritmetiek beschreven voor zekere congruentieklassen van p in een niet-gecomprimeerde representatie. Voor de ondergroep van $\mathbb{F}_{p^6}^*$ wordt ook een versnelling beschreven van XTR, een reeds bestaande gecomprimeerde variant.

Traditiegetrouw wordt machtsverheffen scalaire vermenigvuldiging genoemd voor de groep gebaseerd op een elliptische kromme. In hoofdstuk 5 wordt een alternatieve Montgomery-representatie gepresenteerd voor krommen over een lichaam van karakteristiek twee; traditionele projectieve methoden blijken echter sneller te zijn. Ter vergelijking worden ook krommen over een lichaam van priemorde beschreven.

Tot slot is in hoofdstuk 6 gekeken naar het versnellen van een zogeheten zwarte-doos-geheimenverdeelschema. Het specifieke schema dat onder handen wordt genomen is van Cramer en Fehr. Het blijkt dat ook hier feitelijk sprake is van machtsverheffingen. Door kleine aanpassingen aan het oorspronkelijke schema is het in dit geval mogelijk om de exponenten aanzienlijk kleiner te krijgen wat vanzelfsprekend tot versnellingen leidt.

Dankwoord

Tijdens mijn tien jaar in Eindhoven heb ik steeds het geluk gehad steeds van de juiste personen op de juiste momenten de juiste steun te krijgen. Dit gold ook voor de vier jaar promotie-onderzoek. Ik kan mij nog goed een faculteitsraadsvergadering herinneren waarin de toenmalige decaan voor het eerst gewag maakte van twee mogelijke aanvullingen voor de Crypto-groep. Hierbij noemde hij geen namen, maar merkte enkel op dat het ging om een bankier en een veelbelovende jonge Nederlandse onderzoeker, winnaar van een prestigieuze prijs.

De bankier was en is Arjen Lenstra, mijn eerste promotor. Hij kwam als deeltijdhoogleraar naar Eindhoven als expert op het gebied van algoritmen in de cryptografie[1], juist op een moment dat ik me meer op de efficiëntie van cryptosystemen begon te richten. Hoewel hij slechts zes weken per jaar in Eindhoven is (en daarbij Eindhoven soms als synoniem voor Europa lijkt te hanteren), was hij er altijd als ik hem nodig had. Een indrukwekkende hoeveelheid e-mails werd gebruikt voor de communicatie, maar ook de invloed van de rendez-vous tijdens conferenties mag niet onderschat worden. Hij zorgde ervoor dat ik tijdens belangrijke conferenties presentaties kon geven, hielp me bij de voorbereiding daarvan, toonde me de kunst van het schrijven van een artikel en op de momenten dat ik het even wat minder zag zitten sprak hij me moed in. Wat kun je nog meer van een promotor verwachten?

Uiteraard zijn er meer mensen aan wie ik dank verschuldigd ben, omdat zij direct hebben bijgedragen aan de totstandkoming van dit proefschrift.

Henk van Tilborg heeft mij aangesteld als AiO in Eindhoven. Dit begon met een projectaanvraag bij de STW, lang voordat ik wist dat ik AiO wilde worden. Hij heeft mij ook op vele manieren de vrijheid gegeven, bijvoorbeeld door bij de STW te pleiten voor een ruime interpretatie van de oorspronkelijke projectaanvraag. Ik ben Henk ook zeer dankbaar voor zijn bereidheid als tweede promotor op te treden.

Van september tot de kerst van 2002 ben ik te gast geweest als Marie Curie Fellow bij BRICS in Århus. In Denemarken werd ik begeleid door Ronald Cramer, ooit een veelbelovende jonge Nederlandse onderzoeker, winnaar van een prestigieuze prijs, maar inmiddels een internationaal gerespecteerde wetenschapper. Hoofdstuk 6 zou er niet geweest zijn zonder Ronald. De gesprekken in Århus maar ook latere telefoongesprekken hebben hopelijk hun positieve uitwerking op de rest van het proefschrift en mijn insteek voor de toekomst niet gemist.

[1] Iemand omschreef hem ooit als 'wereldkampioen factoriseren'.

Met Berry Schoenmakers heb ik in het verlengde van mijn afstuderen aanvankelijk veel samengewerkt op het gebied van groepshandtekeningen. Hoewel er van dit werk weinig is terug te vinden in dit proefschrift, beschouw ik het als een buitengewoon leerzame ervaring. Ook nadat ik een andere kant was opgegaan, bleef Berry geïnteresseerd en betrokken, wat bijvoorbeeld tot uiting kwam in Algoritme 3.17 en zijn bereidheid het gehele proefschrift door te lezen voor het naar de drukker ging.

Naast bovengenoemde personen wil ik de overige leden van de promotiecommissie bedanken: Jack van Lint, Eric Verheul, Bart Preneel, Arjeh Cohen, Jos Baeten en Peter van Emde Boas. Laatstgenoemde completeerde de kerncommissie en gaf mij na een voordracht in Utrecht enkele nuttige literatuurreferenties.

Peter Beelen wil ik bedanken voor de vruchtbare discussies over Lucas verzamelingen en ketens, elliptische krommen en ondergroepen van eindige lichamen, maar toch ook wel een beetje voor zijn heldhaftige optreden bij onze avonduurlijke samenwerking.

Dit brengt mij tot de mensen die een steun zijn geweest die niet altijd direct wetenschappelijk van aard was: de STW en de leden van de gebruikerscommissie voor de plezierige voortgangsvergaderingen; Jeroen Doumen als projectgenoot en voor behulpzaamheid voor velerlei dingen; Jan Draisma en Heleen Neggers voor tien mooie jaren in Eindhoven en de ceremoniële hulp bij de afsluiting daarvan; mijn kamergenootjes Julia, Shengli, Sacha en Giuseppe; de AiO's van de ronde tafel in Eindhoven; the colleagues and friends in Århus (jeg kan godt lide folk fra datalogi), with whom I'd still like to bike to Grenå; de buurvrouwen Anita en Henny voor de gezelligheid en de formulierenhulp; en zo zou ik nog even door kunnen gaan.

Nog even specifiek wil ik mijn huisgenoten en oud-huisgenoten van Woongroep Flater, Huishoudencentrum of gewoon Stratumsedijk 53 willen bedanken. Sinds het begin van mijn promotie heb ik er met veel plezier gewoond. In het bijzonder wil ik huisoudste Hajo bedanken voor het op de proppen komen met de fantastische luistervink.

Tot slot mijn ouders en mijn broertje.

Curriculum Vitae

Geboren in Tilburg op 17 februari 1975, maar grotendeels opgegroeid in Veghel, doorliep Martijn Stam het gymnasium (Bernrode) in Heeswijk-Dinther van 1987 tot 1993. Van 1993 tot 1999 studeerde hij Technische Wiskunde aan de Technische Universiteit Eindhoven. Deze studie werd cum laude afgesloten met het afstudeerproject *Toggling Schemes for Electronic Voting* dat was uitgevoerd bij BT Labs, Ipswich, Verenigd Koninkrijk, onder begeleiding van dr. Simon Phoenix, prof.dr.ir. H.C.A. van Tilborg en dr.ir. L.A.M. Schoenmakers. Vrijwel direct na zijn afstuderen ging Martijn aan de slag als AiO in de groep Coderingstheorie en Cryptologie.