

Public-Key Cryptography Theory and Practice

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erreichung des akademischen Grades
Doctor rerum naturalium (Dr. rer. nat.)

von

Dipl.-Inform. Bodo Möller

aus Hamburg

Referenten: Prof. Dr. Johannes Buchmann (TU Darmstadt)
Prof. Dr. Christof Paar (Ruhr-Universität Bochum)

Tag der Einreichung: 1. Juli 2003
Tag der mündlichen Prüfung: 16. September 2003

Darmstadt, 2003
Hochschulkennziffer: D 17

Viele haben auf die eine oder andere Weise dazu beigetragen, dass diese Dissertation so entstehen konnte, wie sie nun vorliegt. Der Versuch einer vollständigen Aufzählung müsste scheitern; hier seien zunächst die erwähnt, die nicht mit Namen genannt werden können, weil sie als anonyme Gutachter für Konferenzen tätig waren und dabei Anregungen zur Darstellung einiger der hier präsentierten Ergebnisse beigetragen haben. Außerdem zu nennen ist David Hopwood, der in einer früheren Fassung der Ausführungen zur beweisbaren Sicherheit des Mix-Verfahrens (hier in Abschnitt 4.2) eine Lücke aufgespürt hat. Prof. Johannes Buchmann hat es auf bemerkenswerte Weise verstanden, die Arbeitsbedingungen zu schaffen, in denen diese Dissertation gedeihen konnte, und hat wertvolle Anregungen geliefert. Auch alle anderen am Fachgebiet Theoretische Informatik hatten teil daran, eine angenehme und fruchtbare Arbeitsatmosphäre zu schaffen. Danke!

Darmstadt, im September 2003

B. M.

Abstract

Part I: Theory

Provable security is an important goal in the design of public-key cryptosystems. For most security properties, it is computational security that has to be considered: an attack scenario describes how adversaries interact with the cryptosystem, trying to attack it; the system can be called secure if adversaries with reasonably bounded computational means have negligible prospects of success. The lack of computational problems that are guaranteed to be hard in an appropriate sense means that there is little hope for absolute proofs of computational security. Instead, reduction-based security proofs have to be used: the computational security of a complex cryptographic scheme is related to the security of simpler underlying cryptographic primitives (under appropriate notions of security). The idea is to show that if the complex scheme is not secure, then this is because one of the primitives is not secure. Security can be described quantitatively as “concrete security”, measured depending on the power given to adversaries.

The DHAES construction (due to Abdalla, Bellare, and Rogaway) allows building a public-key encryption scheme from a key encapsulation mechanism (KEM), a one-time message authentication code (one-time MAC), and a pseudo-random bit string generator. A reduction-based security proof shows that DHAES achieves security against (adaptive) chosen-ciphertext attacks if these underlying primitives are secure. (Such chosen-ciphertext attacks are the most general attack scenario for public-key encryption.)

A specific application for public-key cryptography is considered, namely Chaum’s mix chain concept for untraceable electronic mail via cryptographic remailers: to obtain anonymity without requiring trust in a single authority, messages are recursively public-key encrypted to multiple intermediates (mixes), each of which forwards the message after removing one layer of encryption. To conceal as much information as possible when using variable (source routed) chains, all messages passed to mixes should be of the same length; thus, message length should not decrease when a mix transforms an input message into the corresponding output message directed at the next mix in the chain. Chaum described an implementation for such length-preserving mixes, but it is not secure against active attacks. This thesis presents a new construction for practical length-preserving mixes, which uses the cryptographic primitives described for DHAES. The conventional definition of security against chosen ciphertext attacks for public-key encryption schemes is not applicable to length-preserving mixes, so appropriate security definitions are introduced; it is shown that the mix construction achieves provable security.

Part II: Practice

Most instantiations of public-key cryptography involve computing powers (exponentiation) or computing power products (“multi-exponentiation”) in some commutative semigroup with neutral element. This thesis describes the sliding window technique for arbitrary commutative semigroups with neutral element and its signed-digit variant (“window NAF”) for groups where inversion is fast (e.g. point groups of elliptic curves and class groups of imaginary quadratic number fields), and then presents new techniques. Fractional windows, a generalization of the previously known window methods, can be useful for devices with limited storage. Interleaved exponentiation is a simple strategy for multi-exponentiation; the comparison with previous simultaneous exponentiation methods shows that it often provides better

efficiency. Window NAF splitting is a method for fast exponentiation with precomputation for a fixed base in groups where inversion is fast.

For the case of elliptic curves, side-channel attacks are discussed, i.e. attacks where adversaries use power consumption measurements or similar observations to derive information on secret values. Two methods are shown that are designed to limit potential information leakage available to adversaries: a 2^w -ary left-to-right method employing special representations of scalars, and a 2^w -ary right-to-left method with a special randomized initialization stage.

Zusammenfassung

Teil I: Theorie

Beweisbare Sicherheit ist ein wichtiges Ziel beim Entwurf von Public-Key-Kryptosystemen. Für die meisten Sicherheitseigenschaften muss hierbei praktische Sicherheit gegen Angreifer mit beschränkten Mitteln betrachtet werden: Ein Angriffsszenario beschreibt, wie Gegner mit dem Kryptosystem interagieren dürfen, um es anzugreifen; das System kann dann als sicher bezeichnet werden, wenn Gegner mit einem praktikablen Rechenaufwand nur vernachlässigbare Erfolgsaussichten erzielen können. Da es keine Berechnungsprobleme gibt, die bewiesenermaßen in einem geeigneten Sinne schwer sind, gibt es nur wenig Hoffnung für absolute Sicherheitsbeweise im Sinne solcher praktischer Sicherheit. Statt dessen muss auf reduktionsbasierte Sicherheitsbeweise zurückgegriffen werden: Die praktische Sicherheit eines komplexen kryptographischen Systems wird zu der Sicherheit einfacherer zugrundeliegender kryptographischer Primitive (mit jeweils passenden Sicherheitsbegriffen) in Beziehung gestellt. Der Grundgedanke ist, zu zeigen, dass das komplexe System nur dann unsicher sein kann, wenn eine der Primitiven unsicher ist. Die Sicherheit kann als „konkrete Sicherheit“ quantitativ beschrieben werden in Abhängigkeit von den Mitteln, die Gegnern zur Verfügung stehen.

Mit dem DHAES-Schema von Abdalla, Bellare und Rogaway kann man aus einem Schlüsselaustauschverfahren (key encapsulation mechanism, KEM), einem Einmal-MAC (message authentication code) und einem Pseudozufallsbitstringgenerator ein Public-Key-Verschlüsselungsverfahren konstruieren. Ein reduktionsbasierter Sicherheitsbeweis zeigt, dass DHAES Sicherheit gegen (adaptive) „Chosen-ciphertext“-Angriffe erzielt, falls die zugrundeliegenden Primitive sicher sind. (Solche Chosen-ciphertext-Angriffe sind das allgemeinste Angriffsszenario für Public-Key-Verschlüsselung.)

Eine spezifische Anwendung für Public-Key-Kryptographie wird betrachtet, Chaums Mix-Ketten-Konzept für nicht verfolgbare E-Mail über kryptographische Remailer: Damit Absender Anonymität erreichen können, ohne einer einzelnen Stelle vertrauen zu müssen, werden Nachrichten rekursiv für mehrere Zwischenstellen verschlüsselt (die „Mixe“), die jeweils die Nachricht weitersenden, nachdem sie eine Ebene der Verschlüsselung entfernt haben. Um bei der Benutzung von variablen Mix-Ketten sowenig Information wie möglich zu verraten, sollten alle an Mixe gesandte Nachrichten die gleiche Länge haben; also sollte die Nachrichtenlänge sich nicht verringern, wenn ein Mix eine ankommende Nachricht umwandelt in die Nachricht, die er an den nächsten Mix in der Kette weiterzuleiten hat. Die von Chaum beschriebene Konstruktion für solche längenerhaltene Mixe ist nicht sicher gegen aktive Angreifer. Diese Dissertation stellt eine neue Konstruktion für praktische längenerhaltene Mixe vor, welche auf den für DHAES angegebenen kryptographischen Primitiven aufbaut. Die übliche Definition für Sicherheit gegen „Chosen-ciphertext“-Angriffe bei Public-Key-Verschlüsselung kann für längenerhaltene Mixe nicht angewendet werden; deshalb werden angemessene Sicherheitsdefinitionen vorgestellt. Es wird gezeigt, dass die Mix-Konstruktion beweisbare Sicherheit bietet.

Teil II: Praxis

Bei der Durchführung von Public-Key-Kryptographie sind üblicherweise Potenzen zu berechnen (Exponentiation) oder Produkte von Potenzen („Multi-Exponentiation“). Diese Dissertation beschreibt die „Sliding-window“-Technik für beliebige kommutative Halbgruppen mit neutralem Element und ihre Variante mit vorzeichenbehafteter Codierung („window NAF“),

die für Gruppen mit schneller Invertierung verwendet werden kann (z. B. Punktegruppen elliptischer Kurven oder Klassengruppen imaginärquadratischer Zahlkörper). Anschließend werden neue Techniken präsentiert: Die Verallgemeinerung der bekannten Fenstermethoden zu „fractional windows“ kann für Rechner mit beschränktem Speicherplatz von Vorteil sein. „Interleaved exponentiation“ ist eine einfache Strategie für Multi-Exponentiation; der Vergleich mit den bekannten Multi-Exponentiations-Methoden („simultaneous exponentiation“) zeigt, dass der neue Ansatz oft bessere Effizienz liefert. „Window NAF splitting“ ist eine Methode zur schnellen Exponentiation mit Vorberechnung für eine feststehende Basis in Gruppen mit schneller Invertierung.

Für den Fall von elliptischen Kurven werden schließlich Seitenkanalangriffe diskutiert, d. h. Angriffe, bei denen Angreifer Messungen des Stromverbrauchs oder ähnliche Beobachtungen benutzen, um Information über geheime Werte zu erhalten. Zwei Verfahren werden vorgestellt, die gezielt dem Bekanntwerden von geheimer Information über Seitenkanäle entgegenwirken: eine 2^w -äre Links-nach-rechts-Methode, die auf einer speziellen Darstellung der Skalare beruht, und eine 2^w -äre Rechts-nach-links-Methode mit einem besonderen randomisierten Initialisierungsschritt.

Erklärung¹

Hiermit erkläre ich, dass ich die vorliegende Arbeit – abgesehen von den in ihr ausdrücklich genannten Hilfen – selbständig verfasst habe.

Wissenschaftlicher Werdegang des Verfassers in Kurzfassung²

Okt. 1993 – Okt. 1999	Studium der Informatik mit Ergänzungsfach Mathematik an der Universität Hamburg
20. 10. 1999	Diplomprüfung (Dipl.-Inform.)
Nov. 1999 – Sept. 2003	wissenschaftlicher Mitarbeiter am Fachgebiet Theoretische Informatik (Prof. J. Buchmann), Fachbereich Informatik, Technische Universität Darmstadt

¹gemäß §9 Abs. 1 der Promotionsordnung der TU Darmstadt

²gemäß §20 Abs. 3 der Promotionsordnung der TU Darmstadt

Contents

1	Introduction	13
I	Theory	15
2	Public-Key Cryptography and Provable Security	17
2.1	Information-Theoretic Security	17
2.2	Public-Key Cryptography	18
2.3	Computational Security	20
2.3.1	Example: IND-CPA Security	21
2.3.2	Reduction-Based Security Proofs	22
2.3.3	Asymptotic Security Proofs	23
2.3.4	The Random Oracle Model	24
2.3.5	Conclusions	25
3	Hybrid Public-Key Encryption: The DHAES Construction	27
3.1	Adaptive Chosen-Ciphertext Attacks	28
3.2	Primitives for the DHAES Construction	30
3.3	The DHAES Public-Key Encryption Scheme	32
3.4	The Security of the DHAES Construction	33
3.4.1	Security of the Key Encapsulation Mechanism	33
3.4.2	Security of the One-Time Message Authentication Code	34
3.4.3	Security of the Pseudo-Random Bit String Generator	35
3.4.4	Security Result for the DHAES Construction	35
4	A Public-Key Cryptosystem for Length-Preserving Chaumian Mixes	39
4.1	The Mix Encryption Scheme	41
4.1.1	Encryption	42
4.1.2	Decryption	43
4.2	Provable Security	45
4.2.1	Unlinkability of the Mix Encryption Scheme	46
4.2.2	CCA Security of the Mix Encryption Scheme	47
4.2.3	Security Results	49

II	Practice	53
5	Exponentiation and Multi-Exponentiation in Public-Key Cryptography	55
6	Efficient Exponentiation	59
6.1	A Framework for Exponentiation	59
6.1.1	Left-to-Right Methods	60
6.1.2	Right-to-Left Methods	61
6.2	Sliding Window Exponentiation and Window NAF Exponentiation	62
6.2.1	Modified Window NAFs	63
6.3	Fractional Windows	64
6.3.1	Signed Fractional Windows	64
6.3.2	Unsigned Fractional Windows	67
6.4	Compact Encodings	69
7	Efficient Multi-Exponentiation	71
7.1	Simultaneous Exponentiation	72
7.1.1	Simultaneous 2^w -Ary Exponentiation Method	73
7.1.2	Simultaneous Sliding Window Exponentiation Method	74
7.2	Interleaved Exponentiation	75
7.2.1	Basic Interleaved Exponentiation Method	76
7.2.2	Window-NAF Based Interleaved Exponentiation Method	77
7.3	Comparison of Simultaneous and Interleaved Exponentiation Methods	78
7.3.1	Comparison between the Simultaneous 2^w -Ary Method and the Basic Interleaved Method	79
7.3.2	Comparison between the Simultaneous Sliding Window Method and the Window-NAF Based Interleaved Method	79
7.3.3	Comparison between the Dimitrov-Jullien-Miller Multi-Exponentiation Method and Interleaved Exponentiation	80
7.3.4	Multi-Exponentiation with Fractional Windows	81
7.3.5	Examples	81
7.3.6	Conclusions	82
7.4	Exponentiation and Multi-Exponentiation with Precomputation for Fixed Bases	82
7.4.1	Exponent Splitting	84
7.4.2	Lim-Lee Precomputation	84
7.4.3	Window NAF Splitting	85
8	Elliptic Curve Point Multiplication with Resistance against Side-Channel Attacks	87
8.1	Previous Side-Channel Attack Countermeasures	88
8.2	Security against Side-Channel Attacks	90
8.2.1	Elliptic Curve Point Operations	90
8.2.2	Field Operations	91
8.3	2^w -Ary Left-to-Right Method	92
8.3.1	Recoding Algorithms	92
8.3.2	Point Multiplication Algorithm	94
8.3.3	Uniformity of the Point Multiplication Algorithm	95

8.4	2^w -Ary Right-to-Left Method	97
8.4.1	Initialization Stage	98
8.4.2	Right-to-Left Stage	99
8.4.3	Result Stage	100
8.4.4	Variants	100
8.5	Efficiency Comparison	102
8.6	Scalar Randomization	104
	Bibliography	107
	Index	115

Chapter 1

Introduction

This dissertation examines multiple aspects of public-key cryptography.

Part I looks at the *theory of provably secure public-key cryptography*, focusing on encryption. First, chapter 2 gives an introduction to this topic area and discusses the limitations of provable security. Then, chapter 3 describes the DHAES construction for public-key encryption and gives a reduction-based security proof for it. Chapter 4 concludes the part by presenting a construction for a specific variant of encryption, namely public-key encryption with length-expanding decryption for use with Chaum's mix concept for untraceable electronic mail. Again, security is proved in a reduction-based way.

Part II concerns itself with an important topic for the *practice of public-key cryptography*, namely the implementation of exponentiation and multi-exponentiation. Chapter 5 shows some typical application scenarios. Chapter 6 presents techniques for efficient exponentiation. Chapter 7 presents techniques for efficient multi-exponentiation and for efficient exponentiation with precomputation for fixed bases. Finally, chapter 8 includes the issue of side-channel attacks into the consideration and describes techniques for elliptic curve point multiplication (a specific case of exponentiation) that are designed to resist such attacks.

Of course, no attempt is made to give a complete survey on the theory or the practice of public-key cryptography. This would amount to compiling a vast collection of material, and such a collection would be outdated quickly as much research is done in both areas. Instead, the goal here is to provide an introduction to provable security (chapters 2 and 3) and to give sufficient context for the areas of my own research results concerning provably secure public-key cryptography (chapter 4; cf. my conference publication [63]), efficient implementation of exponentiation and multi-exponentiation (chapters 6 and 7; cf. [58, 62]), and point multiplication with resistance against side-channel attacks (chapter 8; cf. [59, 60, 61]).

Part I
Theory

Chapter 2

Public-Key Cryptography and Provable Security

The term *cryptography* originally refers to hidden writing. Today it has a much broader meaning and covers various related concepts besides encryption, such as authentication. In the following chapters on the theory of provably secure public-key cryptography, we will look at certain constructions specifically for encryption. The present chapter provides some background for this, focusing on the aspect of provability; much of its content applies to other types of public-key cryptography as well.

Notational Conventions

$x \in_{\mathbb{R}} S$ means that x is uniformly randomly chosen from set S (where the distribution is understood to be independent of any other given probability distributions that do not explicitly depend on x).

For a bit string s (an element of $\{0, 1\}^*$), $|s|$ denotes its length. The concatenation of strings s and t is denoted $s \parallel t$. For $|s| \geq w$, $\text{prefix}_w(s)$ is the string consisting of the leftmost w bits of s . Thus, $\text{prefix}_{|s|}(s \parallel t) = s \in \{0, 1\}^{|s|}$.

Algorithms are usually probabilistic (randomized).

2.1 Information-Theoretic Security

Here we look at *symmetric cryptography* before proceeding to public-key cryptography, which is also known as asymmetric cryptography, in the next section. Assume a simple scenario where just two parties are involved, A and B. Then, in symmetric cryptography, A and B must both have knowledge of a piece of data that should be kept secret from others, the *key*. A simple symmetric cryptosystem is the following ([88], [79]).

Protocol 2.1. *Let G be a finite group. Vernam encryption in group G works as follows. Parties A and B agree on a key $K \in_{\mathbb{R}} G$ (a uniformly random element of G). To encrypt some message represented by an element $m \in G$ (the plaintext), A computes the group element $c = m + K$ (the ciphertext). B, when having received c , can recover the plaintext m as $c - K$.*

The Vernam encryption scheme is also known as the *one-time pad* (OTP) because each key should be used only for encrypting a single message. Often the group is $G = \{0, 1\}^k$ for some $k \in \mathbb{N}$ with XOR as group operation, but the scheme works in any finite group.

The only assumption we make on the distribution of m is that it be stochastically independent of the distribution of K . Then for any elements $\tilde{m}, \tilde{c} \in G$, we have

$$\Pr[m = \tilde{m} \mid c = \tilde{c}] = \Pr[m = \tilde{m} \mid K = \tilde{c} - m] = \Pr[m = \tilde{m}]$$

because K has independent uniform distribution. This means that, if K is kept secret, an adversary who observes some ciphertext c cannot learn any information whatsoever on m from it: the Vernam encryption scheme is *information-theoretically secure*.

A caveat is that this result only holds if keys K are perfectly random. In implementations, there will usually be some bias. However miniscule it may be, this means that ciphertexts will reveal at least a tiny amount of information on the corresponding plaintexts. Typically the bias will be small enough so that it does not really matter at all, and no-one will know the details about the bias that would be necessary to exploit it; we should just keep in mind that perfect information-theoretic security remains a mathematical utopia an epsilon away.

2.2 Public-Key Cryptography

In *asymmetric* or *public-key cryptography* ([32], [33]), there is no longer a single key shared by the parties that are involved. Instead, the key generation algorithm produces multiple keys that are associated with another: usually, there is a *secret key* that should be known only to a single entity, and a *public key* that may be made known to everyone.

One form of public-key cryptography is *public-key encryption*, where someone's public key can be used to encrypt *plaintexts* such that the resulting *ciphertexts* can be decrypted using the corresponding secret key. Another important form of public-key cryptography are *digital signature* schemes: the entity holding the secret key can use it to “sign” digital documents by computing a value, a “signature”, such that anyone who knows the public key can verify whether the signature is valid for a given message. Also various other scenarios for cryptographic protocols are known that belong into the area of public-key cryptography. We will focus on public-key encryption and specific closely related notions and not go into the details of digital signatures or other protocols.

A concept related to public-key encryption is the notion of a *key encapsulation mechanism* (KEM; see [80]). As in public-key encryption, someone's public key is used to create a *ciphertext* from which a *plaintext* can be recovered using the associated secret key. Unlike for public-key encryption, the sender does not get to specify an arbitrary plaintext when creating the ciphertext; instead, the randomized “encryption” algorithm outputs both the plaintext and a corresponding ciphertext. This approach is appropriate for use in cryptographic protocols where the plaintext is intended as a key for symmetric cryptography – hence the term key encapsulation.

Note that both symmetric and public-key encryption schemes can be randomized: encrypting a given plaintext for a constant key does not necessarily yield a unique result. However, encryption followed by decryption has a deterministic result, namely the original plaintext; this is in contrast with KEMs. (The variant of public-key encryption that we will consider in chapter 4 is an unusual exception where decryption yields the original plaintext padded with some additional data, which is determined while encrypting. In that public-key cryptosystem with length-expanding decryption, the result of encryption followed by decryption is partially deterministic and partially pseudo-random.)

A formalization of the notion of a public-key encryption scheme will appear in section 3.1, where we will also look at different attack scenarios and see how the security of public-key encryption in a strong sense (i.e. when exposed to an adversary in a far-going attack scenario) can be expressed quantitatively. A formalization of the notion of a KEM will appear in section 3.2 (see section 3.4.1 for security notions for a KEM). In the present chapter, we will preview some of these subjects, but without going all the way yet (we treat these concepts somewhat informally, and we confine to rather limited attack scenarios for ease of exposition).

As basic examples, we look at three variants of the *Diffie-Hellman* (DH) key encapsulation mechanism ([33], [1], [2], [80]).

Protocol 2.2. *Let G be a finite group (whose order $\#(G)$ is known) and $g \in G$. A key pair consisting of a secret key SK and a public key PK is generated as follows: pick*

$$\text{SK} \in_{\mathbb{R}} \{0, 1, \dots, \#(G) - 1\}$$

and let

$$\text{PK} = g^{\text{SK}}.$$

A sender who knows the public key PK generates a ciphertext $\mathfrak{R} \in G$ and the corresponding plaintext $K \in G$ as follows: pick

$$x \in_{\mathbb{R}} \{0, 1, \dots, \#(G) - 1\}$$

and let

$$\mathfrak{R} = g^x$$

and

$$K = \text{PK}^x.$$

Now if the sender transmits \mathfrak{R} to the owner of the secret key, the latter can recover the plaintext K by computing

$$\mathfrak{R}^{\text{SK}}.$$

(Observe that $\mathfrak{R}^{\text{SK}} = g^{x \cdot \text{SK}} = \text{PK}^x = K$).

Protocol 2.3. *Again, let G be a finite group (of known order) and $g \in G$.*

Let k be a positive integer, and let $h: G \rightarrow \{0, 1\}^k$ be some hash function. (By this, we mean that h should intuitively behave like a randomly chosen map from the first set to the second: we want to be able assume that for any $y \in G$, the k -bit string $h(y)$ is random. However, note that this is just a way to think about what happens and is certainly not how h can be defined in practice.)

Now the Diffie-Hellman key encapsulation mechanism can be performed similar to protocol 2.2 with the help of the hash function. Key generation is as above:

$$\text{SK} \in_{\mathbb{R}} \{0, 1, \dots, \#(G) - 1\}, \quad \text{PK} = g^{\text{SK}}.$$

The sender now proceeds as follows to generate a ciphertext $\mathfrak{R} \in G$ and a corresponding plaintext $K \in \{0, 1\}^k$: pick

$$x \in_{\mathbb{R}} \{0, 1, \dots, \#(G) - 1\},$$

then let

$$\mathfrak{R} = \text{PK}^x$$

and

$$K = h(g^x).$$

The owner of the secret key can recover K from \mathfrak{K} by computing

$$h(\mathfrak{K}^{\text{SK}}).$$

Protocol 2.4. A further variant of the Diffie-Hellman scheme works mostly as the previous one, but applies a hash function differently. We describe just the differences to protocol 2.3.

This time, we have a hash function $h: G \times G \rightarrow \{0, 1\}^k$. The sender computes $K \in \{0, 1\}^k$ as

$$K = h(\mathfrak{K}, g^x),$$

and the owner of the secret key computes it as

$$K = h(\mathfrak{K}, \mathfrak{K}^{\text{SK}}).$$

In these three protocols, we have assumed that in addition to the specific public keys, senders also have certain additional information: they know which group G to compute in, they know which element $g \in G$ to use, and they know the order $\#(G)$ of the group. Such information is collectively referred to as *domain parameters*. It is possible to consider generating domain parameters part of key generation, but it is also possible to use fixed domain parameters for many key pairs.

Observe that key encapsulation mechanisms can easily be combined with symmetric encryption schemes to create public-key encryption schemes. For example, any of the Diffie-Hellman variants described above can be followed by Vernam encryption (protocol 2.1) in an appropriate group using the DH result K as a key. In the case of the DH variant without a hash function (protocol 2.2), this combination is called *ElGamal encryption* [35].

Note that it is impossible to achieve full information-theoretic security in a meaningful sense for public-key cryptography: the public key necessarily must contain at least partial information on the secret key, or the two keys (secret and public) would not be associated with each other. For example, in the Diffie-Hellman schemes that we presented above, key PK determines SK modulo the order of g in G (secret keys that are congruent modulo $\text{ord}(g)$ are equivalent if the protocol is followed by both parties). Instead of information-theoretic security, in section 2.3 we will consider *computational security*, i.e. security against adversaries with bounded computational means.

(Some public-key protocols exist that are designed to achieve at least a certain notion of security in more than a computational sense. *Fail-stop signature schemes* [72] are signature schemes where a security failure of the system can be detected with very high probability: many different possible secret keys match each given public key, and if an adversary breaks the system computationally and produces a seemingly valid signature, then this signature will likely be based on a secret key differing from the legitimate one, and differ from the signature on the same message that would have been produced by the legitimate signer with his original secret key. In such cases, the key owner can demonstrate that there has been a security failure and that thus the fake signature should not be considered valid.)

2.3 Computational Security

As information-theoretic security is out of the question for public-key cryptography, we have to examine *computational security*. Adversaries are modelled as probabilistic algorithms that

somehow interact with the cryptographic scheme that they are attacking (details of this interaction depend on the specific notion of security). Then we can try to analyze the probabilities with which an adversary can achieve certain attack goals under given complexity bounds. These bounds might be explicit (e.g. a maximum number of algorithm steps) or implicit. Based on such probabilities, we will define the *advantage* of the adversary in the respective attack scenario as a real number in the interval $[0, 1]$ where value 0 (no advantage) means that the cryptographic scheme is secure under the notion of security under consideration against the respective adversary, and where value 1 means that the cryptographic scheme is totally insecure under the notion of security considered against the respective adversary (with its complexity bounds).

(Beware that other definitions for the advantage can be found in the literature: sometimes, the advantage ranges from $-1/2$ to $1/2$, where value 0 means no advantage as above and where both $-1/2$ and $1/2$ indicate total insecurity. Such a measure Adv' can be transformed into a measure Adv following our convention by taking $\text{Adv} = 2 \cdot |\text{Adv}'|$.)

What we are really interested in is the inherent security of the cryptographic scheme under a certain notion of security, not just the security against some specific adversary. Something that will usually have to be specified is an upper time bound for the adversaries that are to be considered. We can also include additional parameters besides the time bound: for example, attack models often provide the adversary with “encryption oracles” and “decryption oracles” or the like, which will process requests sent by the adversary; in such models, we can have parameters for the maximum number of queries to specific oracles, and the maximum size of all queries to them. Then the advantage of arbitrary adversaries can be defined as a function of all such parameters: it is the infimum of advantages of all adversaries that observe all the specified bounds. This is known as the *concrete security* approach ([7], [8]): security is described quantitatively, measured depending on the power given to adversaries.

2.3.1 Example: IND-CPA Security

As a first example, we show a basic notion of security for public-key encryption, namely *indistinguishability under chosen-plaintext attack* (IND-CPA), which goes back to [37]. (For stronger security notions, see the next chapter.) Assume that some public-key encryption scheme PKE is given. In the IND-CPA scenario, adversaries engage in the following attack game:

1. The adversary queries a *key generation oracle*, which computes a key pair (PK, SK) using the key generation algorithm specified for PKE. The public key PK is made available to the adversary, and SK is kept secret.
2. Now the adversary chooses two binary strings m_0 and m_1 of the same length and sends them to a so-called *encryption oracle*, which secretly picks $b \in_{\text{R}} \{0, 1\}$, applies the encryption algorithm of PKE to m_b using the key PK , and responds with the resulting string, which we call the *challenge ciphertext*.
3. The adversary outputs a bit $\tilde{b} \in \{0, 1\}$, which is supposed to be a guess for the value of bit b .

Now for a specific adversary A (an interactive probabilistic algorithm with bounded running time), we define its *CPA advantage* against the public-key encryption scheme as

$$\text{Adv}_{\text{CPA}, \text{PKE}, A} = \left| \Pr[\tilde{b} = 1 \mid b = 1] - \Pr[\tilde{b} = 1 \mid b = 0] \right|.$$

Note that because

$$\begin{aligned} \Pr[\tilde{b} = 1 \mid b = 1] - \Pr[\tilde{b} = 1 \mid b = 0] &= \Pr[\tilde{b} = 1 \mid b = 1] + \Pr[\tilde{b} = 0 \mid b = 0] - 1 \\ &= 2\Pr[\tilde{b} = b] - 1, \end{aligned}$$

we can also write this as

$$\text{AdvCPA}_{\text{PKE},A} = 2 \cdot \left| \Pr[\tilde{b} = b] - \frac{1}{2} \right|.$$

(Observe that if there is an adversary that tends to guess wrong, this is just as bad as if it tends to guess right: by always inverting the adversary’s output bit \tilde{b} , one case can be transformed into the other.)

The term “chosen-plaintext attack” refers to the fact that the adversary can encrypt arbitrary plaintexts of its own choice because it knows the public key PK. Observe that a public-key encryption scheme cannot be secure under this definition unless it is randomized (presuming that it is sound in the sense that decrypting the encryption of a given plaintext will actually recover the original plaintext): for a deterministic scheme, the adversary could simply encrypt m_0 or m_1 ; one of the ciphertexts will equal the challenge ciphertext, and if $m_0 \neq m_1$, this immediately reveals b .

2.3.2 Reduction-Based Security Proofs

Proving the computational security of a cryptographic scheme would amount to showing that something that is easy to do with knowledge of all cryptographic keys involved is a hard computational problem for an adversary who does not know the relevant secret keys. However, except in the very narrow range where we actually have information-theoretic security, this touches the limits of complexity theory, and there is little hope for actually provable computational security. Instead, one can resort to *reduction-based security proofs*, in which it is shown that a cryptographic scheme complies to the security notion in question provided that a certain underlying computational problem is hard, or that a complex cryptographic scheme is secure under an appropriate security definition provided that the simpler cryptographic schemes (or *primitives*) from which it is built are secure under their respective security notions. The point of such proofs by reduction is that expressions of security for cryptographic schemes can be related to computational problems that are simple to describe and more directly accessible for complexity analyses, and the intractability of which may be supported by experience, i.e. by the failure to find efficient algorithms after intensive research.

Observe that saying that a computational problem “is hard” is quite ambiguous: it might merely mean that the problem has *some* difficult instances (cf. the expression “NP-hard” in complexity theory); or it might mean that easy instances are indeed rare. In the context of cryptography, we need problems that are hard in the stricter sense. To avoid the ambiguity, we will usually speak of “secure cryptographic primitives” rather than “hard computational problems” (any “hard problem” in the appropriate sense can be considered a cryptographic primitive).

Saying that some cryptographic scheme “is secure” is clearly simplistic. The concrete security approach can be used to make quantitative statements about security. Reduction-based security proofs work as follows: if the advantage that an adversary may be able to obtain against a cryptographic scheme can be bounded by a function of the advantages of

related adversaries against the underlying primitives (where these adversaries are derived from the adversary against the full scheme), then one can obtain security statements that essentially state that the full scheme must be secure unless one of the underlying primitives is vulnerable. (Some care must be taken as this intuitive interpretation may sometimes be too simplistic, e.g. if a derived adversary incurs a significant slowdown compared with the original adversary that it is based on.)

Security assumptions made for primitives often have basically the following form: “No adversary running in at most n steps can achieve an advantage better than ϵ .” Observe that while such statements may appear to describe an inherent property of the primitive in question, in fact they implicitly make assumptions on the adversaries as well. The reason is that the machine model that adversaries are supposed to run in is left unspecified. (For example, one “step of computation” in some machine formalization might require a large number of steps in another one.) The formal assumption describes restrictions on what adversaries can achieve, but nothing specific about the primitive taken for itself.

In principle we could fill this void and define a precise machine model for adversaries. Then, with respect to this particular model, the security assumption would be either true or false. For some possible models, it will be true, for others it will be false. (We could consider a machine model in which the problem at hand can be solved immediately because there is some specific built-in operation that achieves this.) Of course we usually hope that the models in which the security assumption is false are contrived, and that for realistic ones it is true.

2.3.3 Asymptotic Security Proofs

Often cryptographic schemes are thought to be parameterized by a *security parameter*, a non-negative integer k , which is possibly bounded from below, i.e. $k \geq k_0$ for some k_0 . Key generation, encryption and any other legitimate use of the cryptographic scheme is expected to be possible in time polynomial in k (assuming appropriate length bounds for inputs such as plaintexts). In such parameterized settings, we will usually be concerned with sets of keys and of messages that have unbounded cardinality. It is understood that to make it possible to describe cryptographic schemes through algorithms, elements of such sets will have to be represented as words over some finite alphabet.

For example, for Diffie-Hellman (protocols 2.2, 2.3 and 2.4) using the multiplicative group of some prime field, i.e. $G = (\mathbb{Z}/p\mathbb{Z})^*$, k could be the length of prime p in bits (here we have to insist that $k \geq 2$). Elements of G can be represented as bit strings of length k that contain the minimum positive integer in the respective residue class modulo p , expressed in binary.

The use of security parameters provides an approach to provable computational security for cryptographic schemes. A function $f: \mathbb{N} \rightarrow \mathbb{R}$ is called *negligible* if $f(n) = O(1/P(n))$ for any polynomial P , i.e. if for any $m \in \mathbb{N}$ there is an n_m such that $|f(n)| < 1/n^m$ for $n \geq n_m$. We want to call a cryptographic scheme secure according to some specified security notion if breaking it is hard in the sense that any adversary with run-time bounded polynomially in security parameter k has advantage negligible in k . Observe that security statements of this form are *asymptotic*.

As has already been noted in section 2.3.2, we cannot really hope for actual provable computational security for public-key cryptography: remember that the famous question whether the class of languages accepted by deterministic polynomial-time machines coincides with the class of languages accepted by non-deterministic polynomial-time machines ($\mathbf{P} =$

NP) remains unsettled, and observe that because **NP** algorithms can internally guess secret keys, efficient adversaries could be constructed if $\mathbf{P} = \mathbf{NP}$. So we still have to confine to reduction-based security proofs that rely on assumptions about the security of underlying primitives (which, in turn, are captured in the presumed negligibility of the advantage of any polynomially bounded adversary targeting the respective primitive).

One superficial advantage of asymptotic security results showing a negligible advantage compared with security results describing concrete security as discussed in the previous section is that here we can appeal to well-known equivalences between different machine models: asymptotic security statements do not depend on specific machine models. However, a canonical class of equivalent models exists *only* in asymptotics, and this in itself is a basic drawback. Security statements that are asymptotic in the security parameter imply that, given any adversary, its advantage can be kept arbitrarily close to zero if only the security parameter is chosen sufficiently large (provided that the underlying primitives are indeed secure). However, when a cryptographic scheme is actually employed, one has to fix some specific value of the security parameter, so this parameter clearly cannot change depending on which adversary one considers. With the security parameter fixed, the problem degenerates to one with a finite number of cases (since generating an instance of the problem is assumed to have polynomial complexity). This means that the problem can now be solved in constant time. An adversary against a public-key cryptosystem could even use a fixed table that maps each public key to a corresponding secret key to solve each instance of the problem instantly.

While thus an asymptotic statement obtained through a reduction-based security proof does not really say anything about practical instantiations of the cryptographic scheme, it can still be considered a valuable *heuristic* result: essentially, one can hope that the concrete choice of the security parameter will be large enough such that the asymptotic result has a close correspondence to reality.

2.3.4 The Random Oracle Model

One approach to heuristic security is the *random oracle model* [9]. Remember the remark (for protocol 2.3 above) that it is convenient to assume that hash functions behave like randomly chosen functions. The random oracle model formalizes this idealization by replacing concrete hash functions by randomly chosen functions. The adversary's advantage in the attack game is measured under this random distribution. The adversary does not obtain a complete description of the hash function. Instead, values of the hash function can only be obtained individually by way of oracle queries: a *random oracle* is substituted for the hash function.

The intuition behind the random oracle model is that security proofs in this idealization indicate that a cryptographic scheme is secure against *generic attacks* that do not exploit specific properties of some particular hash function. However, clearly no specific choice of a hash function can behave like a random oracle – the random oracle model does not correspond to any explicit assumption on security properties of hash functions.

It is easy to see that it is already problematic to consider a hash function as a single fixed function. A *collision* of a hash function

$$h: \text{dom}(h) \rightarrow \{0, 1\}^k$$

where $\text{dom}(h) \subseteq \{0, 1\}^*$ is a pair (m_0, m_1) of strings such that $m_0 \neq m_1$ but

$$h(m_0) = h(m_1).$$

Unless $\#(\text{dom}(h)) \leq 2^k$, it is clear that h will have a collision. If h behaves somehow “randomly”, one might expect that finding a collision would be difficult (if k is chosen sufficiently large). However, if h is fixed, there are conceivably simple algorithms for “finding” a collision: such algorithms do not have to do any actual work, they can simply output a collision of h that is built into the algorithm. Hence, hash functions used in practice should be modeled as keyed function families $(h_K)_{K \in \mathcal{K}}$ where a specific hash function h_K , $K \in_{\mathbb{R}} \mathcal{K}$, is determined at system set-up time. Then the advantage of adversaries can be considered under this random distribution. This approach makes it possible to formalize the security of hash functions including the intractability of finding collisions, but does not provide justification for the random oracle model.

Particular doubt has been cast on the methodology of employing the random oracle model for security proofs by showing that an essential discrepancy remaining between the keyed function family model for practical hash functions and the random oracle model allows constructing cryptographic schemes that can be proven secure in the random oracle model, but are insecure for any instantiation of the hash function [20, 21]. Descriptions of actual hash functions will be “small” (of polynomial size, considered asymptotically), while the complete specification of a randomly chosen function as in the random oracle model would be “large” (require exponential size). The example constructions from [20, 21] use a diagonalization technique to exploit this discrepancy.

These negative results have been presented in asymptotic form in [20, 21]. As we have seen in section 2.3.3, asymptotic security statements cannot say anything on practical instantiations of cryptographic schemes anyway, so the published result is not directly meaningful. However, the asymptotic form is not essential, and similar results could be obtained for particular machine models in non-asymptotic settings [19].

2.3.5 Conclusions

In sections 2.3.3 and 2.3.4, we have considered asymptotic security and the random oracle model. No specific choice of a security parameter can be “asymptotic”, and no specific choice of a hash function can be a “random oracle”; so security results should avoid these concepts if we want them to have a direct practical interpretation. In the following chapters, provable security will be handled according to a reduction-based concrete-security approach: cryptographic schemes are built from simpler cryptographic primitives; the security of the compound scheme can be related quantitatively to the security of the underlying primitives, where security is expressed through the advantage that can be achieved by an adversary. The security proofs will not refer explicitly to security parameters or to asymptotic security, and they will avoid random oracles. However, it is understood that the use of such concepts might be appropriate to examine in more detail the security assumptions that we make on cryptographic primitives. We will simply assume that appropriate primitives are available, so this issue will remain out of sight. Since we do not have justification for the security assumptions on primitives, in any case we end up relying on heuristical considerations, and we cannot get far if we are not willing to do so.

Chapter 3

Hybrid Public-Key Encryption: The DHAES Construction

To improve over the security of the ElGamal public-key encryption scheme, which as we will see does not resist extended notions of an attack, the scheme *DHAES* [1] was devised. This scheme combines multiple cryptographic primitives: a key encapsulation mechanism (KEM), a one-time message authentication code (one-time MAC), and a symmetric cipher. Constructions for public-key encryption that involve symmetric encryption are often called *hybrid*.

DHAES stands for *Diffie-Hellman Authenticated Encryption Scheme* or *Diffie-Hellman Augmented Encryption Scheme*. The scheme as originally published uses the Diffie-Hellman key exchange mechanism as in protocol 2.4. A later variant of the scheme published as *DHIES*, which stands for *Diffie-Hellman Integrated Encryption Scheme* [2], uses protocol 2.3 instead (there is some indication that this change of the KEM was ill-advised from a security point of view [80]). Other KEMs can be used without changing the essence of the DHAES/DHIES scheme. The generalization for arbitrary key encapsulation mechanisms was first laid out in [30]. In the following, the term “the DHAES construction” refers to the general scheme using any KEM.

In this chapter, we will look at the DHAES construction for arbitrary key encapsulation mechanisms, but only for specific symmetric ciphers, namely XOR-based stream ciphers (where the ciphertext is generated from the plaintext and vice versa by XORing a pseudo-random bit string) – a case that was accidentally not covered by the original security result in [1] and [2]. The security result for the DHAES construction that we will arrive at leans on [30].

Section 3.1 describes chosen-ciphertext attacks against public-key encryption schemes and shows that the ElGamal public-key encryption scheme is not secure under this strong notion. Then section 3.2 describes the cryptographic primitives needed by the DHAES construction, and section 3.3 describes how they are used to encrypt and decrypt with DHAES. Finally, section 3.4 shows how the security of DHAES follows from the security of the underlying primitives.

3.1 Adaptive Chosen-Ciphertext Attacks

In section 2.3.1, we have looked at a basic security notion for public-key encryption, indistinguishability under chosen-plaintext attack (IND-CPA). In practice, adversaries are often in a better position to launch an attack than provided in that attack model: they may be able to make up ciphertexts and submit these *chosen ciphertexts* for decryption. Depending on the specific attack scenario, the decryption results may or may not be made completely available to the adversary. We are interested in a very strong characterization of security for public-key encryption, so we assume the worst and use attack models where the adversary has access to a *decryption oracle*.

Remember that in the attack scenario for IND-CPA, the adversary chooses two messages m_0 and m_1 , obtains the encryption of a random one m_b of these, and then has to guess whether $b = 0$ or $b = 1$. The adversary can be considered as running its attack in two stages: first the *find stage*, which ends when the adversary outputs m_0 and m_1 , and then the *guess stage*, which starts when the adversary is given the challenge ciphertext. If we extend this attack scenario to allow a *chosen-ciphertext attack*, we always provide the adversary with a decryption oracle for the find stage. The strongest notion of a chosen-ciphertext attack is an *adaptive chosen-ciphertext attack* [74], where the adversary also has access to a decryption oracle in the guess stage; for the guess-stage oracle, we have to impose the restriction that the adversary may not use the literal challenge ciphertext as a query (otherwise, the oracle's answer would make it trivial to determine b for any sound public-key encryption scheme if $m_0 \neq m_1$).

Sometimes both non-adaptive chosen-ciphertext attacks (CCA1) and adaptive chosen-ciphertext attacks (CCA2) are considered in the literature, but usually only the strongest notion of attack is relevant. In the following, the term chosen-ciphertext attack (CCA) without further qualification is understood to refer to the adaptive kind.

Before we proceed to formalize CCA security for public-key encryption, first we define the notion of a *public-key encryption scheme* more completely. Here we assume that all messages are encoded as bit strings (if we start with a scheme defined for group elements, such as the ElGamal encryption scheme as sketched in section 2.2, we have to add appropriate encoding and decoding functionality to the system's specification). For this definition, remember that algorithms are usually probabilistic (refer to the notational conventions at the beginning of the present chapter). Note that our formalization uses no explicit security parameter; this corresponds to having some fixed security parameter built into the key generation algorithm.

Definition 3.1. A public-key encryption scheme

PKE

specifies a key generation algorithm

PKE.KeyGen

that produces key pairs (PK,SK) consisting of a public key PK and a secret key SK, an algorithm

PKE.Encrypt

using the public key, and an algorithm

PKE.Decrypt

using the secret key. For a plaintext m (an arbitrary bit string, possibly subject to length limitations for the specific public-key encryption scheme),

$$\text{PKE.Encrypt}(\text{PK}, m)$$

returns a bit string c , a ciphertext. On arbitrary input c' ,

$$\text{PKE.Decrypt}(\text{SK}, c')$$

may either return some string m' or fail and return the special value `invalid`. If the key pair (PK, SK) has been produced by PKE.KeyGen and c has been produced by $\text{PKE.Encrypt}(\text{PK}, m)$, then evaluation of $\text{PKE.Decrypt}(\text{SK}, c)$ will return string m .

Security in the sense of *indistinguishability under chosen-ciphertext attack* (IND-CCA) is described through the following attack game:

1. The adversary queries a *key generation oracle*, which uses PKE.KeyGen to determine a key pair

$$(\text{PK}, \text{SK})$$

and responds with PK (while secretly storing SK).

2. [**Find stage.**] The adversary makes a sequence of queries to a *decryption oracle*. Each query is an arbitrary string s , and the oracle responds with $\text{PKE.Decrypt}(\text{SK}, s)$.
3. The adversary chooses messages m_0 and m_1 subject only to the condition that $|m_0| = |m_1|$ and sends them to an *encryption oracle*. The encryption oracle chooses a uniformly random bit $b \in \{0, 1\}$ and determines

$$c = \text{PKE.Encrypt}(\text{PK}, m_b),$$

which is returned to the adversary as the *challenge ciphertext*.

4. [**Guess stage.**] The adversary again makes a sequence of queries to a *decryption oracle* as in the find stage, where this time the decryption oracle refuses being asked for the challenge ciphertext c (it returns `invalid` for this case).
5. The adversary outputs a bit $\tilde{b} \in \{0, 1\}$.

The bit \tilde{b} output by the adversary is supposed to be its guess for the value of b .

Now let A be any adversary in the above attack game, i.e. an interactive probabilistic algorithm with bounded running time. Its *CCA advantage* against the public key encryption scheme is

$$\text{AdvCCA}_{\text{PKE}, A} = \left| \Pr[\tilde{b} = 1 \mid b = 1] - \Pr[\tilde{b} = 1 \mid b = 0] \right|.$$

Similarly to what we have seen in section 2.3.1, this can also be written as

$$2 \cdot \left| \Pr[\tilde{b} = b] - \frac{1}{2} \right|.$$

Let us consider the security of the ElGamal public-key encryption scheme (see section 2.2). If messages are encoded as group elements, then IND-CPA security (section 2.3.1) can be

proven provided that the scheme is used in groups for which the so-called Decision Diffie-Hellman (DDH) assumption holds; see [85] for details on the reduction. (The DDH assumption states essentially that adversaries trying to tell apart the random distributions (g^a, g^b, g^{ab}) and (g^a, g^b, y) with $a, b \in_{\mathbb{R}} \{0, \dots, \#(G) - 1\}$, $y \in_{\mathbb{R}} G$ cannot do significantly better than guess.) Now let us consider ElGamal encryption under the stronger IND-CCA security notion. ElGamal ciphertexts for plaintext m have the general structure $(\mathfrak{R}, m + K)$, and for any group element x , computing $(\mathfrak{R}, x + m + K)$ yields a ciphertext corresponding to plaintext $x + m$. Thus, given the challenge ciphertext, the adversary can easily derive a related ciphertext that may be used as a query to the decryption oracle so that the plaintext can be recovered from the oracle's reply simply by subtracting a group element. This makes it easy to successfully attack the encryption scheme in the IND-CCA scenario; clearly, ElGamal encryption is not secure in this sense.

Thus, if the IND-CCA notion of cryptographic security must be met, the ElGamal public-key encryption scheme is not sufficient. In this chapter, we will examine the DHAES construction, which achieves this goal.

3.2 Primitives for the DHAES Construction

The DHAES construction as presented in the following requires three underlying cryptographic *primitives*: a key encapsulation mechanism (KEM), a one-time message authentication code (one-time MAC), and a pseudo-random bit string generator for use in an XOR-based stream cipher. (Note that DHAES/DHIES as described in [1] and [2] can use other symmetric ciphers as well.)

Definition 3.2. A key encapsulation mechanism

KEM

specifies a key generation algorithm

KEM.KeyGen

that produces key pairs (PK, SK) consisting of a public key PK and a secret key SK, an algorithm

KEM.Encrypt

using the public key, and an algorithm

KEM.Decrypt

using the secret key. In contrast with public-key encryption, the Encrypt algorithm of a key encapsulation mechanism takes no input apart from the public key:

KEM.Encrypt(PK)

generates a ciphertext \mathfrak{R} of a fixed length

KEM.CipherLen

corresponding to a (pseudo-)random string K of a fixed length

KEM.OutLen

and outputs the pair (K, \mathfrak{R}) . Evaluation of

KEM.Decrypt(SK, \mathfrak{R})

will return said string K if the key pair (PK, SK) has been produced by KEM.KeyGen. On arbitrary input \mathfrak{R}' , the computation KEM.Decrypt(SK, \mathfrak{R}') may either return some string K' or fail (return the special value *invalid*).

The random message K will be known both to the party running KEM.Encrypt(PK) and to the secret key owner who runs KEM.Decrypt(SK, \mathfrak{R}). In this sense, ciphertext \mathfrak{R} encapsulates the random message K . It can be used as a key for symmetric-key cryptography; hence the term “key encapsulation”.

One choice for a key encapsulation mechanism is the Diffie-Hellman variant from protocol 2.4. KEM.CipherLen can be kept particularly small by using the group of rational points on an appropriate elliptic curve [12] and using (except in internal computations) just x coordinates of points (see [55]). Specifications for various key encapsulation mechanisms can be found in [80].

Definition 3.3. A one-time message authentication code (*one-time MAC*)

MAC

specifies a key length

MAC.KeyLen,

an output length

MAC.OutLen,

and a deterministic algorithm that takes as input a key K of length MAC.KeyLen and a bit string s (of theoretically arbitrary length, although practical realizations will typically have some large fixed limit) and returns a string

MAC(K, s)

of length MAC.OutLen.

Candidate one-time message authentication codes are UHASH [50] and HMAC [5]. (The UHASH scheme is the combination of a *key derivation function* with an *almost strongly universal hash function* [89] and is the core of UMAC as specified in [50]. Note that there is an earlier version of UMAC described in [11], for which security arguments are provided that are based on a different approach.)

Definition 3.4. A pseudo-random bit string generator

STREAM

specifies a key length

STREAM.KeyLen,

an output length

STREAM.OutLen,

and a deterministic algorithm taking as input a key K of length STREAM.KeyLen and generating an output string

STREAM(K)

of length STREAM.OutLen. Alternatively,

STREAM.OutLen

may be infinite; in this case, for any integer n ,

STREAM(K, n)

generates an output stream of length n .

A convenient example implementation is the so-called *counter mode* of a symmetric block cipher (the output sequence is the prefix of appropriate length of $E_K(0) \parallel E_K(1) \parallel E_K(2) \parallel \dots$ where E_K denotes block cipher encryption using key K ; see [6] and [52]).

3.3 The DHAES Public-Key Encryption Scheme

We specify DHAES as a public-key encryption scheme PKE following definition 3.1, assuming that primitives KEM, MAC, and STREAM as described in section 3.2 are available. We require that

$$\text{KEM.OutLen} = \text{MAC.KeyLen} + \text{STREAM.KeyLen}$$

and that

$$\text{STREAM.OutLen} = \infty.$$

DHAES permits encrypting messages of arbitrary length.

The *key generation algorithm* PKE.KeyGen for DHAES is simply KEM.KeyGen.

The *encryption algorithm* determines PKE.Encrypt(PK, m) as follows:

1. Use KEM.Encrypt(PK) to generate a pair (K, \mathfrak{K}) .
2. Split K in the form

$$K = K_{\text{MAC}} \parallel K_{\text{STREAM}}$$

such that $|K_{\text{MAC}}| = \text{MAC.KeyLen}$ (and thus $|K_{\text{STREAM}}| = \text{STREAM.KeyLen}$).

3. Compute $\mathfrak{C} = m \oplus \text{STREAM}(K_{\text{STREAM}}, |m|)$.
4. Compute $\mathfrak{M} = \text{MAC}(K_{\text{MAC}}, \mathfrak{C})$.
5. Return the ciphertext $\mathfrak{K} \parallel \mathfrak{M} \parallel \mathfrak{C}$.

The *decryption algorithm* computes PKE.Decrypt(PK, $\mathfrak{K} \parallel \mathfrak{M} \parallel \mathfrak{C}$) as follows (note that the ciphertext can be uniquely split into its three components because KEM.CipherLen and MAC.OutLen are fixed):

1. Compute

$$K = \text{KEM.Decrypt}(\text{SK}, \mathfrak{R}).$$

If this computation fails, abort with an error (return invalid).

2. Split K in the form

$$K = K_{\text{MAC}} \parallel K_{\text{STREAM}}$$

such that $|K_{\text{MAC}}| = \text{MAC.KeyLen}$.

3. Compute

$$\widetilde{\mathfrak{M}} = \text{MAC}(K_{\text{MAC}}, \mathfrak{C})$$

and test whether

$$\widetilde{\mathfrak{M}} = \mathfrak{M}.$$

If this is not the case, abort with an error (return invalid).

4. Return the string

$$\mathfrak{C} \oplus \text{STREAM}(K_{\text{STREAM}}, |\mathfrak{C}|)$$

as decryption result.

3.4 The Security of the DHAES Construction

In section 3.1, we have seen how the security of a public-key encryption scheme against adaptive chosen-ciphertext attacks can be captured quantitatively. The DHAES construction is designed to be secure against such attacks provided that the underlying primitives fulfill certain security notions.

The present section first defines according quantitative expressions of security for the primitives used in the DHAES construction (sections 3.4.1, 3.4.2, and 3.4.3) and then gives a security result that relates the IND-CCA security of DHAES to the security of these primitives (section 3.4.4).

3.4.1 Security of the Key Encapsulation Mechanism

Security against *adaptive chosen-ciphertext attacks* for the key encapsulation mechanism KEM is expressed through the following attack game (cf. [30, section 7.1.2]):

1. The adversary queries a *key generation oracle*, which uses KEM.KeyGen to compute a key pair

$$(\text{PK}, \text{SK})$$

and responds with PK (and secretly stores SK).

2. The adversary makes a sequence of queries to a *decryption oracle*. Each query is an arbitrary string s of length KEM.CipherLen ; the oracle responds with

$$\text{KEM.Decrypt}(\text{SK}, s).$$

Thus each oracle response is either a string of length KEM.OutLen or the special value *invalid*.

3. The adversary queries a *key encapsulation oracle*, which works as follows: it uses

$$\text{KEM.Encrypt}(\text{PK})$$

to obtain a pair $(K_0, \mathfrak{K}_{\text{oracle}})$, it generates a uniformly random string K_1 such that $|K_0| = |K_1|$, chooses a uniformly random bit $b_{\text{KEM}} \in \{0, 1\}$, and responds with

$$(K_{b_{\text{KEM}}}, \mathfrak{K}_{\text{oracle}})$$

as *challenge*.

4. The adversary again makes a sequence of queries to a *decryption oracle* as in stage 2, where this time the oracle refuses the specific query $\mathfrak{K}_{\text{oracle}}$ (and responds invalid for this case).
5. The adversary outputs a bit $\tilde{b}_{\text{KEM}} \in \{0, 1\}$.

The *CCA advantage* of an adversary A (an interactive probabilistic algorithm with bounded running time) against KEM in this attack game is

$$\text{AdvCCA}_{\text{KEM}, A} = \left| \Pr[\tilde{b}_{\text{KEM}} = 1 \mid b_{\text{KEM}} = 1] - \Pr[\tilde{b}_{\text{KEM}} = 1 \mid b_{\text{KEM}} = 0] \right|.$$

This can also be written as

$$2 \cdot \left| \Pr[\tilde{b}_{\text{KEM}} = b_{\text{KEM}}] - \frac{1}{2} \right|.$$

3.4.2 Security of the One-Time Message Authentication Code

To express the security of MAC, we use the following attack game (cf. [30, section 7.2.2]):

1. The adversary submits a string s to a *MAC oracle*. This oracle generates a uniformly random string K of length MAC.KeyLen and responds with $\text{MAC}(K, s)$.
2. The adversary outputs a list $(s_1, t_1), (s_2, t_2), \dots, (s_m, t_m)$ of pairs of strings.

An adversary A again is an interactive probabilistic algorithm with bounded running time; note that the running time bound implies a bound on the length m of the list. We say that adversary A has *produced a forgery* if

$$\text{MAC}(K, s_k) = t_k$$

and $s_k \neq s$ for some k ($1 \leq k \leq m$). The adversary's advantage against MAC, denoted

$$\text{AdvForge}_{\text{MAC}, A},$$

is the probability that it produces a forgery in the above game.

3.4.3 Security of the Pseudo-Random Bit String Generator

To express the security of STREAM, we use the following attack game:

1. The adversary queries a *bit string oracle*. This oracle generates a uniformly random string K of length STREAM.KeyLen , computes $\text{stream}_0 = \text{STREAM}(K)$, generates a uniformly random string stream_1 with $|\text{stream}_0| = |\text{stream}_1|$, chooses a uniformly random bit $b_{\text{STREAM}} \in \{0, 1\}$, and responds with

$$\text{stream}_b$$

as *challenge*.

2. The adversary outputs a bit $\tilde{b}_{\text{STREAM}} \in \{0, 1\}$.

The *advantage* of an adversary A (again an interactive probabilistic algorithm with bounded running time) against STREAM is

$$\text{Adv}_{\text{STREAM}, A} = \left| \Pr[\tilde{b}_{\text{STREAM}} = 1 \mid b_{\text{STREAM}} = 1] - \Pr[\tilde{b}_{\text{STREAM}} = 1 \mid b_{\text{STREAM}} = 0] \right|,$$

which can also be written as

$$2 \cdot \left| \Pr[\tilde{b}_{\text{STREAM}} = b_{\text{STREAM}}] - \frac{1}{2} \right|.$$

3.4.4 Security Result for the DHAES Construction

Let A be an adversary launching an adaptive chosen-ciphertext attack (see the attack game in section 3.1) against DHAES with some key encapsulation mechanism KEM, some one-time message authentication code MAC, and some pseudo-random bit string generator STREAM. It can be shown that there are adversaries A_1, A_2, A_3 against KEM, MAC, STREAM all having essentially the same running time as A such that

$$\text{Adv}_{\text{CCA}_{\text{PKE}, A}} \leq 2 \cdot (\text{Adv}_{\text{CCA}_{\text{KEM}, A_1}} + \text{Adv}_{\text{Forge}_{\text{MAC}, A_2}} + \text{Adv}_{\text{STREAM}, A_3}).$$

This quantitative security result can be interpreted as saying essentially that if DHAES is not secure, then this is because one of the underlying cryptographic primitives is not secure.

In the remainder of this section, we will prove this result. Let \mathbf{G}_0 denote the attack game described in section 3.1. We will modify it in multiple steps, essentially disabling the underlying cryptographic schemes (key encapsulation mechanism, one-time message authentication code, and pseudo-random bit string generator) one after another.

For DHAES, stage 3 (invocation of the encryption oracle) in the attack game from section 3.1 can be expressed as follows. First the adversary submits messages m_0 and m_1 such that $|m_0| = |m_1|$. The encryption oracle uses $\text{KEM.Encrypt}(\text{PK})$ to generate a pair $(K_{\text{oracle}}, \mathfrak{R}_{\text{oracle}})$ and splits K_{oracle} in the form

$$K_{\text{oracle}} = K_{\text{MAC}} \parallel K_{\text{STREAM}}$$

such that $|K_{\text{MAC}}| = \text{MAC.KeyLen}$. Then it chooses a uniformly random bit $b \in \{0, 1\}$, computes

$$\mathfrak{C} = m_b \oplus \text{STREAM}(K_{\text{STREAM}}, |m_b|)$$

and

$$\mathfrak{M} = \text{MAC}(K_{\text{MAC}}, \mathfrak{C}),$$

and returns

$$\mathfrak{K}_{\text{oracle}} \parallel \mathfrak{M} \parallel \mathfrak{C}$$

as challenge ciphertext. In the following, when we talk about the encryption oracle stage, it is understood that we refer to this procedure.

\mathbf{G}_1 is like \mathbf{G}_0 except that a uniformly random string is used for K_{oracle} , whereas $\mathfrak{K}_{\text{oracle}}$ is still generated by $\text{KEM.Encrypt}(\text{PK})$. In the decryption oracle, K_{oracle} is substituted whenever $\text{KEM.Decrypt}(\text{SK}, \mathfrak{K}_{\text{oracle}})$ would have to be computed. This applies to both the find and the guess stage, i.e. stages 2 and 4 in the attack game from section 3.1. Thus, the invocation of $\text{KEM.Encrypt}(\text{PK})$ to generate $\mathfrak{K}_{\text{oracle}}$ must be advanced from stage 3 to an earlier stage. In \mathbf{G}_0 , this is only a descriptive change and does not affect the behavior observed by the adversary.

\mathbf{G}_2 is like \mathbf{G}_1 except that the decryption oracle always responds with invalid when faced with any query prefixed with string $\mathfrak{K}_{\text{oracle}}$.

\mathbf{G}_3 is like \mathbf{G}_2 except that the encryption oracle uses a uniformly random string *stream* of the appropriate length instead of computing $\text{STREAM}(K_{\text{STREAM}}, |m_b|)$.

Now we consider an adversary A as in section 3.1, exposed to these different attack games \mathbf{G}_x , $0 \leq x \leq 3$, and look at the respective success probabilities $\Pr_{\mathbf{G}_x}[\tilde{b} = b]$. Based on A , adversaries A_1, A_2, A_3 against KEM, MAC, STREAM will be built all having essentially the same running time as A .

A_1 attacks KEM as follows (cf. section 3.4.1). At first, it generates $b \in \{0, 1\}$ uniformly at random and queries its key encapsulation oracle to obtain a pair $(K_{\text{oracle}}, \mathfrak{K}_{\text{oracle}})$. Then, it runs the adversary A , playing the roles of the decryption oracle and encryption oracle: when A queries its decryption oracle, A_1 uses its own decryption oracle to compute $\text{KEM.Decrypt}(\text{SK}, \mathfrak{K})$ when performing the decryption algorithm from section 3.3, substituting K_{oracle} when $\text{KEM.Decrypt}(\text{SK}, \mathfrak{K}_{\text{oracle}})$ would have to be computed; when A queries its encryption oracle, A_1 performs the encryption oracle stage using the pregenerated pair $(K_{\text{oracle}}, \mathfrak{K}_{\text{oracle}})$ and the pregenerated bit b . Finally, when A outputs its bit \tilde{b} , A_1 outputs 1 if $\tilde{b} = b$ and 0 otherwise. Observe that

$$\left| \Pr_{\mathbf{G}_1}[\tilde{b} = b] - \Pr_{\mathbf{G}_0}[\tilde{b} = b] \right| = \text{AdvCCA}_{\text{KEM}, A_1}$$

(\mathbf{G}_0 corresponds to $b_{\text{KEM}} = 0$, \mathbf{G}_1 corresponds to $b_{\text{KEM}} = 1$ in section 3.4.1).

A_2 attacks MAC as follows (cf. section 3.4.2). At first, it generates $b \in \{0, 1\}$ and a string K_{STREAM} of length STREAM.KeyLen uniformly at random and uses $\text{KEM.Encrypt}(\text{PK})$ to generate $\mathfrak{K}_{\text{oracle}}$. Then, it runs the adversary A , playing the roles of the key generation oracle (so that A_2 knows SK), decryption oracle, and encryption oracle. Whenever A submits any query $\mathfrak{K} \parallel \mathfrak{M} \parallel \mathfrak{C}$ to the decryption oracle, A_2 adds the pair $(\mathfrak{C}, \mathfrak{M})$ to its own output; queries prefixed with $\mathfrak{K}_{\text{oracle}}$ are refused, all other questions are answered using SK. In the encryption oracle stage, A_2 substitutes the pregenerated string K_{STREAM} and the pregenerated bit b , and it invokes its MAC oracle on \mathfrak{C} instead of using MAC directly to obtain \mathfrak{M} for the challenge ciphertext. A 's final output bit \tilde{b} is ignored. Observe that

$$\left| \Pr_{\mathbf{G}_2}[\tilde{b} = b] - \Pr_{\mathbf{G}_1}[\tilde{b} = b] \right| \leq \text{AdvForge}_{\text{MAC}, A_2}$$

(A_2 runs A as in game \mathbf{G}_2 , and A would observe different behavior in game \mathbf{G}_1 only in the cases where a forgery is produced).

A_3 attacks **STREAM** as follows (cf. section 3.4.3). First, it generates $b \in \{0, 1\}$ and a string K_{oracle} of length KEM.OutLen uniformly at random. Then, it runs the adversary A , playing the roles of the key generation oracle, encryption oracle, and decryption oracle, substituting the pregenerated string K_{oracle} and the pregenerated bit b in the encryption oracle stage, and refusing any decryption oracle query prefixed with $\mathfrak{K}_{\text{oracle}}$ (both in the find stage and in the guess stage). When A outputs its bit \tilde{b} , A_3 outputs 1 if $\tilde{b} = b$ and 0 otherwise. Observe that

$$\left| \Pr_{\mathbf{G}_3}[\tilde{b} = b] - \Pr_{\mathbf{G}_2}[\tilde{b} = b] \right| = \text{Adv}_{\text{STREAM}, A_3}$$

(\mathbf{G}_2 corresponds to $b_{\text{STREAM}} = 0$, \mathbf{G}_3 corresponds to $b_{\text{STREAM}} = 1$ in section 3.4.3).

Finally observe that $\Pr_{\mathbf{G}_3}[\tilde{b} = b] = \frac{1}{2}$.

From this we obtain the inequality

$$\begin{aligned} \text{Adv}_{\text{CCA}_{\text{PKE}}, A} &= 2 \cdot \left| \frac{1}{2} - \Pr_{\mathbf{G}_0}[\tilde{b} = b] \right| \\ &= 2 \cdot \left| \sum_{1 \leq x \leq 3} \left(\Pr_{\mathbf{G}_x}[\tilde{b} = b] - \Pr_{\mathbf{G}_{x-1}}[\tilde{b} = b] \right) \right| \\ &\leq 2 \cdot (\text{Adv}_{\text{CCA}_{\text{KEM}}, A_1} + \text{Adv}_{\text{Forge}_{\text{MAC}}, A_2} + \text{Adv}_{\text{STREAM}, A_3}), \end{aligned}$$

which concludes the proof.

Chapter 4

A Public-Key Cryptosystem for Length-Preserving Chaumian Mixes

Chaum's *mix* concept [24] is intended to allow users to send untraceable electronic mail without having to trust a single authority. The idea is to use a number of intermediates such that it suffices for just one of these to be trustworthy in order to achieve untraceability. The sender does not have to decide which particular intermediate he is willing to trust, he just must be convinced that at least one in a given list will behave as expected. These intermediates, the *mixes*, are remailers accepting public-key encrypted input. Messages must be of a fixed size (shorter messages can be padded, longer messages can be split into multiple parts). To send a message, it is routed through a chain of mixes M_1, \dots, M_n : the sender obtains the public key of each mix; then he recursively encrypts the message (including the address of the final recipient) yielding $E_{M_1}(E_{M_2}(\dots E_{M_n}(\textit{payload}) \dots))$ where E_{M_i} denotes encryption with M_i 's public key, and sends the resulting ciphertext to mix M_1 . Each mix removes the corresponding layer of encryption and forwards the decrypted message to the next mix; thus mix M_n will finally recover *payload*.

Each mix is expected to collect a large batch of messages before forwarding the decryption results. The messages in the batch must be reordered (mixed) to prevent message tracing. It is important to prevent replay attacks: a mix must not process the same message twice, or active adversaries would be able to trace messages by duplicating them at submission to cause multiple delivery. (Timestamps can be used to limit the timespan for which mixes have to remember which messages they have already processed; see [24] and [29].)

Usually it is desirable to allow *source routing*, i.e. let senders choose mix chains on a per-message basis. This increases the flexibility of the whole scheme: senders can make use of new mixes that go into operation, and they can avoid mixes that appear not to work properly; in particular, they can avoid mixes that suppress messages (be it intentionally or because of technical problems), which might be noticed when sending probe messages to oneself over mix chains. For source routing, in the recursively encrypted message, each layer must contain the address of the next mix in the chain so that each mix knows where to forward the message. A problem with the straightforward implementation of source routing is that messages will shrink as they proceed through the chain, not only because of the forwarding address for each layer that must be included, but also because public-key encryption increases message size. For optimal untraceability, we need *length-preserving mixes*: the messages that mixes receive should essentially look like the resulting messages that they forward to other mixes.

A construction for length-preserving mixes is given in [24] (a variant of this is used in the fielded system Mixmaster [56]): mix messages consist of a fixed number of slots of a fixed size. The first slot is public-key encrypted so that it can be read by the first mix in the chain. Besides control information directed at that mix (such as the address of the next mix in the chain or, in case of the last mix, the address of the final recipient), decryption yields a symmetric key that the mix uses to decrypt all the other slots. Then slots are shifted by one position: the decryption of the second slot becomes the new first slot, and so on. A new final slot is added consisting of random (garbage) data to obtain a mix message of the desired fixed length, which can then be forwarded to the next mix. On the way through the chain, each mix message consists of a number of slots with valid data followed by enough slots with garbage to fill all available slots; mixes are oblivious of how many slots contain valid data and how many are random. Each mix in the chain, when decrypting and shifting the slots, will “decrypt” the garbage slots already present and add a new final slot, thus increasing the number of garbage slots by one. We note that while the transformation of an incoming mix message to the corresponding outgoing mix message is *length-preserving*, the decryption step itself is actually *length-expanding* because some of the data obtained by decryption is control data intended for the current mix.

The problem with this slot-based approach for obtaining a hybrid public-key cryptosystem with length-expanding decryption is that it is not secure against active attacks: assume that an adversary controls at least one mix, and that all senders submit well-formed messages. Now when the victim submits a message, the adversary can mark it by modifying one of the slots. This mark will persist as the message is forwarded through the mix chain: due to the decryption and slot shifting performed by each mix, the corresponding slot will always be somehow different from what it should look like (whereas unmarked messages will not show such defects). If the final mix is controlled by the adversary, the adversary may be able to notice the modification, e.g. detect a region of garbage bits in an otherwise comprehensible message, and thus break untraceability.

To rule out such attacks, the public-key cryptosystem should provide security against adaptive chosen ciphertext attacks (CCA security). The usual notion of CCA security for public-key encryption schemes as presented in section 3.1 is not applicable to public-key cryptosystems with length-expanding decryption because the encryption operation must be defined differently here: encryption cannot be treated as an atomic black-box operation that, given a fixed public key, takes a plaintext and returns a ciphertext (within this approach, length-expanding decryption could not recover the original plaintext). Rather, the encryption process that we will use first is input part of the desired *ciphertext* and determines a corresponding part of what will be the decryption result (it is this step that provides length expansion); then it is input the payload plaintext and finally outputs the complete ciphertext, which includes the portion requested in the first input.

Section 4.1 describes a secure and practical hybrid construction for length-preserving mixes, which is also more flexible than the slot-based approach. The structure of each layer of encryption resembles that of the public-key encryption scheme DHAES (see chapter 3). Section 4.2 discusses appropriate security notions and gives provable security results for the construction.

This chapter shows essentially how Mixmaster ([56], [64]) should have been specified to be cryptographically secure against active attacks. (Note that an entirely different model of operation for mix networks is assumed by Ohkubo and Abe [68] and Jakobsson and Juels [44]: these constructions assume the existence of a shared authenticated bulletin board, whereas

here we are interested in an open system that can be used with point-to-point communication by e-mail or similar means.) Various ideas and techniques that are familiar from the previous chapter on the DHAES construction are transferred to fit the new notions of security needed in the context of length-preserving mixes.

4.1 The Mix Encryption Scheme

The construction presented in the following allows much flexibility in the choice of cryptographic schemes, even in a single chain. The single parameter that must be fixed for all mixes is the desired mix message length ℓ . Also it may be desirable to define a maximum length for the actual message payload, i.e. the part of the plaintext as recovered by the final mix that can be chosen by the sender: as a message proceeds through the mix chain, more and more of the data will be pseudo-random gibberish; the length of the useful part of the final plaintext reveals that chains leaving less than this amount cannot have been used. The length n of the mix chain (i.e. the number of consecutive mixes) need not be fixed.

For purposes of exposition, we number the mixes M_1, \dots, M_n according to their position in the chain chosen by the sender. (Note that in practice the same mix might appear multiple times in one chain.) For each mix M_i , the following must be defined and, with the exception of the secret key SK_{M_i} , known to senders who want to use the mix:

- A *key encapsulation mechanism*

$$KEM_{M_i}$$

(see definition 3.2) and a key pair

$$(PK_{M_i}, SK_{M_i})$$

for this key encapsulation mechanism as generated by

$$KEM_{M_i}.KeyGen.$$

- A *one-time message authentication code*

$$MAC_{M_i}$$

(see definition 3.3). In our construction, MAC_{M_i} will only be used for strings s of fixed length $\ell - KEM_{M_i}.CipherLen - MAC_{M_i}.OutLen$.

- A *pseudo-random bit string generator*

$$STREAM_{M_i}$$

(see definition 3.4) with a fixed output size

$$STREAM_{M_i}.OutLen.$$

This will be used for an XOR-based stream cipher.

- An integer

PlainLen_{M_i}

specifying the length of the prefix of each decrypted message that is considered control data directed to mix M_{M_i} and will not be forwarded. (This is the amount of message expansion: the decrypted message minus the prefix must be of size ℓ because that is what will be sent to the next mix.)

The parameters must fulfill the following conditions:

$$\text{KEM}_{M_i}.\text{CipherLen} + \text{MAC}_{M_i}.\text{OutLen} + \text{PlainLen}_{M_i} < \ell \quad (4.1)$$

$$\text{KEM}_{M_i}.\text{OutLen} = \text{STREAM}_{M_i}.\text{KeyLen} + \text{MAC}_{M_i}.\text{KeyLen} \quad (4.2)$$

$$\text{STREAM}_{M_i}.\text{OutLen} = \text{PlainLen}_{M_i} + \ell \quad (4.3)$$

4.1.1 Encryption

We now describe encryption for sending a message through a chain M_1, \dots, M_n . Let *payload* be the message of length

$$|\text{payload}| = \ell - \sum_{1 \leq i \leq n} (\text{KEM}_{M_i}.\text{CipherLen} + \text{MAC}_{M_i}.\text{OutLen} + \text{PlainLen}_{M_i}) \quad (4.4)$$

(messages shorter than this maximum should be randomly padded on the right). For each i , let plain_i be the control message of length PlainLen_{M_i} directed to the respective mix. The encryption algorithm for these arguments is denoted

$$\text{chain_encrypt}_{M_1, \dots, M_n}(\text{plain}_1, \dots, \text{plain}_n; \text{payload})$$

or

$$\text{chain_encrypt}_{M_1, \dots, M_n}(\text{plain}_1, \dots, \text{plain}_n; \text{payload}; \lambda)$$

where λ is the empty string. The algorithm is defined recursively. Let $1 \leq i \leq n$, and let C_i be a string of length

$$|C_i| = \sum_{1 \leq k < i} (\text{KEM}_{M_k}.\text{CipherLen} + \text{MAC}_{M_k}.\text{OutLen} + \text{PlainLen}_{M_k}) \quad (4.5)$$

(thus specifically $|C_1| = 0$, i.e. C_1 is the empty string). Then algorithm

$$\text{chain_encrypt}_{M_i, \dots, M_n}(\text{plain}_i, \dots, \text{plain}_n; \text{payload}; C_i)$$

works as follows:

1. Use $\text{KEM}_{M_i}.\text{Encrypt}(\text{PK}_{M_i})$ to generate a pair (K_i, \mathfrak{K}_i) .
2. Split K_i in the form

$$K_i = K_{i,\text{MAC}} \parallel K_{i,\text{STREAM}}$$

such that $|K_{i,\text{MAC}}| = \text{MAC}_{M_i}.\text{KeyLen}$ (and, by (4.2), $|K_{i,\text{STREAM}}| = \text{STREAM}_{M_i}.\text{KeyLen}$).

3. Compute $\text{STREAM}_{M_i}(K_{i,\text{STREAM}})$ and split this string in the form

$$\text{stream}_{i,L} \parallel \text{stream}_{i,R}$$

such that the left part $\text{stream}_{i,L}$ is of length

$$\ell - |C_i| - \text{KEM}_{M_i}.\text{CipherLen} - \text{MAC}_{M_i}.\text{OutLen}.$$

4. For $i = n$ (last mix), by (4.4) and (4.5) it follows that $|\text{stream}_{n,L}| = \text{PlainLen}_n + |\text{payload}|$. In this case, set

$$\mathfrak{C}_n = (\text{stream}_{n,L} \oplus (\text{plain}_n \parallel \text{payload})) \parallel C_n.$$

Otherwise, let

$$C_{i+1} = \text{stream}_{i,R} \oplus (C_i \parallel 0^{\text{KEM}_{M_i}.\text{CipherLen} + \text{MAC}_{M_i}.\text{OutLen} + \text{PlainLen}_{M_i}}),$$

by recursion compute

$$x_i = \text{chain_encrypt}_{M_{i+1}, \dots, M_n}(\text{plain}_{i+1}, \dots, \text{plain}_n; \text{payload}; C_{i+1}),$$

and let

$$\mathfrak{C}_i = (\text{stream}_{i,L} \oplus (\text{plain}_i \parallel \text{prefix}_{|\text{stream}_{i,L}| - \text{PlainLen}_{M_i}}(x_i))) \parallel C_i.$$

5. Compute $\mathfrak{M}_i = \text{MAC}_i(K_{i,\text{MAC}}, \mathfrak{C}_i)$.

6. Return the ciphertext

$$\mathfrak{R}_i \parallel \mathfrak{M}_i \parallel \mathfrak{C}_i,$$

which is of length ℓ .

The following illustration depicts a ciphertext generated by the encryption algorithm for a chain of length three, namely

$$\text{chain_encrypt}_{M_1, M_2, M_3}(\text{plain}_1, \text{plain}_2, \text{plain}_3; \text{payload}).$$

In the illustration, we have concatenation horizontally and XOR vertically (i.e. boxes in the same row represent bit strings that are concatenated, and the ciphertext is the XOR of the multiple rows shown).

						plain ₃		payload	
				plain ₂		R ₃		M ₃	
		plain ₁		R ₂		M ₂		stream _{3,L}	
R ₁		M ₁		stream _{2,L}					
stream _{1,L}									

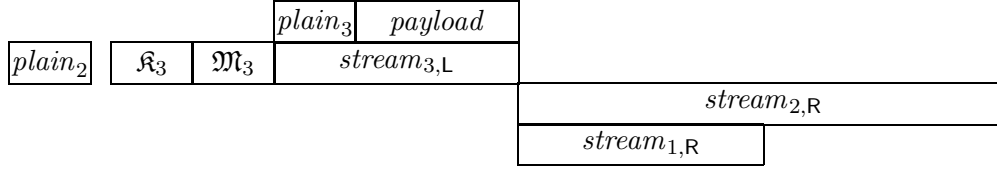
4.1.2 Decryption

The decryption algorithm for mix M_i ($1 \leq i \leq n$) works as follows, given a length- ℓ ciphertext $\mathfrak{R} \parallel \mathfrak{M} \parallel \mathfrak{C}$ (split into its three components according to parameters $\text{KEM}_{M_i}.\text{CipherLen} = |\mathfrak{R}|$ and $\text{MAC}_{M_i}.\text{OutLen} = |\mathfrak{M}|$). We denote it

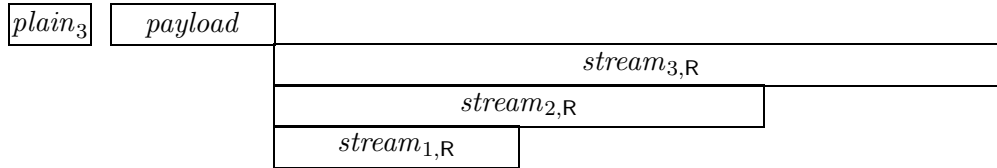
$$\text{mix_decrypt}_{M_i}(\mathfrak{R} \parallel \mathfrak{M} \parallel \mathfrak{C}).$$

Remember that M_i in general is not aware of the mix chain used by the sender or even of its own position i in the chain.

The string $plain_1$ is directed to M_1 . The remainder of the decryption result is a ciphertext that should be forwarded to the next mix in the chain, M_2 , which will then obtain the following result:



Similarly, mix M_3 will obtain the following final decryption result:



4.2 Provable Security

The mix concept is intended to provide security when at least one mix can be trusted and behaves correctly. (However note that denial-of-service attacks by incorrectly operating mixes cannot be ruled out; the only option is to avoid mixes that appear to malfunction.) Thus we assume that some single mix M_i works as expected while all other mixes are controlled by the adversary and may not follow the protocol (also the cryptographic schemes KEM_{M_j} , MAC_{M_j} , and $STREAM_{M_j}$ associated with mixes M_j , $j \neq i$, might be not secure, and KEM_{M_j} might even not be a valid key encapsulation mechanism according to the description given in section 4.1).

This leaves only M_i to be attacked: outer layers of encryption for mixes that appear before M_i in a mix chain are easily removed by the adversary and thus are not relevant for security; and inner layers of encryption for mixes that appear after M_i in a mix chain are involved in the encryption process

$$chain_encrypt_{M_i, \dots, M_n}(plain_i, \dots, plain_n; payload; C_i)$$

described in section 4.1.1, but cannot provide protection against the adversary.

An important security property for mixes is *unlinkability*: an adversary who observes a batch of encrypted messages as these are sent to a mix and who sees the resulting decrypted messages when these are forwarded elsewhere should not be able to tell better than by chance which decrypted messages correspond to which encrypted messages.

Beyond this basic security notion, we also want to achieve security against active attacks. As noted in the introduction the the current chapter, it is crucial to detect message replay and ignore replayed message. However, this does not rule out active attacks based on message modifications. To show that our construction for length-preserving mixes is secure against this kind of attacks, we will model an active adversary who is able to launch an adaptive chosen ciphertext attack (CCA).

We point out that in the model of operation that we assume, unlinkability cannot be fully guaranteed in the presence of active attacks: an active adversary who suppresses all but a single one of the legitimate encrypted messages in a batch and substitutes new messages for

them can easily trace the remaining legitimate message. This can be avoided only heuristically (each sender of mix messages should occasionally route a message to himself to see if it gets through). To avoid related flooding attacks (where an adversary injects lots of messages to cause a mix to process all messages currently in the pool), mixes should not start a new batch whenever a certain fixed number of messages has arrived; instead, each batch should be kept open for additional messages during some time frame.

Sections 4.2.1 and 4.2.2 show how the security of the mix construction can be captured formally by describing unlinkability and CCA security, respectively. We have already seen security definitions for the underlying key encapsulation mechanism (section 3.4.1), one-time message authentication code (section 3.4.2), and pseudo-random bit string generator (section 3.4.3). Section 4.2.3 presents security results that relate the security of the mix encryption scheme to the security of these primitives: we will see that if the mix encryption scheme is not secure, then this is because one of the underlying cryptographic schemes is not secure.

4.2.1 Unlinkability of the Mix Encryption Scheme

An intuitive approach to defining unlinkability is as follows: the adversary sees a batch of m ciphertexts and a random permutation of the resulting m decryption results; for each decryption result, the adversary makes a guess on the index of the corresponding ciphertext. With random guessing, the expected number of correct guesses is one. An adversary who can do better than this breaks unlinkability.

More specifically, we let the adversary select the plaintexts as far as possible (remember that as we have length-expanding decryption, it is not possible to choose all of the decryption result arbitrarily when generating a ciphertext). In the formal definition, we confine to a setting with two plaintexts where only one ciphertext is shown to the adversary (an adversary in the unrestricted setting can be used to build an adversary in this setting). This is captured in the following attack game:

1. The adversary queries a *key generation oracle*, which uses $\text{KEM}_{M_i}.\text{KeyGen}$ to compute a key pair

$$(\text{PK}, \text{SK})$$

and responds with PK (and secretly stores SK).

2. The adversary uses an *encryption oracle* as follows (cf. the encryption algorithm from section 4.1.1 in the case of a length-1 mix chain, i.e. with no recursion and $|C_i| = 0$):

- The adversary submits a pair of string pairs

$$((\text{plain}_{i,0}, m_0), (\text{plain}_{i,1}, m_1))$$

where

$$|\text{plain}_{i,b}| = \text{PlainLen}_{M_i}$$

and

$$|m_b| = \ell - \text{KEM}_{M_i}.\text{CipherLen} - \text{MAC}_{M_i}.\text{OutLen} - \text{PlainLen}_{M_i}$$

for $b = 0, 1$.

- The encryption oracle chooses a bit $b \in \{0, 1\}$ uniformly at random. Then it uses $\text{KEM}_{M_i}.\text{Encrypt}(\text{PK})$ to generate a pair $(K_{\text{oracle}}, \mathfrak{K}_{\text{oracle}})$ and splits K_{oracle} in the form

$$K_{\text{oracle}} = K_{\text{MAC}} \parallel K_{\text{STREAM}}$$

such that $|K_{\text{MAC}}| = \text{MAC}_{M_i}.\text{KeyLen}$; it computes $\text{STREAM}_{M_i}(K_{\text{STREAM}})$ and splits the resulting string $stream$ in the form

$$stream = stream_L \parallel stream_R$$

such that $|stream_L| = \ell - \text{KEM}_{M_i}.\text{CipherLen} - \text{MAC}_{M_i}.\text{OutLen}$; and it computes

$$\mathfrak{C} = stream_L \oplus (plain_{i,b} \parallel m_b)$$

and

$$\mathfrak{M} = \text{MAC}_{M_i}(K_{\text{MAC}}, \mathfrak{C}).$$

Then it outputs the pair

$$(\mathfrak{K}_{\text{oracle}} \parallel \mathfrak{M} \parallel \mathfrak{C}, stream_R).$$

3. The adversary outputs a bit $\tilde{b} \in \{0, 1\}$.

The bit \tilde{b} output by the adversary is supposed to be its guess for the value of b . Note that the decryption result corresponding to the ciphertext $\mathfrak{K}_{\text{oracle}} \parallel \mathfrak{M} \parallel \mathfrak{C}$ is

$$(plain_{i,b}, m_b \parallel stream_R)$$

(see section 4.1.2), and all of this except for the choice of the bit b is known to the adversary in the above game.

Let A be any adversary (interactive probabilistic algorithm with bounded running time) in this attack game. Its *advantage against unlinkability* for M_i 's instantiation of the mix encryption scheme is

$$\begin{aligned} \text{AdvLink}_{M_i, A} &= \left| \Pr[\tilde{b} = 1 \mid b = 1] - \Pr[\tilde{b} = 1 \mid b = 0] \right| \\ &= 2 \cdot \left| \Pr[\tilde{b} = b] - \frac{1}{2} \right|. \end{aligned}$$

4.2.2 CCA Security of the Mix Encryption Scheme

Due to the recursive nature of our encryption algorithm for mix chains, we cannot directly apply the usual definitions of security under adaptive chosen ciphertext attack (CCA) for ordinary public-key encryption as presented in section 3.1. We adapt the attack game described there as follows to take into account the special properties of our construction:

1. The adversary queries a *key generation oracle*, which uses $\text{KEM}_{M_i}.\text{KeyGen}$ to compute a key pair

$$(\text{PK}, \text{SK})$$

and responds with PK (and secretly stores SK).

2. [**Find stage.**] The adversary makes a sequence of queries to a *decryption oracle*. Each query is an arbitrary string s of length ℓ , and the oracle responds with

$$\text{mix_decrypt}_{M_i}(s),$$

using the secret key SK from stage 1. That is, each oracle response is either a pair (plain, P) where $|\text{plain}| = \text{PlainLen}_{M_i}$ and $|P| = \ell$, or the special value *invalid*.

3. The adversary uses an *interactive encryption oracle* as follows (compare with the encryption algorithm in section 4.1.1):

- First the adversary submits some string C_i subject only to the condition that

$$0 \leq |C_i| < \ell - \text{KEM}_{M_i}.\text{CipherLen} - \text{MAC}_{M_i}.\text{OutLen}.$$

- The interactive encryption oracle uses $\text{KEM}_{M_i}.\text{Encrypt}(\text{PK})$ to generate a pair $(K_{\text{oracle}}, \mathfrak{K}_{\text{oracle}})$ and splits K_{oracle} in the form

$$K_{\text{oracle}} = K_{\text{MAC}} \parallel K_{\text{STREAM}}$$

such that $|K_{\text{MAC}}| = \text{MAC}_{M_i}.\text{KeyLen}$; it computes $\text{STREAM}_{M_i}(K_{\text{STREAM}})$ and splits the resulting string *stream* in the form

$$\text{stream} = \text{stream}_L \parallel \text{stream}_R$$

such that $|\text{stream}_L| = \ell - |C_i| - \text{KEM}_{M_i}.\text{CipherLen} - \text{MAC}_{M_i}.\text{OutLen}$; and it computes

$$C_{i+1} = \text{stream}_R \oplus (C_i \parallel 0^{\text{KEM}_{M_i}.\text{CipherLen} + \text{MAC}_{M_i}.\text{OutLen} + \text{PlainLen}_{M_i}})$$

and sends C_{i+1} to the adversary.

- The adversary submits a pair of string pairs

$$((\text{plain}_{i,0}, m_0), (\text{plain}_{i,1}, m_1))$$

satisfying

$$|\text{plain}_{i,b}| = \text{PlainLen}_{M_i}$$

and

$$|m_b| = \ell - |C_i| - \text{KEM}_{M_i}.\text{CipherLen} - \text{MAC}_{M_i}.\text{OutLen} - \text{PlainLen}_{M_i}$$

for $b = 0, 1$.

- The interactive encryption oracle chooses a uniformly random bit $b \in \{0, 1\}$, determines

$$\mathfrak{C} = (\text{stream}_L \oplus (\text{plain}_{i,b} \parallel m_b)) \parallel C_i$$

and

$$\mathfrak{M} = \text{MAC}_{M_i}(K_{\text{MAC}}, \mathfrak{C}),$$

and responds with the *challenge ciphertext*

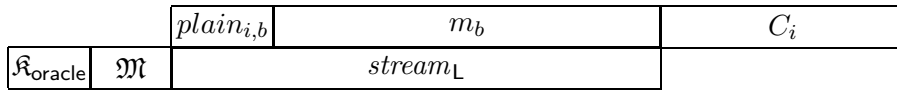
$$\mathfrak{K}_{\text{oracle}} \parallel \mathfrak{M} \parallel \mathfrak{C}.$$

4. [**Guess stage.**] The adversary again makes a sequence of queries to a *decryption oracle* as in stage 2, where this time the decryption oracle refuses being asked for the challenge ciphertext $\mathfrak{K}_{\text{oracle}} \parallel \mathfrak{M} \parallel \mathfrak{C}$ from stage 3 (it returns *invalid* for this case).
5. The adversary outputs a bit $\tilde{b} \in \{0, 1\}$.

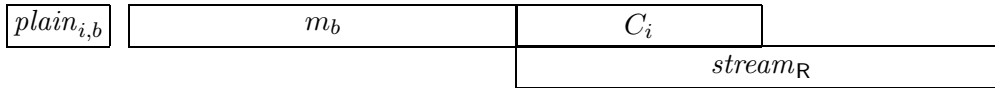
The bit \tilde{b} output by the adversary is its guess for the value of b .

The essential difference to the adaptive chosen ciphertext attack game for ordinary public-key encryption is that we have made the encryption oracle interactive to reflect the recursiveness of the encryption algorithm for mix chains, where encryption to mixes later in the chain than M_i can have potentially arbitrary effects on a large part of the plaintext.

We look at illustrations for the ciphertext resulting from the invocation of the interactive encryption oracle. As in previous illustrations, we have concatenation horizontally and XOR vertically. The structure of the challenge ciphertext returned by the interactive encryption oracle in the adaptive chosen ciphertext attack game is as follows:



Applying algorithm $mix_decrypt_{M_i}$ from section 4.1.2 to this ciphertext would yield the following result:



This can also be written as follows:



Let A be any adversary (interactive probabilistic algorithm with bounded running time) in the above attack game. Its *CCA advantage* against M_i 's instantiation of the mix encryption scheme is

$$\begin{aligned} \text{AdvCCA}_{M_i,A} &= \left| \Pr[\tilde{b} = 1 \mid b = 1] - \Pr[\tilde{b} = 1 \mid b = 0] \right| \\ &= 2 \cdot \left| \Pr[\tilde{b} = b] - \frac{1}{2} \right|. \end{aligned}$$

4.2.3 Security Results

First let A be an adversary against the unlinkability of the mix encryption scheme, attacking a mix M_i as described in section 4.2.1. It can be shown that there are adversaries A_1 and A_2 against KEM_{M_i} and STREAM_{M_i} , respectively, with essentially the same running time as A such that

$$\text{AdvLink}_{M_i,A} \leq 2 \cdot (\text{AdvCCA}_{\text{KEM}_{M_i},A_1} + \text{Adv}_{\text{STREAM}_{M_i},A_2}).$$

Details of the proof of unlinkability follow below.

Now let A be an adversary against the CCA security of the mix encryption scheme, attacking a mix M_i as described in section 4.2.2. It can be shown that there are adversaries A_1, A_2, A_3 against $\text{KEM}_{M_i}, \text{MAC}_{M_i}, \text{STREAM}_{M_i}$ all having essentially the same running time as A such that

$$\text{AdvCCA}_{M_i,A} \leq 2 \cdot (\text{AdvCCA}_{\text{KEM}_{M_i},A_1} + \text{AdvForge}_{\text{MAC}_{M_i},A_2} + \text{Adv}_{\text{STREAM}_{M_i},A_3}).$$

Details of the proof of CCA security follow below.

Proof of Unlinkability

We prove the security result for the unlinkability of the mix encryption scheme. Let \mathbf{G}_0 denote the attack game from section 4.2.1. Let \mathbf{G}_1 be like \mathbf{G}_0 except that a uniformly random string is used for K_{oracle} , whereas $\mathfrak{K}_{\text{oracle}}$ is still generated by $\text{KEM}_{M_i}.\text{Encrypt}(\text{PK})$. Let \mathbf{G}_2 be like \mathbf{G}_1 except that the encryption oracle uses a uniformly random string $stream$ instead of computing $\text{STREAM}_{M_i}(K_{\text{STREAM}})$.

Now we consider an adversary A as in section 4.2.1, exposed to these different attack games \mathbf{G}_x , $0 \leq x \leq 2$, and look at the respective success probabilities $\Pr_{\mathbf{G}_x}[\tilde{b} = b]$. Based on A , adversaries A_1 and A_2 against KEM_{M_i} and STREAM_{M_i} , respectively, will be built, each with essentially the same running time as A .

A_1 attacks KEM_{M_i} as follows (cf. section 3.4.1; note that this A_1 never actually uses its decryption oracle). At first, it generates $b \in \{0, 1\}$ uniformly at random. Then, it runs the adversary A ; when A queries its encryption oracle, A_1 queries its key encapsulation oracle to obtain a pair $(K_{\text{oracle}}, \mathfrak{K}_{\text{oracle}})$ and performs stage 2 from section 4.2.1 using this pair and the pregenerated bit b . Finally, when A outputs its bit \tilde{b} , A_1 outputs 1 if $\tilde{b} = b$ and 0 otherwise. Observe that

$$\left| \Pr_{\mathbf{G}_1}[\tilde{b} = b] - \Pr_{\mathbf{G}_0}[\tilde{b} = b] \right| = \text{AdvCCA}_{\text{KEM}_{M_i}, A_1}$$

(\mathbf{G}_0 corresponds to $b_{\text{KEM}} = 0$, \mathbf{G}_1 corresponds to $b_{\text{KEM}} = 1$ in section 3.4.1).

A_2 attacks STREAM_{M_i} as follows (cf. section 3.4.3). First, it generates $b \in \{0, 1\}$ and a string K_{oracle} of length $\text{KEM}_{M_i}.\text{OutLen}$ uniformly at random. Then, it runs the adversary A , playing the role of the key generation oracle and the role of the encryption oracle, wherein it substitutes the pregenerated string K_{oracle} and the pregenerated bit b in stage 2 of section 4.2.1. When A outputs its bit \tilde{b} , A_2 outputs 1 if $\tilde{b} = b$ and 0 otherwise. Observe that

$$\left| \Pr_{\mathbf{G}_2}[\tilde{b} = b] - \Pr_{\mathbf{G}_1}[\tilde{b} = b] \right| = \text{AdvSTREAM}_{M_i, A_2}$$

(\mathbf{G}_1 corresponds to $b_{\text{STREAM}} = 0$, \mathbf{G}_2 corresponds to $b_{\text{STREAM}} = 1$ in section 3.4.3).

Finally observe that $\Pr_{\mathbf{G}_2}[\tilde{b} = b] = \frac{1}{2}$.

From this we obtain the inequality

$$\begin{aligned} \text{AdvLink}_{M_i, A} &= 2 \cdot \left| \frac{1}{2} - \Pr_{\mathbf{G}_0}[\tilde{b} = b] \right| \\ &= 2 \cdot \left| \sum_{1 \leq x \leq 2} \left(\Pr_{\mathbf{G}_x}[\tilde{b} = b] - \Pr_{\mathbf{G}_{x-1}}[\tilde{b} = b] \right) \right| \\ &\leq 2 \cdot (\text{AdvCCA}_{\text{KEM}_{M_i}, A_1} + \text{AdvSTREAM}_{M_i, A_2}), \end{aligned}$$

which concludes the proof.

Proof of CCA Security

We prove the CCA security result given above. (This proof is similar to the security proof for DHAES in section 3.4.4.) Let \mathbf{G}_0 denote the attack game from section 4.2.2. We will modify it in multiple steps, essentially disabling the underlying cryptographic schemes (key

encapsulation mechanism, one-time message authentication code, and pseudo-random bit string generator) one after another.

\mathbf{G}_1 is like \mathbf{G}_0 except that a uniformly random string is used for K_{oracle} , whereas $\mathfrak{K}_{\text{oracle}}$ is still generated by $\text{KEM}_{M_i}.\text{Encrypt}(\text{PK})$. In the decryption oracle, K_{oracle} is substituted whenever $\text{KEM}_{M_i}.\text{Decrypt}(\text{SK}, \mathfrak{K}_{\text{oracle}})$ would have to be computed. This applies to both the find and the guess stage, i.e. stages 2 and 4 in the attack game from section 4.2.2. Thus, the invocation of $\text{KEM}_{M_i}.\text{Encrypt}(\text{PK})$ to generate $\mathfrak{K}_{\text{oracle}}$ must be advanced from stage 3 to an earlier stage. In \mathbf{G}_0 , this is only a descriptive change and does not affect the behavior observed by the adversary.

\mathbf{G}_2 is like \mathbf{G}_1 except that the decryption oracle always responds with invalid when faced with any query prefixed with string $\mathfrak{K}_{\text{oracle}}$.

\mathbf{G}_3 is like \mathbf{G}_2 except that the interactive encryption oracle uses a uniformly random string *stream* instead of computing $\text{STREAM}_{M_i}(K_{\text{STREAM}})$.

Now we consider an adversary A as in section 4.2.2, exposed to these different attack games \mathbf{G}_x , $0 \leq x \leq 3$, and look at the respective success probabilities $\Pr_{\mathbf{G}_x}[\tilde{b} = b]$. Based on A , adversaries A_1, A_2, A_3 against $\text{KEM}_{M_i}, \text{MAC}_{M_i}, \text{STREAM}_{M_i}$ will be built all having essentially the same running time as A .

A_1 attacks KEM_{M_i} as follows (cf. section 3.4.1). At first, it generates $b \in \{0, 1\}$ uniformly at random and queries its key encapsulation oracle to obtain a pair $(K_{\text{oracle}}, \mathfrak{K}_{\text{oracle}})$. Then, it runs the adversary A , playing the roles of the decryption oracle and encryption oracle: when A queries its decryption oracle, A_1 uses its own decryption oracle to compute $\text{KEM}.\text{Decrypt}(\text{SK}, \mathfrak{K})$ when performing the decryption algorithm from section 4.1.2, substituting K_{oracle} when $\text{KEM}.\text{Decrypt}(\text{SK}, \mathfrak{K}_{\text{oracle}})$ would have to be computed; when A queries its encryption oracle, A_1 performs the encryption oracle stage using the pregenerated pair $(K_{\text{oracle}}, \mathfrak{K}_{\text{oracle}})$ and the pregenerated bit b . Finally, when A outputs its bit \tilde{b} , A_1 outputs 1 if $\tilde{b} = b$ and 0 otherwise. Observe that

$$\left| \Pr_{\mathbf{G}_1}[\tilde{b} = b] - \Pr_{\mathbf{G}_0}[\tilde{b} = b] \right| = \text{AdvCCA}_{\text{KEM}_{M_i}, A_1}$$

(\mathbf{G}_0 corresponds to $b_{\text{KEM}} = 0$, \mathbf{G}_1 corresponds to $b_{\text{KEM}} = 1$ in section 3.4.1).

A_2 attacks MAC_{M_i} as follows (cf. section 3.4.2). At first, it generates $b \in \{0, 1\}$ and a string K_{STREAM} of length $\text{STREAM}_{M_i}.\text{KeyLen}$ uniformly at random and uses $\text{KEM}_{M_i}.\text{Encrypt}(\text{PK})$ to generate $\mathfrak{K}_{\text{oracle}}$. Then, it runs the adversary A , playing the roles of the key generation oracle (so that A_2 knows SK), decryption oracle, and interactive encryption oracle. Whenever A submits any query $\mathfrak{R} \parallel \mathfrak{M} \parallel \mathfrak{C}$ to the decryption oracle, A_2 adds the pair $(\mathfrak{C}, \mathfrak{M})$ to its own output; queries prefixed with $\mathfrak{K}_{\text{oracle}}$ are refused, all other questions are answered using SK . In stage 3 of section 4.2.2 (encryption oracle), A_2 substitutes the pregenerated string K_{STREAM} and the pregenerated bit b , and it invokes its MAC oracle on \mathfrak{C} instead of using MAC_{M_i} directly to obtain \mathfrak{M} for the challenge ciphertext. A 's final output bit \tilde{b} is ignored. Observe that

$$\left| \Pr_{\mathbf{G}_2}[\tilde{b} = b] - \Pr_{\mathbf{G}_1}[\tilde{b} = b] \right| \leq \text{AdvForge}_{\text{MAC}_{M_i}, A_2}$$

(A_2 runs A as in game \mathbf{G}_2 , and A would observe different behavior in game \mathbf{G}_1 only in the cases where a forgery is produced).

A_3 attacks STREAM_{M_i} as follows (cf. section 3.4.3). First, it generates $b \in \{0, 1\}$ and a string K_{oracle} of length $\text{KEM}_{M_i}.\text{OutLen}$ uniformly at random. Then, it runs the adversary A , playing the roles of the key generation oracle, interactive encryption oracle, and decryption

oracle, substituting the pregenerated string K_{oracle} and the pregenerated bit b in stage 3 of section 4.2.2 and refusing any decryption oracle query prefixed with $\tilde{\mathfrak{K}}_{\text{oracle}}$ (both in the find stage and in the guess stage). When A outputs its bit \tilde{b} , A_3 outputs 1 if $\tilde{b} = b$ and 0 otherwise. Observe that

$$\left| \Pr_{\mathbf{G}_3}[\tilde{b} = b] - \Pr_{\mathbf{G}_2}[\tilde{b} = b] \right| = \text{Adv}_{\text{STREAM}_{M_i, A_3}}$$

(\mathbf{G}_2 corresponds to $b_{\text{STREAM}} = 0$, \mathbf{G}_3 corresponds to $b_{\text{STREAM}} = 1$ in section 3.4.3).

Finally observe that $\Pr_{\mathbf{G}_3}[\tilde{b} = b] = \frac{1}{2}$.

From this we obtain the inequality

$$\begin{aligned} \text{Adv}_{\text{CCA}_{M_i, A}} &= 2 \cdot \left| \frac{1}{2} - \Pr_{\mathbf{G}_0}[\tilde{b} = b] \right| \\ &= 2 \cdot \left| \sum_{1 \leq x \leq 3} \left(\Pr_{\mathbf{G}_x}[\tilde{b} = b] - \Pr_{\mathbf{G}_{x-1}}[\tilde{b} = b] \right) \right| \\ &\leq 2 \cdot (\text{Adv}_{\text{CCA}_{\text{KEM}_{M_i, A_1}}} + \text{Adv}_{\text{Forge}_{\text{MAC}_{M_i, A_2}}} + \text{Adv}_{\text{STREAM}_{M_i, A_3}}), \end{aligned}$$

which concludes the proof.

Part II
Practice

Chapter 5

Exponentiation and Multi-Exponentiation in Public-Key Cryptography

Many schemes in public-key cryptography require computing powers

$$g^e$$

(*exponentiation*) or power products

$$\prod_{1 \leq j \leq k} g_j^{e_j}$$

(*multi-exponentiation*) in a commutative semigroup G with neutral element 1_G , e.g. in the group $(\mathbb{Z}/n\mathbb{Z})^*$ or more generally in the multiplicative semigroup $(\mathbb{Z}/n\mathbb{Z})$ for some integer n , in the group of rational points on an elliptic curve over a finite field [12], or in the class group of an imaginary-quadratic order [40]. The exponents e , e_j are non-negative integers with a typical length of a few hundred or a few thousand bits.

A specific example for the use of exponentiation that we have seen is Diffie-Hellman (protocols 2.2, 2.3 and 2.4), which can be used as an essential primitive in more complex cryptographic schemes such as DHAES (see chapter 3) or the mix encryption scheme from chapter 4. Another well-known example is RSA [76], which can be used both for public-key encryption and for digital signatures. Multi-exponentiation with $k = 2$ is used e.g. for verification of DSA signatures [66] in groups $(\mathbb{Z}/n\mathbb{Z})^*$, and for verification of ECDSA signatures [4] in groups of rational points on elliptic curves over finite fields. Multi-exponentiation with $k = 3$ is used e.g. for verification of ElGamal signatures [35] in groups $(\mathbb{Z}/n\mathbb{Z})^*$. Larger values of k appear in protocols of Brands [16]. For $k \geq 2$, in general it is unnecessarily inefficient to compute the powers $g_i^{e_i}$ separately and then multiply them. Instead, specific algorithms for multi-exponentiation can be applied.

Bases $g, g_j \in G$ sometimes are fixed between many computations. With fixed bases, it is often advantageous to perform a single time a possibly relatively expensive *precomputation* in order to prepare a table that can be used to speed up exponentiations involving those bases. (For multi-exponentiation, some of the bases may be fixed while others are variable: for example, verifying a DSA [66] or ECDSA [4] signature involves computing the product of two powers where one of the bases is part of *domain parameters* that can be shared between a large number of signers while the other base is specific to a single signer.)

We assume that the exponents consist of independent random bits up to a respective maximum bit-length; i.e., e and each e_i is a uniformly distributed random integer in some interval $[0, 2^L - 1]$. In practice the actual distribution may differ, but for typical cases this simplified assumption is reasonably close. In this setting, we consider general algorithms for arbitrary exponents; we do not examine algorithms based on tailor-made addition chains in \mathbb{Z}^k for given e or e_1, \dots, e_k (cf. [14]).

In the following chapters, we look at efficient algorithms for exponentiation and multi-exponentiation based on either just multiplication in the given semigroup or optionally, in the case of a group, on multiplication and division. This amounts to constructing addition chains or addition-subtraction chains for the exponent e for exponentiation, and to constructing vector addition chains or vector addition-subtraction chains for the vector of exponents (e_1, \dots, e_k) for multi-exponentiation (see e.g. the survey [38]).

For purposes of performance analysis, we distinguish between squarings and general multiplications, as the former can often be implemented more efficiently. If we allow division, our performance analysis does not distinguish between divisions and multiplications; this is reasonable e.g. for point groups of elliptic curves and for class groups of imaginary-quadratic number fields, as inversion is almost immediate in such groups. If inversion is expensive (e.g. in $(\mathbb{Z}/n\mathbb{Z})^*$), the group should be treated as a semigroup, i.e. inversion should be avoided.

Chapter 6 considers the case of a single exponentiation. Both left-to-right exponentiation and right-to-left exponentiation methods are described. The sliding window and window NAF technique and, as an improvement to the state of the art, the fractional window technique are presented. The fractional window technique is a generalization of the sliding window and window NAF approach; it can be used to improve performance in devices with limited storage.

Chapter 7 presents different approaches for computing power products. It looks at the conventional simultaneous exponentiation approach and presents an alternative strategy, interleaved exponentiation. The comparison shows that in general groups, sometimes the conventional method and sometimes interleaved exponentiation is more efficient. In groups where inverting elements is easy (e.g. elliptic curves), the window-NAF based interleaved exponentiation method usually wins over the conventional method. Exponentiation and multi-exponentiation with precomputation is also considered in that chapter.

Finally, chapter 8 considers specific techniques that can be used in elliptic curve cryptography if side-channel attacks are a concern. Following the convention to treat point groups as additive rather than multiplicative, we speak of point multiplication (by a scalar) instead of exponentiation. Two 2^w -ary methods are presented that implement point multiplication in a fixed sequence of basic point operations to avoid exposing specific information on scalars: a left-to-right method based on special representations of scalars, and a right-to-left method employing a special initialization stage.

Notational Conventions

We write $e[j]$ for bit j of a non-negative integer e ; thus $e = \sum_j e[j]2^j$. For negative j , we define that $e[j] = 0$. We write $e[j' \dots j]$ for the integer consisting of the concatenation of bits j' down to j of e ; e.g., if

$$e = 10111_2 = 23,$$

then

$$e[3 \dots 1] = 011_2 = 3$$

and

$$e[1 \dots -2] = 1100_2 = 12.$$

$\text{LSB}_m(e) = e[(m-1) \dots 0] = e \bmod 2^m$ is the integer formed by the m least significant bits of e , and $\text{LSB}(e) = \text{LSB}_1(e)$.

When writing digits, we use the convention that \bar{b} denotes a digit of value $-b$ where b is understood to be a positive integer; for example, $10\bar{1}_2 = 2^2 - 2^0 = 3$.

Chapter 6

Efficient Exponentiation

In this chapter, we look at the basic case of exponentiation, computing g^e . We will examine methods for multi-exponentiation afterwards in chapter 7. Special strategies for exponentiation with precomputation where the base g is fixed for multiple exponentiations will also be presented there (in section 7.4) because an important basic approach for exponentiation with precomputation essentially turns exponentiations into multi-exponentiations.

Section 6.1 gives a framework for exponentiation algorithms. Section 6.2 describes within the framework the sliding window exponentiation method and the window NAF exponentiation method. Section 6.3 presents an improvement to the state of the art, fractional windows, a technique that closes a gap in the sliding window and window NAF methods and is useful for devices with limited storage: it can improve the performance by making use of memory that would have to remain unused with the previously known methods. Then section 6.4 discusses how the exponent representations employed by this technique can be implemented with small memory overhead.

6.1 A Framework for Exponentiation

Remember that we are interested in algorithms that compute g^e using just multiplication in the semigroup at hand, and optionally also division in the case of a group. (In specific groups, additional useful efficiently computable endomorphisms may be available besides squaring and possibly inversion; see e.g. [82]. This may lead to better exponentiation algorithms for these groups. We will not consider the special techniques for such groups.)

Many algorithms for computing g^e for arbitrary large positive integers e in this setting fit into one of two variants of a common framework, which we describe in this section. Exponents e are represented in base 2 as

$$e = \sum_{0 \leq i \leq \ell} b_i \cdot 2^i,$$

using digits $b_i \in B \cup \{0\}$ where B is some set of integers with $1 \in B$. We call this a B -representation of e . Details of it are intrinsic to the specific exponentiation method. (Note that for given B , B -representations are usually not canonical.) The elements of B must be non-negative unless G is a group where inversion is possible in reasonable time. Given a B -representation, *left-to-right* or *right-to-left* methods can be used for exponentiation. Left-to-right methods look at the elements b_i starting at b_ℓ and proceed down to b_0 ; right-to-left

methods start at b_0 and proceed up to b_ℓ . Depending on how the values b_i can be obtained from an input value e , it may be easy to compute them on the fly instead of storing the B -representation beforehand. Left-to-right methods and right-to-left methods can be considered dual to each other (cf. the duality observation for representations of arbitrary addition chains as directed multi-graphs in [47, p. 481]); both involve two stages.

6.1.1 Left-to-Right Methods

For *left-to-right* methods, first, in the *precomputation stage*, powers g^b for all $b \in B$ are computed and stored; if division in G is permissible (because inversion is fast) and $|b| \in B$ for each $b \in B$, it suffices to precompute g^b for those $b \in B$ that are positive. We refer to this collection of precomputed powers g^b as the *precomputed table*. How to implement the precomputation stage efficiently depends on the specific choice of B . In certain semigroups, in order to accelerate the evaluation stage, precomputed elements can be represented in a special way such that multiplications with these elements take less time (for example, precomputed points on an elliptic curve may be converted from projective into affine representation [27]). Note that if both the base element g and the digit set B are fixed, the precomputation stage need not be repeated for multiple exponentiations if the precomputed table is kept in memory. In cases without such fixed precomputation, B is usually a set consisting of small integers such that the precomputation stage requires only a moderate amount of time. If

$$B = \{1, 3, \dots, \beta\} \quad \text{or} \quad B = \{\pm 1, \pm 3, \dots, \pm \beta\}$$

with $\beta \geq 3$ odd, the precomputation stage can be implemented with one squaring and $(\beta-1)/2$ multiplications as follows: first compute g^2 ; then iteratively compute $g^3 = g \cdot g^2$, \dots , $g^\beta = g^{\beta-2} \cdot g^2$. This applies to all encoding techniques that will be presented in this chapter.

In the *evaluation stage* (or *left-to-right stage*) of a left-to-right method, given the precomputed table and the representation of e as digits b_i , the following algorithm is executed to compute the desired power from the precomputed elements g^b :

```

A ← 1_G
for i = ℓ down to 0 do
  A ← A²
  if b_i ≠ 0 then
    A ← A · g^{b_i}
return A

```

If division is permissible, the following modified algorithm can be used:

```

A ← 1_G
for i = ℓ down to 0 do
  A ← A²
  if b_i ≠ 0 then
    if b_i > 0 then
      A ← A · g^{b_i}
    else
      A ← A / g^{|b_i|}
return A

```

Note that in these algorithms squarings can be omitted while A is 1_G ; similarly, the first multiplication or division can be replaced by an assignment or an assignment followed by inversion of A .

6.1.2 Right-to-Left Methods

For *right-to-left* methods, no precomputed elements are used. Instead, first the *right-to-left stage* yields values in a number of accumulators A_b , one for each positive element $b \in B$. If division is permissible, B may contain negative digits; we require that $|b| \in B$ for each $b \in B$. Second, the *result stage* combines the accumulator values to obtain the final result. The following algorithm description comprises both stages, but the result stage is condensed into just the “return” line: how to implement it efficiently depends on the specific choice of B . For brevity, we show just the algorithm with division (if B does not contain negative digits, the “else”-branch will never be taken and can be left out).

```

{right-to-left stage}
for  $b \in B$  do
  if  $b > 0$  then
     $A_b \leftarrow 1_G$ 
 $A \leftarrow g$ 
for  $i = 0$  to  $\ell$  do
  if  $b_i \neq 0$  then
    if  $b_i > 0$  then
       $A_{b_i} \leftarrow A_{b_i} \cdot A$ 
    else
       $A_{|b_i|} \leftarrow A_{|b_i|} / A$ 
     $A \leftarrow A^2$ 

{result stage}
return  $\prod_{\substack{b \in B \\ b > 0}} A_b^b$ 

```

The squaring operation may be omitted in the final iteration as the resulting value of A will not be used. For each A_b , the first multiplication or division can be replaced by an assignment or an assignment followed by inversion (implementations can use flags to keep track which of the A_b still contain the value 1_G). In case some A_b are still 1_G at the end of the right-to-left stage, operations can be saved in the result stage.

If

$$B = \{1, 3, \dots, \beta\} \quad \text{or} \quad B = \{\pm 1, \pm 3, \dots, \pm \beta\}$$

with β odd (as in all encoding techniques that we will consider in this chapter), the result stage can be implemented as follows ([90], [47, exercise 4.6.3-9]):

```

for  $b = \beta$  to 3 step  $-2$  do
   $A_{b-3} \leftarrow A_{b-3} \cdot A_b$ 
   $A_1 \leftarrow A_1 \cdot A_b^2$ 
return  $A_1$ 

```

This result stage algorithm requires $(\beta - 1)/2$ squarings and $\beta - 1$ multiplications.

6.2 Sliding Window Exponentiation and Window NAF Exponentiation

A well-known method for exponentiation in semigroups is the *sliding window* technique (cf. [84, p. 912] and [38, section 3]). The encoding is parameterized by a small positive integer w , the *window size*. The digit set is $B = \{1, 3, \dots, 2^w - 1\}$. Encodings using these digits can be computed on the fly by scanning the ordinary binary representation of the exponent either in left-to-right or in right-to-left direction: in the respective direction, repeatedly look out for the first non-zero bit and then examine the sequence of w bits starting at this bit position; one of the odd digits in B suffices to cover these w bits. For example, given $e = 88 = 1011000_2$, for window size $w = 3$, left-to-right scanning yields

$$\underline{101} \underline{1000}_2 \rightarrow 51000_2,$$

and right-to-left scanning yields

$$\underline{\underline{1011}} \underline{000}_2 \rightarrow 1003000_2.$$

The length of the resulting representation

$$e = \sum_{0 \leq i \leq \ell} b_i \cdot 2^i$$

is at most that of the binary representation, i.e. a maximum index ℓ suffices to represent any $(\ell + 1)$ -bit exponent. The average density of non-zero digits is $1/(w + 1)$ for $e \rightarrow \infty$. (A more precise analysis shows that the average number of non-zero digits in the representation for $(\ell + 1)$ -bit exponents is

$$\frac{\ell + 1}{w + 1} + 1 - \frac{w(w + 3)}{2(w + 1)^2} + O(\varrho^{-\ell})$$

for a real number $\varrho > 1$ [26, Proposition 2.12]).

Including negative digits into B allows decreasing the average density: a $\{\pm 1\}$ -representation such that no two adjacent digits are non-zero (“property M” from [75]) is called a *non-adjacent form* or *NAF*. More generally, let

$$B = \{\pm 1, \pm 3, \dots, \pm(2^w - 1)\};$$

then the following algorithm (from [82]) generates a B -representation of e such that at most one of any $w + 1$ consecutive digits is non-zero. There is a unique representation with this property, the *width- $(w + 1)$ NAF* of e . We use the term *window NAF* (wNAF) if w is understood (a width-2 NAF can simply be called a NAF). This idea is also known as the *signed window* approach; $w + 1$ can be considered the window size.

```

c ← e
i ← 0
while c > 0 do
  if LSB(c) = 1 then
    b ← LSBw+1(c)
    if b ≥ 2w then
      b ← L - 2w+1

```

```

    c ← c - b
  else
    b ← 0
    bi ← b; i ← i + 1
    c ← c/2
  return bi-1, ..., b0

```

Compared with the binary representation, the length can grow at most by one digit, so a maximum index ℓ is sufficient to represent any ℓ -bit exponent as a window NAF. Width- $(w + 1)$ NAFs have an average density of $1/(w + 2)$ for $e \rightarrow \infty$ ([78], [81], [57], [82]). (Here a more precise analysis shows that the average number of non-zero digits in the representation for ℓ -bit exponents is

$$\frac{\ell}{w + 2} + 1 - \frac{w(w + 3)}{2(w + 2)^2} + O(\varrho^{-\ell})$$

for a real number $\varrho > 1$ [26, Proposition 2.6]).

For left-to-right exponentiation using the sliding window or window NAF technique, the precomputation stage has to compute g^b for $b \in \{1, 3, \dots, 2^w - 1\}$, which for $w > 1$ can be achieved with one squaring and $2^{w-1} - 1$ multiplications (see section 6.1.1).

For right-to-left exponentiation using the sliding window or window NAF technique, the result stage has to compute

$$\prod_{b \in \{1, 3, \dots, 2^w - 1\}} A_b^b$$

given accumulator values A_b resulting from the right-to-left stage. This can be done in $2^{w-1} - 1$ squarings and $2^w - 2$ multiplications (see section 6.1.2).

6.2.1 Modified Window NAFs

The efficiency of exponentiation given a B -representation depends on the number of non-zero digits and the length of the representation (i.e. the minimum index I such that $b_i = 0$ for $i \geq I$). Window NAFs may have increased length compared with the ordinary binary representation: e.g., the (width-2) NAF for $3 = 11_2$ is $10\bar{1}_2$, and the NAF for $7 = 111_2$ is $100\bar{1}_2$. (Remember that \bar{b} denotes a digit of value $-b$.)

Such length expansion can easily be avoided in about half of the cases and thus exponentiation made more efficient by weakening the non-adjacency property (cf. [15] for the case of width-2 NAFs). A *modified window NAF* (more precisely, a *modified width- $(w + 1)$ NAF*) is a B -representation obtained from a width- $(w + 1)$ NAF as follows: if the $w + 2$ most significant digits (ignoring any leading zeros) have the form

$$1 \underbrace{00 \dots 0}_w \bar{b},$$

then substitute

$$01 \underbrace{0 \dots 0}_{w-1} \beta$$

where $\beta = 2^w - b$. For the above examples, we see that the modified (width-2) NAF for 3 is 11_2 , which is shorter than the NAF; however, the modified NAF for 7 remains $100\bar{1}$: this is a case where length expansion cannot be avoided without increasing the number of non-zero digits.

6.3 Fractional Windows

In small devices, the choice of w for exponentiation using the sliding window or window NAF technique described in section 6.2 may be dictated by memory limitations. The exponentiation algorithms given in section 6.1 need storage for $1 + 2^{w-1}$ elements of G , and thus memory may be wasted: e.g., if sufficient storage is available for up to four elements, only three elements can actually be used ($w = 2$).

In this section, we will see how the efficiency of exponentiation can be improved by using *fractional windows*, a generalization of the sliding window and window NAF techniques. Section 6.3.1 describes this new encoding technique for the case that negative digits are allowed (*signed fractional windows*). Section 6.3.2 describes a simpler variant for the case that only non-negative digits are permissible (*unsigned fractional windows*).

6.3.1 Signed Fractional Windows

Let $w \geq 2$ be an integer and m an odd integer such that $1 \leq m \leq 2^w - 3$. (We could additionally allow the border cases $m = -1$ and $m = 2^w - 1$, which would turn out to yield the width- $(w + 1)$ and width- $(w + 2)$ NAF, respectively.) The digit set for the *signed fractional window* representation with parameters w and m is

$$B = \{ \pm 1, \pm 3, \dots, \pm(2^w + m) \}.$$

Let the mapping

$$\text{digit}: \{0, 1, \dots, 2^{w+2}\} \rightarrow B \cup \{0\}$$

be defined as follows:

- If x is even, then $\text{digit}(x) = 0$;
- otherwise if $0 < x \leq 2^w + m$, then $\text{digit}(x) = x$;
- otherwise if $2^w + m < x < 3 \cdot 2^w - m$, then $\text{digit}(x) = x - 2^{w+1}$;
- otherwise we have $3 \cdot 2^w - m \leq x < 2^{w+2}$ and let $\text{digit}(x) = x - 2^{w+2}$.

Observe that if x is odd, then $x - \text{digit}(x) \in \{0, 2^{w+1}, 2^{w+2}\}$. The following algorithm encodes e into signed fractional window representation:

```

d ← LSBw+2(e)
c ← ⌊e/2w+2⌋
i ← 0
while d ≠ 0 ∨ c ≠ 0 do
  b ← digit(d)
  bi ← b; i ← i + 1
  d ← d - b

  d ← LSB(c) · 2w+1 + d/2
  c ← ⌊c/2⌋
return bi-1, ..., b0

```


This algorithm is a direct variant of the window NAF generation algorithm shown in section 6.2, but based on the new mapping *digit*. Here we have expressed the algorithm in a way that shows that the loop is essentially a finite state machine (with $2^{w+1} + 1$ states for storing, after b has been subtracted from the previous value of d , the even number d with $0 \leq d \leq 2^{w+2}$); new bits taken from c are considered input symbols and the generated digits b_i are considered output symbols.

The following table shows what can happen in the loop in the example case $w = 2$, $m = 1$.

d	$b = \text{digit}(d)$	$d - b$
0001 ₂	1	0000 ₂
0011 ₂	3	0000 ₂
0101 ₂	5	0000 ₂
0111 ₂	-1	1000 ₂
1001 ₂	1	1000 ₂
1011 ₂	-5	10000 ₂
1101 ₂	-3	10000 ₂
1111 ₂	-1	10000 ₂

The average density achieved by the signed fractional window representation with parameters w and m is

$$\frac{1}{w + \frac{m+1}{2^w} + 2}$$

for $e \rightarrow \infty$. (Assume that an endless sequence of random bits is the input to the finite state machine described above: whenever the state machine is about to output a non-zero digit, the intermediate value $d \bmod 2^{w+2}$ consists of $w + 1$ independent random bits plus the least significant bit, which is necessarily set. Thus with probability $p = \frac{1}{2} - \frac{m+1}{2^{w+1}}$, we have $d - \text{digit}(d) = 2^{w+1}$, and with probability $1 - p$, we have $d - \text{digit}(d) \in \{0, 2^{w+2}\}$. In the first case, the next non-zero output digit will follow after exactly w intermediate zeros; in the second case, the next non-zero output digit will follow after $w + 2$ intermediate zeros on average. Thus the total average for the number of intermediate zeros is $p \cdot w + (1 - p) \cdot (w + 2) = w + \frac{m+1}{2^w} + 1$, which yields the above expression for the density.) Comparing this with the $1/(w + 2)$ density for width- $(w + 1)$ NAFs, we see that the effective window size has been increased by $(m + 1)/2^w$, which is why we speak of “fractional windows”.

As in section 6.2.1, length expansion can be avoided in many cases by modifying the representation. The *modified signed fractional window representation* is obtained as follows: if the $w + 2$ most significant digits are of the form

$$1 \underbrace{00 \dots 0}_w \bar{b},$$

then substitute

$$01 \underbrace{0 \dots 0}_{w-1} \beta$$

where $\beta = 2^w - b$; if the $w + 3$ most significant digits are of the form

$$1 \underbrace{00 \dots 0}_{w+1} \bar{b}$$

Table 6.1: Left-to-right exponentiation with window NAFs or signed fractional windows, $\ell = 160$

	$w = 2$		wNAF	$w = 3$			$w = 4$ wNAF
	wNAF	s. fract. $m = 1$		s. fract. $m = 1$	s. fract. $m = 3$	s. fract. $m = 5$	
precomputation stage:							
table entries	2	3	4	5	6	7	8
squarings	1	1	1	1	1	1	1
multiplications	1	2	3	4	5	6	7
evaluation stage:							
squarings	≤ 160	≤ 160	≤ 160	≤ 160	≤ 160	≤ 160	≤ 160
multiplications	≈ 40.0	≈ 35.6	≈ 32.0	≈ 30.5	≈ 29.1	≈ 27.8	≈ 26.7

with $b > 2^w$, then substitute

$$0 \underbrace{10 \dots 0}_w \beta$$

where $\beta = 2^{w+1} - b$; and if the $w + 3$ most significant digits are of the form

$$1 \underbrace{000 \dots 0}_{w+1} \bar{b}$$

with $b < 2^w$, then substitute

$$003 \underbrace{0 \dots 0}_{w-1} \beta$$

where $\beta = 2^w - b$.

Precomputation for left-to-right exponentiation can be done in one squaring and $2^{w-1} + (m-1)/2$ multiplications (see section 6.1.1), and the result stage for right-to-left exponentiation can be implemented in $2^{w-1} + (m-1)/2$ squarings and $2^w + m - 1$ multiplications (see section 6.1.2).

Table 6.1 shows expected performance figures for left-to-right exponentiation using the signed fractional window method in comparison with the usual window NAF method for 160-bit scalars; a typical application is elliptic curve cryptography. (The expected number of evaluation stage multiplications for ℓ -bit scalars is approximately

$$\frac{\ell}{w + \frac{m+1}{2^w} + 2}$$

for the signed fractional window method, and

$$\frac{\ell}{w + 2}$$

for the window NAF method.) The signed fractional window method with $w = 2$, $m = 1$ achieves an evaluation stage speed-up of about 2.3% compared with the window NAF method with $w = 2$, assuming that squarings take as much time as general multiplications. In fact, squarings can be faster, which will increase the relative speed-up (this is usually the case when projective coordinates are used for representing points on an elliptic curve).

Table 6.2 is for right-to-left exponentiation; it takes into account the optimizations to the right-to-left stage noted in section 6.1.2. (As one multiplication can be saved for each

Table 6.2: Right-to-left exponentiation with window NAFs or signed fractional windows, $\ell = 160$

	$w = 2$		$w = 3$				$w = 4$
	wNAF	s. fract. $m = 1$	wNAF	s. fract. $m = 1$	s. fract. $m = 3$	s. fract. $m = 5$	wNAF
right-to-left stage:							
squarings	≤ 160	≤ 160	≤ 160	≤ 160	≤ 160	≤ 160	≤ 160
multiplications	≈ 39.0	≈ 33.6	≈ 29.0	≈ 26.5	≈ 24.1	≈ 21.8	≈ 19.7
result stage:							
input variables	2	3	4	5	6	7	8
squarings	1	2	3	4	5	6	7
multiplications	2	4	6	8	10	12	14

additional accumulator, usually in the right-to-left stage, the expected number of right-to-left stage multiplications for ℓ -bit scalars is approximately

$$\frac{\ell}{w + \frac{m+1}{2^w} + 2} + 1 - 2^{w-1} - \frac{m-1}{2}$$

for the signed fractional window method, and

$$\frac{\ell}{w + 2} + 1 - 2^{w-1}$$

for the window NAF method.) The table shows that at this exponent bit length, for $w = 3$ fractional windows bring hardly any advantage for right-to-left exponentiation due to the relatively high computational cost of the result stage. For $\ell = 160$, the fractional window method with $w = 2$, $m = 1$ achieves a 1.2% total speed-up compared with the window NAF method with $w = 2$, assuming that squarings take as much time as general multiplications.

6.3.2 Unsigned Fractional Windows

The *unsigned fractional window* representation uses the digit set

$$B = \{1, 3, \dots, 2^w + m\}$$

and can be obtained by a variant of the technique from section 6.3.1. Here, let the mapping

$$digit: \{0, 1, \dots, 2^{w+1}\} \rightarrow B \cup \{0\}$$

be defined as follows:

- If x is even, then $digit(x) = 0$;
- otherwise if $0 < x \leq 2^w + m$, then $digit(x) = x$;
- otherwise let $digit(x) = x - 2^w$.

If x is odd, then $x - digit(x) \in \{0, 2^w\}$. The following algorithm encodes e into unsigned fractional window representation:

Table 6.3: Left-to-right exponentiation with sliding windows or unsigned fractional windows, $\ell = 1023$

	$w = 2$		$w = 3$				$w = 4$ slid. w.
	slid. w.	u. fract. $m = 1$	slid. w.	u. fract. $m = 1$	u. fract. $m = 3$	u. fract. $m = 5$	
precomputation stage:							
table entries	2	3	4	5	6	7	8
squarings	1	1	1	1	1	1	1
multiplications	1	2	3	4	5	6	7
evaluation stage:							
squarings	≤ 1023	≤ 1023	≤ 1023	≤ 1023	≤ 1023	≤ 1023	≤ 1023
multiplications	≈ 341.0	≈ 292.3	≈ 255.8	≈ 240.7	≈ 227.3	≈ 215.4	≈ 204.6

```

 $d \leftarrow \text{LSB}_{w+1}(e)$ 
 $c \leftarrow \lfloor e/2^{w+1} \rfloor$ 
 $i \leftarrow 0$ 
while  $d \neq 0 \vee c \neq 0$  do
   $b \leftarrow \text{digit}(d)$ 
   $b_i \leftarrow b; i \leftarrow i + 1$ 
   $d \leftarrow d - b$ 

   $d \leftarrow \text{LSB}(c) \cdot 2^w + d/2$ 
   $c \leftarrow \lfloor c/2 \rfloor$ 
return  $b_{i-1}, \dots, b_0$ 

```

Similarly to the signed case, it can be seen that the average density of the unsigned fractional window representation is

$$\frac{1}{w + \frac{m+1}{2^w} + 1}$$

for $e \rightarrow \infty$. The precomputation or result stage is as before.

Table 6.3 shows expected performance figures for left-to-right exponentiation using the unsigned fractional window method in comparison with the usual sliding window method for 1024-bit scalars; a typical application is exponentiation in the multiplicative semigroup $(\mathbb{Z}/n\mathbb{Z})$ for an integer n . (The expected number of evaluation stage multiplications for $(\ell+1)$ -bit scalars is approximately

$$\frac{\ell}{w + \frac{m+1}{2^w} + 1}$$

for the unsigned fractional window method, and

$$\frac{\ell}{w + 1}$$

for the sliding window method.) If squarings take as much time as general multiplications, the unsigned fractional window method with $w = 2$, $m = 1$ is approximately 3.7% faster than the sliding window method with $w = 2$.

Table 6.4 shows the figures for right-to-left exponentiation, taking into account the optimizations to the right-to-left stage noted in section 6.1.2. (As one multiplication can be saved

Table 6.4: Right-to-left exponentiation with sliding windows or unsigned fractional windows, $\ell = 1023$

	$w = 2$		$w = 3$				$w = 4$
	slid. w.	u. fract. $m = 1$	slid. w.	u. fract. $m = 1$	u. fract. $m = 3$	u. fract. $m = 5$	slid. w.
right-to-left stage:							
squarings	≤ 1023	≤ 1023	≤ 1023	≤ 1023	≤ 1023	≤ 1023	≤ 1023
multiplications	≈ 340.0	≈ 290.3	≈ 252.8	≈ 236.7	≈ 222.3	≈ 209.4	≈ 197.6
result stage:							
input variables	2	3	4	5	6	7	8
squarings	1	2	3	4	5	6	7
multiplications	2	4	6	8	10	12	14

for each additional accumulator, usually in the right-to-left stage, the expected number of right-to-left stage multiplications for $(\ell + 1)$ -bit scalars is approximately

$$\frac{\ell}{w + \frac{m+1}{2^w} + 1} + 1 - 2^{w-1} - \frac{m-1}{2}$$

for the unsigned fractional window method, and

$$\frac{\ell}{w+1} + 1 - 2^{w-1}$$

for the sliding window method.)

6.4 Compact Encodings

When storing a window NAF or fractional window representation where a single digit may take $w + 1$ bits of memory (as it is the case for width- $(w + 1)$ NAFs if we take into account that the digit may be zero, and for signed fractional window representations), then it is not necessary to store digits separately in $w + 1$ bits each. If memory is scarce, it is possible to exploit the properties of the representation to obtain a more compact encoding into bit strings (cf. [45], where this technique is introduced for width-2 NAFs).

We can encode a zero digit as a single zero bit, and a non-zero digit as a one bit followed by a representation of the respective digit, which together takes $w + 1$ bits in the case of window NAFs and $w + 2$ bits in the case of signed fractional window representations. After each non-zero digit, there will be w zero digits (unless conversion into a modified window NAF has taken place), and these can be omitted from the encoding. Thus, compared with the usual binary representation of the number, in the case of window NAFs we only have growth by a small constant; in the case of signed fractional window representations (and similarly in the case of unsigned fractional window representations), we also have growth by an additional bit for each non-zero digit of the representation.

This bit string encoding can easily be adapted to the case that the bit string will be read in the reverse of the direction in which it was written (for example, non-zero digits should be encoded as a representation of the respective digit followed by a one bit rather than the other way around).

Chapter 7

Efficient Multi-Exponentiation

In this chapter, we compare different approaches for computing power products

$$\prod_{1 \leq i \leq k} g_i^{e_i}$$

in commutative semigroups with neutral element where each e_i is a uniformly chosen random integer in an interval $[0, 2^{L_i} - 1]$. Let L be the bit-length of the longest of the e_i (i.e., $e_i \in [0, 2^L - 1]$ for all i , and there is an i such that $e_i \geq 2^{L-1}$). It is well known that the trivial approach to perform multi-exponentiation by computing the powers $g_i^{e_i}$ separately and then multiplying them is, in general, unnecessarily inefficient compared with specific methods for multi-exponentiation. To illustrate how these methods work, the following alternative notation for this power product will be employed (compare with the notation for matrix products):

$$(g_1, \dots, g_k) \bullet \begin{pmatrix} e_1 \\ \vdots \\ e_k \end{pmatrix}$$

Repeated single exponentiations with precomputation for a fixed base will also be considered in this chapter because multi-exponentiation techniques can be used for this task.

Like left-to-right methods for single exponentiations (see section 6.1.1), the multi-exponentiation methods that we will look at work in two stages: first, in the *precomputation stage*, an auxiliary table of semigroup elements is computed from the elements g_i ; then, in the *evaluation stage* (or *left-to-right stage*), the final result is computed using these auxiliary values.

An often-used approach for multi-exponentiation combines all input elements g_i with each other in the precomputation stage ([35], [83], [91]); then the evaluation stage looks at all exponents simultaneously. We refer to these multi-exponentiation methods as *simultaneous exponentiation*. Section 7.1 describes two variants of simultaneous exponentiation: Straus's simultaneous 2^w -ary method [83] and the simultaneous sliding window method of Yen, Laih, and Lenstra [91]. (The method described in [35], which is known as “Shamir's trick”, appears as a special case of both of these.) For these methods, we assume that L_i is the same for all i .

Section 7.2 presents an alternative approach, *interleaved exponentiation*, which treats the g_i separately in the precomputation stage and where the evaluation stage uses an interleaving of the generators and exponents for the various i rather than handling multiple i simultaneously. This approach can be used with the various encoding techniques that we

have examined in chapter 6. Specifically, the *basic interleaved exponentiation method* is the combination of interleaved exponentiation with the left-to-right sliding window technique, and the *window-NAF based interleaved exponentiation method* is the combination of interleaved exponentiation with the window NAF technique. The former method can be used for arbitrary commutative semigroups with neutral element, the latter method is applicable for groups where inverting elements is easy.

In section 7.3, we compare the efficiency of simultaneous exponentiation methods and interleaved exponentiation methods. Our comparison shows that in general semigroups, sometimes simultaneous exponentiation and sometimes interleaved exponentiation is more efficient. In groups where inverting elements is easy (e.g. elliptic curves), window-NAF based interleaved exponentiation usually wins over simultaneous exponentiation.

Section 7.4 discusses variants that can be advantageous when the bases g_i are fixed for many multi-exponentiations. In such cases, precomputation need not be repeated, so it can pay out to invest more work in precomputation to obtain speed-ups. Note that single exponentiation is a special case of multi-exponentiation ($k = 1$), and that these variants can be very useful for this case. One of the techniques presented there is window NAF splitting, which can be used for efficient exponentiation or multi-exponentiation with precomputation in groups where inversion is easy; it provides a convenient alternative to the patented Lim-Lee method.

7.1 Simultaneous Exponentiation

We look at two multi-exponentiation methods using simultaneous exponentiation (as opposed to interleaved exponentiation, which will be introduced in section 7.2): Straus's 2^w -ary method (section 7.1.1) and the sliding window method of Yen, Lai, and Lenstra (section 7.1.2).

As noted in the introduction to this chapter, all algorithms that we consider are related and work in two stages: first, the *precomputation stage* prepares an auxiliary table of semigroup elements; then, the *evaluation stage* (*left-to-right stage*) computes the final result using this table. For comparing different methods, we examine the two stages separately.

Parameter w is always a positive integer, the *window size*; larger window sizes make the precomputation stage less efficient, but speed up the evaluation stage. It is not possible to give a general rule for selecting an optimal w (cf. section 7.3).

Relevant features of the precomputation stage are the number of squarings and general multiplications required for computing the auxiliary table, and the number of table entries. The precomputed tables will always contain the values g_1, \dots, g_k , all of which are trivially available. It will be visible that computing each additional table entry requires one multiplication or, for some of the table entries in the simultaneous 2^w -ary method, one squaring. In addition to this, k squarings are needed by the simultaneous sliding window method if $w > 1$.

The evaluation stage requires both squarings and multiplications. For each multi-exponentiation method, we look at the number of squarings and the expected number of general multiplications for given k , L , and w . In this section, we assume that the maximum bit-length L_i is the same for all i .

The window size w is assumed to be small in comparison with the maximum bit-length L (otherwise the precomputation stage would become unreasonably expensive).

It should be noted that a slight optimization for the precomputation stage is possible in all methods by first looking which table entries are actually needed (either during the evaluation

stage, or because other precomputed table entries that are needed in the evaluation stage depend on them) and limiting precomputation to these. As this optimization will usually only have a small effect in practice, we neglect it in our comparisons.

For the number of squarings in the evaluation stage, we assume that the following optimization is used: as initially variable A is 1_G (the neutral element of G) in all algorithms, squarings can easily be avoided until a different value has been assigned to A .

Formulas for the expected number of multiplications during the evaluation stage given in the following are actually asymptotics for large L/w rather than precise values (we do not take into account the special probability distributions encountered at both ends of the exponents). As in practice w will be much smaller than L , the error can be neglected for our purposes.

Just as squarings can be eliminated in the evaluation stage while A is 1_G , the first multiplication of A by a table entry can be replaced by an assignment. This minor optimization is not used in our figures below; note that it applies similarly to all algorithms discussed in this chapter (and does not affect asymptotics), so comparisons between different methods remain just as valid.

7.1.1 Simultaneous 2^w -Ary Exponentiation Method

Straus's simultaneous 2^w -ary exponentiation method [83] (see also [53]) looks at w bits of each of the exponents for each evaluation stage semigroup multiplication, i.e. kw bits in total. The special case where $w = 1$ is also known as "Shamir's trick" since it was described in [35] with a reference to Shamir (but note that [83] is a much earlier publication).

Precomputation Stage

Precompute $\prod_{1 \leq i \leq k} g_i^{E_i}$ for all non-zero k -tuples $(E_1, \dots, E_k) \in \{0, \dots, 2^w - 1\}^k$.

Number of table entries: $2^{kw} - 1$. Of these, k are trivially available; $2^{k(w-1)} - 1$ can be computed by squaring other table entries (all the E_i are even); the remaining $2^{kw} - 2^{k(w-1)} - k$ entries require one general multiplication each.

Evaluation Stage

For the following algorithm, remember that for a non-negative integer e , $e[j' \dots j]$ denotes the integer consisting of the concatenation of bits j' down to j of e .

```

A ← 1_G
for j = [(L - 1)/w] · w down to 0 step w do
  for n = 1 to w do
    A ← A2
    if (e1[j + w - 1 ... j], ..., ek[j + w - 1 ... j]) ≠ (0, ..., 0) then
      A ← A · ∏i giei[j + w - 1 ... j] {multiply A by table entry}
return A

```

Number of squarings: $\lfloor \frac{L-1}{w} \rfloor \cdot w$.

Expected number of multiplications: $L \cdot \frac{1 - \frac{1}{2^{kw}}}{w}$.

Example

The simultaneous 2^w -ary exponentiation method can be illustrated as follows in a small toy example for the case $k = 3$ with $e_1 = 1011010_2$, $e_2 = 11001_2$, $e_3 = 1001011_2$ and window size $w = 2$:

$$(g_1, \dots, g_k) \bullet \left(\begin{array}{|c|c|} \hline 0 & 1 \\ \hline 0 & 0 \\ \hline 0 & 1 \\ \hline \end{array} \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 0 & 1 \\ \hline 0 & 0 \\ \hline \end{array} \begin{array}{|c|c|} \hline 1 & 0 \\ \hline 1 & 0 \\ \hline 1 & 0 \\ \hline \end{array} \begin{array}{|c|c|} \hline 1 & 0 \\ \hline 0 & 1 \\ \hline 1 & 1 \\ \hline \end{array} \right)_2$$

Each box corresponds to one evaluation stage multiplication.

7.1.2 Simultaneous Sliding Window Exponentiation Method

The simultaneous sliding window exponentiation method of Yen, Laih, and A. Lenstra [91] is an improved variant of the 2^w -ary method described in section 7.1.1. Due to the use of a sliding window, table entries are required only for those tuples (E_1, \dots, E_k) where at least one of the E_i is odd. (Note that while values g_i^2 no longer appear in the precomputed table, the precomputation stage now needs them as intermediate values unless $w = 1$.) Also the expected number of multiplications required in the evaluation stage is reduced. Like the 2^w -ary method, this method looks at w bits of each of the exponents for each evaluation stage semigroup multiplication (kw bits in total). For $w = 1$, this again is ‘‘Shamir’s trick’’. For $k = 1$, this is the usual sliding window method for a single exponentiation (see section 6.2).

Precomputation Stage

Precompute $\prod_{1 \leq i \leq k} g_i^{E_i}$ for all k -tuples $(E_1, \dots, E_k) \in \{0, \dots, 2^w - 1\}^k$ where at least one of the E_i is odd.

Number of table entries: $2^{kw} - 2^{k(w-1)}$.

Number of squarings: k if $w > 1$; none otherwise.

Number of general multiplications: $2^{kw} - 2^{k(w-1)} - k$.

Evaluation Stage

```

A ← 1G
j ← L - 1
while j ≥ 0 do
  if ∀i ∈ {1, ..., k}: ei[j] = 0 then
    A ← A2; j ← j - 1
  else
    jnew ← max(j - w, -1)
    J ← jnew + 1
    while ∀i ∈ {1, ..., k}: ei[J] = 0 do
      J ← J + 1
    {now j ≥ J > jnew}
    for i = 1 to k do
      Ei ← ei[j ... J]
    while j ≥ J do
      A ← A2; j ← j - 1
    A ← A · ∏i giEi  {multiply A by table entry}

```

```

while  $j > j_{\text{new}}$  do
     $A \leftarrow A^2; j \leftarrow j - 1$ 
return  $A$ 

```

Number of squarings: $L - w$ up to $L - 1$.

Expected number of multiplications: $L \cdot \frac{1}{w + \sum_{n \geq 1} \frac{1}{2^{kn}}} = L \cdot \frac{1}{w + \frac{1}{2^k - 1}}$.

Example

The simultaneous sliding window exponentiation method can be illustrated as follows in a small toy example for the case $k = 3$ with $e_1 = 1011010_2$, $e_2 = 11001_2$, $e_3 = 1001011_2$ and window size $w = 2$:

$$(g_1, \dots, g_k) \bullet \left(\begin{array}{|c|c|} \hline \mathbf{1} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} \\ \hline \mathbf{1} & \mathbf{0} \\ \hline \end{array} \begin{array}{|c|c|} \hline \mathbf{1} & \mathbf{1} \\ \hline \mathbf{1} & \mathbf{1} \\ \hline \mathbf{0} & \mathbf{1} \\ \hline \end{array} \begin{array}{|c|} \hline \mathbf{0} \\ \hline \mathbf{0} \\ \hline \mathbf{0} \\ \hline \end{array} \begin{array}{|c|c|} \hline \mathbf{1} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{1} \\ \hline \mathbf{1} & \mathbf{1} \\ \hline \end{array} \begin{array}{|c|} \hline \mathbf{2} \\ \hline \mathbf{2} \\ \hline \mathbf{2} \\ \hline \end{array} \right)$$

Each bold box corresponds to one evaluation stage multiplication. The thin box indicates a range of bits that is scanned by the algorithm as part of one window, and where the right-most column of bits is found to contain all zeros so that the case occurs that $J > j_{\text{new}} + 1$.

7.2 Interleaved Exponentiation

Let a B_j -representation

$$e_j = \sum_{0 \leq i \leq \ell_j} b_{j,i} \cdot 2^i$$

be given for each of the exponents in a power product

$$\prod_{1 \leq j \leq k} g_j^{e_j},$$

where each B_j is a digit set as in section 6.1. Then the multi-exponentiation can be performed by *interleaving* the left-to-right algorithms for the individual exponentiations $g_j^{e_j}$. For each j , precomputed elements g_j^b are needed as in section 6.1.1.

Let ℓ be the maximum of the ℓ_j . We may assume that $\ell = \ell_1 = \dots = \ell_k$ (pad with leading zeros where necessary). When the B_j -representations have been obtained by the sliding window technique (section 6.2) or the unsigned fractional window technique (section 6.3), we can assume $\ell = L - 1$; when the B_j -representations have been obtained by the window NAF technique (section 6.2) or the signed fractional window technique (section 6.3), we can assume $\ell = L$.

If division is permissible, interleaved exponentiation to compute $\prod_{1 \leq j \leq k} g_j^{e_j}$ can be performed as follows:

```

 $A \leftarrow 1_G$ 
for  $i = \ell$  down to  $0$  do
     $A \leftarrow A^2$ 
    for  $j = 1$  to  $k$  do
        if  $b_{j,i} \neq 0$  then
            if  $b_{j,i} > 0$  then

```

```

        A ← A · gjbj,i
    else
        A ← A/gj|bj,i|
    return A

```

As in section 6.1.1, initial squarings can be omitted while A is 1_G , and the first multiplication or division can be replaced by an assignment possibly followed by inversion. The algorithm variant without division is obvious.

In the following, we look at two specific interleaved exponentiation methods more closely: the *basic interleaved exponentiation method* (section 7.2.1), which uses the sliding window technique and is suitable for arbitrary commutative semigroups with neutral element, and the *window-NAF based interleaved exponentiation method* (section 7.2.2), which uses the window NAF technique and can be applied for groups where inverting elements is easy.

The comments in the introduction to section 7.1 apply similarly, with the exception that we no longer assume all the L_i to be identical. Instead of a single window size w , in this section we have k possibly different window sizes w_i ($1 \leq i \leq k$) used for the respective parts of the multi-exponentiation; each w_i is a small positive integer. Again we assume that initial squarings are eliminated while A is 1_G .

Note that for the algorithms described in this section, the precomputed table has disjoint parts for different bases g_i . If multiple multi-exponentiations have to be performed and some of the bases g_i appear again, then the corresponding parts of earlier precomputed tables can be reused.

Observe that it is easy to implement interleaved exponentiation for a variable parameter k . As the special case $k = 1$ of the basic and window-NAF based interleaved exponentiation methods yields the usual sliding windows exponentiation method and window-NAF based exponentiation method, respectively, this makes it unnecessary to implement these separately.

7.2.1 Basic Interleaved Exponentiation Method

The basic interleaved exponentiation method is a generalization of the left-to-right sliding window method for a single exponentiation (see sections 6.1.1 and 6.2), to which it corresponds in the case $k = 1$. We show how it can be used without computing sliding window representations of all exponents beforehand.

Precomputation Stage

For $i = 1, \dots, k$, precompute g_i^E for all odd E such that $1 \leq E \leq 2^{w_i} - 1$.

Number of table entries: $\sum_{1 \leq i \leq k} 2^{w_i - 1}$.

Number of squarings: $\#\{i \in \{1, \dots, k\} \mid w_i > 1\}$.

Number of general multiplications: $(\sum_{1 \leq i \leq k} 2^{w_i - 1}) - k$.

Evaluation Stage

```

A ← 1G
for i = 1 to k do
    window_handlei ← nil
for j = L - 1 down to 0 do
    A ← A2

```

```

for  $i = 1$  to  $k$  do
  if  $window\_handle_i = \text{nil}$  and  $e_i[j] = 1$  then
     $J \leftarrow j - w_i + 1$ 
    while  $e_i[J] = 0$  do
       $J \leftarrow J + 1$ 
    {now  $j \geq J > j - w_i$  and  $J \geq 0$ }
     $window\_handle_i \leftarrow J$ 
     $E_i \leftarrow e_i[j \dots J]$ 
  if  $window\_handle_i = j$  then
     $A \leftarrow A \cdot g_i^{E_i}$  {multiply  $A$  by table entry}
     $window\_handle_i \leftarrow \text{nil}$ 
return  $A$ 

```

Number of squarings: $L - \max_{1 \leq i \leq k} (w_i)$ up to $L - 1$.

Expected number of multiplications:

$$\sum_{1 \leq i \leq k} L_i \cdot \frac{1}{w_i + \sum_{n \geq 1} \frac{1}{2^n}} = \sum_{1 \leq i \leq k} L_i \cdot \frac{1}{w_i + 1}.$$

Example

The basic interleaved exponentiation method can be illustrated as follows in a small toy example for the case $k = 3$ with $e_1 = 1011010_2$, $e_2 = 11001_2$, $e_3 = 1001011_2$ and parameters $w_1 = w_2 = w_3 = 3$:

$$(g_1, \dots, g_k) \cdot \begin{pmatrix} \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{1} & 0_2 \\ 0 & 0 & \boxed{1} & \boxed{1} & \boxed{0} & 0 & \boxed{1}_2 \\ \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1}_2 \end{pmatrix}$$

Each bold box corresponds to one evaluation stage multiplication. Each thin box indicates a range of bits that is scanned by the algorithm as part of one window.

7.2.2 Window-NAF Based Interleaved Exponentiation Method

The window-NAF based interleaved exponentiation can be used in groups where elements can be inverted very efficiently so that division is not significantly more expensive than multiplication: it applies the window NAF technique described in section 6.2 for interleaved exponentiation. Note that modified window NAFs can be employed (see section 6.2.1).

Precomputation Stage

For $i = 1, \dots, k$, precompute g_i^E for all odd E such that $1 \leq E \leq 2^{w_i} - 1$.

Number of table entries: $\sum_{1 \leq i \leq k} 2^{w_i - 1}$.

Number of squarings: $\#\{i \in \{1, \dots, k\} \mid w_i > 1\}$.

Number of general multiplications: $(\sum_{1 \leq i \leq k} 2^{w_i - 1}) - k$.

Evaluation Stage

See page 75 for the interleaved exponentiation algorithm.

Number of squarings: $L - \max_{1 \leq i \leq k} (w_i)$ up to L .

Expected number of multiplications: $\sum_{1 \leq i \leq k} L_i \cdot \frac{1}{w_i + 2}$.

The sliding window encoding technique used by the basic interleaved exponentiation method in section 7.2.1 has an average density of $1/(w_i + 1)$. With window NAFs, the average density goes down to $1/(w_i + 2)$ for exactly the same precomputation. Thus using window NAFs effectively increases window sizes by one.

7.3 Comparison of Simultaneous and Interleaved Exponentiation Methods

There is no general rule for selecting window sizes for the multi-exponentiation algorithms that we have looked at. Various factors have to be considered: first of all, absolute memory constraints can impose limits on possible window sizes. Second, even if a particular window size appears to minimize the total amount of computation for a multi-exponentiation, sometimes slightly smaller windows may improve the actual performance; this is because larger window sizes mean larger precomputed tables, i.e. possibly additional memory allocation overhead and less effective memory caching. Last but not least, implementations can use different representations for semigroup elements during different stages of the multi-exponentiation: for instance, extra effort may be spent during the precomputation stage in order to obtain representations of precomputed elements that speed up multiplication with them in the evaluation stage (e.g. affine rather than projective representations of points on elliptic curves [27]).

These effects, however, do not mean that we cannot compare algorithms without looking at concrete cases: we can compare different aspects separately (table size, precomputation stage efficiency, evaluation stage efficiency) and look if an algorithm wins on all counts.

For the following comparisons, we assume that all maximum exponent lengths L_i are the same (an assumption that we made in section 7.1 on simultaneous exponentiation, but not in section 7.2 on interleaved exponentiation). As before, let L be the length of the largest of the exponents e_i .

In section 7.3.1, we compare the simultaneous 2^w -ary method with the basic interleaved method and show that the latter is usually more efficient for $k = 2$ if squarings are about as costly as multiplications. In section 7.3.2, we compare the simultaneous sliding window method with the window-NAF based interleaved method and show that the latter is more efficient for $k = 2$ and $k = 3$ (assuming that computing and storing the window NAFs is not too costly). Section 7.3.3 briefly discusses the alternative multi-exponentiation method from [34] and shows that is obviated by our interleaved exponentiation methods; section 7.3.4 shows an example for the use of fractional windows with interleaved multi-exponentiation. In section 7.3.5, we look at some concrete figures for the number of multiplications required by different methods for example values of k and L assuming that window size choices are not dictated by scarce memory. Section 7.3.6 gives conclusions.

7.3.1 Comparison between the Simultaneous 2^w -Ary Method and the Basic Interleaved Method

While the simultaneous sliding window method is more efficient than the simultaneous 2^w -ary method, this section focuses on the latter. The reason is that the 2^w -ary method is often used in practice (e.g. [18]), possibly because it is perceived to be simpler to implement. The basic interleaved exponentiation method is quite simple (in particular, indexes into the precomputed table are easy to handle), and as we will see, is often more efficient than the simultaneous 2^w -ary exponentiation method. So when the intention is to avoid the simultaneous sliding window method, the basic interleaved method appears preferable for many applications.

Assume that, given k and L , a certain w turns out to provide optimal efficiency for the simultaneous 2^w -ary exponentiation method (section 7.1.1) when performed in a specific environment. Then the precomputed table has $2^{kw} - 1$ entries (including the k trivial entries g_1, \dots, g_k), $2^{k(w-1)} - 1$ of which can be computed with one squaring each, while each of the remaining $2^{kw} - 2^{k(w-1)} - k$ non-trivial entries requires one general multiplication.

For the basic interleaved exponentiation method (section 7.2.1), we can use uniform window sizes $w_1 = \dots = w_k = kw$. Then the precomputed table has $k2^{kw-1}$ entries, $k2^{kw-1} - k$ of which require one general multiplication each; also k additional squarings are needed (unless $k = w = 1$).

Thus in case $k = 2$, the number of table entries grows from $2^{2w} - 1$ to 2^{2w} , and instead of $2^{2w} - 3$ semigroup operations of which $2^{2(w-1)} - 1$ are squarings, we need 2^{2w} semigroup operations of which only 2 are squarings. If squarings are about as expensive as general multiplications, then for $k = 2$ the overall cost of precomputation is comparable for these two multi-exponentiation methods.

The number of squarings in the evaluation stage is always nearly L for both methods. The expected number of general multiplications in the evaluation stage is smaller for the interleaved method (except if $k = w = 1$, in which case both algorithms perform exactly the same operations): dividing the value for the basic interleaved exponentiation method by the value for the simultaneous 2^w -ary exponentiation method yields

$$\frac{k}{kw + 1} \cdot \frac{w}{1 - \frac{1}{2^{kw}}} = \frac{kw}{kw + 1} \cdot \frac{2^{kw}}{2^{kw} - 1},$$

and this is less than 1 for $kw > 1$ (the minimum is $64/75$ at $kw = 4$).

Note that using $w_1 = \dots = w_k = kw$ is not necessarily an optimal choice of window sizes for the basic interleaved exponentiation method; using smaller or larger windows might lead to better performance. (Indeed, if we look just at the number of operations and ignore memory usage, then there is no reason why window sizes should depend on k .) While the above proof only covers the case $k = 2$, there are actually many other cases where the basic interleaved method is more efficient than the simultaneous 2^w -ary method, even if general multiplications are much more expensive than squaring; see table 7.1 in section 7.3.5. Also note that the precomputation effort grows exponentially in k for simultaneous methods, but not for interleaved methods.

7.3.2 Comparison between the Simultaneous Sliding Window Method and the Window-NAF Based Interleaved Method

Similarly to section 7.3.1, assume that a certain w provides optimal efficiency for the simultaneous sliding window exponentiation method (section 7.1.2) for given k and L . In the

following analysis, we require $k > 1$. The precomputed table has $2^{kw} - 2^{k(w-1)}$ entries (including k trivial ones), $2^{kw} - 2^{k(w-1)} - k$ of which requires one general multiplication to compute. In addition to this, k squarings are required for precomputation unless $w = 1$.

For the window-NAF based interleaved exponentiation method (section 7.2.2), we can use window sizes $w_1 = \dots = w_k = kw - 1$. This leads to a precomputed table with $k2^{kw-2}$ entries, where k of these are trivially available and the $k2^{kw-2} - k$ non-trivial ones require one general multiplication each. In addition to this, we need k squarings unless $kw = 2$.

The difference between the number of tables entries (and between the number of general multiplications) for these two methods is

$$(2^{kw} - 2^{k(w-1)}) - k2^{kw-2} = 2^{kw} \left(1 - 2^{-k} - \frac{k}{4} \right).$$

This is positive for $k \leq 3$ and negative for $k \geq 4$. Thus, with the w_i chosen like this, the precomputation stage of the window-NAF based interleaved exponentiation method is more efficient if $k = 2$ or $k = 3$ (except for the case $k = 3$, $w = 1$, where the window-NAF based interleaved exponentiation method saves one general multiplication, but requires three additional squarings).

The evaluation stage requires close to L squarings for both methods. The expected number of general multiplications is smaller for the window-NAF based interleaved method: $L/(w + 1/k)$ instead of $L/(w + \frac{1}{2^k - 1})$.

The window-NAF based interleaved method with this choice of window sizes will often provide better performance than the simultaneous sliding window method for $k \geq 4$ as well: if additional memory allocation is not a problem, then the efficiency gain of the evaluation stage usually compensates for the growth of the precomputed table.

Similar to the situation in the preceding section, $w_1 = \dots = w_k = kw - 1$ is not necessarily an optimal choice, and smaller or larger window sizes might be better (see section 7.3.5).

7.3.3 Comparison between the Dimitrov-Jullien-Miller Multi-Exponentiation Method and Interleaved Exponentiation

A multi-exponentiation method for computing $g_1^{e_1} g_2^{e_2}$ that requires a precomputed table containing four values (including g_1 and g_2) if inverting is easy, or eight values if inversions have to be done during the precomputation stage to avoid them in the evaluation stage, was described by Dimitrov, Jullien, and Miller in [34]. This algorithm is related to the simultaneous sliding window exponentiation method of Yen, Laih, and Lenstra [91] (see section 7.1.2), but uses a signed encoding of exponents in order to reduce the size of the precomputed table. While the Yen-Laih-Lenstra method with a window size of 1 requires an expected number of $L \cdot 0.75$ general multiplications during the evaluation stage, the new method requires only about $L \cdot 0.534$ multiplications according to [34] (the number of squarings stays about the same). Yen-Laih-Lenstra with a window size of 2 needs only $L \cdot 3/7 \approx L \cdot 0.429$ multiplications (table 3 of [34] erroneously assumes a value of $L \cdot 0.625$), but has the disadvantage of requiring more precomputed elements, which may be a problem in constrained environments.

We do not examine the algorithm of [34] in detail; note that it is outperformed by the window-NAF based interleaved method of section 7.2.2 with $w_1 = w_2 = 2$ if inversion is cheap (precomputed table with four elements, $L \cdot 0.5$ evaluation stage multiplications) and by the basic interleaved method of section 7.2.1 with $w_1 = w_2 = 3$ otherwise (precomputed table with eight elements, $L \cdot 0.5$ evaluation stage multiplications).

7.3.4 Multi-Exponentiation with Fractional Windows

Assume we have to compute a power product $g_1^{e_1} g_2^{e_2}$ in a group where inversion is easy, and that we have storage for five precomputed elements. For using interleaved exponentiation as described in section 7.2, we can represent e_1 as a width-3 NAF and e_2 in signed fractional window representation (section 6.3.1) with $w = 2$, $m = 1$. This means we use precomputed elements $g_1, g_1^3, g_2, g_2^3, g_2^5$. The evaluation stage needs at most L squarings and approximately $(\frac{1}{4} + \frac{1}{4+1/2})L = \frac{17}{36}L$ multiplications on average, compared with $\frac{1}{2}L$ multiplications for interleaved exponentiation with width-3 NAFs for both exponents (precomputed elements g_1, g_1^3, g_2, g_2^3).

(A similar scenario is considered in [77], using a different multi-exponentiation algorithm; for groups where inversion is easy, that technique using the same amount of storage as needed in our above example runs slightly slower according to the heuristical results in [77, table 12].)

7.3.5 Examples

As noted before, endless variations are possible for defining optimization goals. In this section, we ignore memory usage and squarings and the issue of different element representations; we make comparisons based just on the expected number of general multiplications required by the various methods, precomputation and evaluation stage combined. Window sizes are chosen such that this cost measure is minimized. Note that the number of squarings is approximately the same for the simultaneous sliding window method, the basic interleaved method, and the window-NAF based interleaved method: no more than k squarings are needed in the precomputation stage, and close to L squarings are needed in the evaluation stage. The simultaneous 2^w -ary method requires $2^{k(w-1)} - 1$ squarings for precomputation and again close to L evaluation stage squarings; so ignoring the cost of squaring tends to favor this method.

Table 7.1 compares the number of general multiplications needed by these four methods for various k and L values, using the following estimates:

$$c_1 = 2^{kw} - 2^{k(w-1)} - k + L \cdot \frac{1 - \frac{1}{2^{kw}}}{w}$$

for the simultaneous 2^w -ary exponentiation method,

$$c_2 = 2^{kw} - 2^{k(w-1)} - k + L \cdot \frac{1}{w + \frac{1}{2^{k-1}}}$$

for the simultaneous sliding window exponentiation method,

$$c_3 = k \cdot \left(2^{w-1} - 1 + \frac{L}{w+1} \right)$$

for the basic interleaved exponentiation method, and

$$c_4 = k \cdot \left(2^{w-1} - 1 + \frac{L}{w+2} \right)$$

for the window-NAF based interleaved exponentiation method, where interleaved exponentiation uses a uniform window size $w_1 = \dots = w_k = w$. The entries for the most efficient methods in a particular configuration are printed in bold: for groups where inversion is easy

so that the window-NAF based method can be used, it wins in all of these examples; for general semigroups, sometimes the simultaneous sliding window method and sometimes the basic interleaved method requires the least number of multiplications. (Remember that for $w = 1$ there is no difference between the simultaneous 2^w -ary method and the simultaneous sliding window method; for $w > 1$, the former is always less efficient.)

7.3.6 Conclusions

In many cases, the basic interleaved exponentiation method compares favorably to the simultaneous 2^w -ary method, in particular if $k = 2$ and squarings are about as costly as general multiplications. In groups where inverting elements is easy so that the window-NAF based interleaved exponentiation method is available, its efficiency is superior even to the sliding window variant of simultaneous exponentiation both in the precomputation stage and the evaluation stage if $k = 2$ or $k = 3$, and it is usually more efficient for larger k as well. In all cases, interleaved exponentiation provides the following advantages over simultaneous exponentiation:

- Improved efficiency if the bit-lengths of the exponents e_i differ significantly.
- More flexibility in choice of the size of the auxiliary table (and, hence, the time spent on precomputation), particularly if k is large.
- Better handling of situations where one or more of the g_i are fixed while others are variable between multiple multi-exponentiation: a corresponding part of the precomputation has to be done only once. (This is the case in DSA [66] and ECDSA [4] signature verification if multiple signatures are verified that are based on the same underlying parameters.)

Thus, depending on circumstances, either the simultaneous sliding window method or one of the interleaved exponentiation methods may be advantageous.

7.4 Exponentiation and Multi-Exponentiation with Precomputation for Fixed Bases

When many multi-exponentiations use the same bases g_1, \dots, g_k , it is sufficient to execute the precomputation stage just once, and we can try to make the evaluation stage more efficient by investing more work in precomputation. (This also applies to the special case $k = 1$, i.e. to repeated single exponentiations use the same base.) We cannot easily reduce the number of general multiplications in the evaluation stage, but we can reduce the number of squarings by using exponent splitting (section 7.4.1), the Lim-Lee “comb” method (section 7.4.2), or window NAF splitting (section 7.4.3). Which approach is the most efficient depends on details of the situation such as exponent lengths, the permissible size of the precomputed table, the relative cost of squarings versus general multiplications, and whether inexpensive inversion is available so that window NAF methods are applicable.

Let m be an arbitrary positive integer. Assuming that fixed exponent length bounds L_i are known, we show how to evaluate power products $\prod_{1 \leq i \leq k} g_i^{e_i}$ with at most $m - 1$ evaluation stage squarings (or occasionally m squarings in section 7.4.3), using a precomputed table independent of the specific exponents e_i .

Table 7.1: Expected number of general multiplications for a multi-exponentiation $\prod_{1 \leq i \leq k} g_i^{e_i}$ with exponents up to L bits (c_1 : simultaneous 2^w -ary method, c_2 : simultaneous sliding window method, c_3 : basic interleaved method, c_4 : window-NAF based interleaved method)

k		$L = 160$	$L = 256$	$L = 512$	$L = 1024$	$L = 2048$
1	c_1	44.5 ($w=4$)	64.6 ($w=5$)	114.2 ($w=5$)	199.0 ($w=6$)	353.3 ($w=7$)
	c_2	39.0 ($w=4$)	57.7 ($w=5$)	100.3 ($w=5$)	177.3 ($w=6$)	319.0 ($w=7$)
	c_3	39.0 ($w=4$)	57.7 ($w=5$)	100.3 ($w=5$)	177.3 ($w=6$)	319.0 ($w=7$)
	c_4	33.7 ($w=4$)	49.7 ($w=4$)	88.1 ($w=5$)	159.0 ($w=6$)	287.0 ($w=6$)
2	c_1	85.0 ($w=2$)	130.0 ($w=2$)	214.0 ($w=3$)	382.0 ($w=3$)	700.0 ($w=4$)
	c_2	78.6 ($w=2$)	119.7 ($w=2$)	199.6 ($w=3$)	353.2 ($w=3$)	660.4 ($w=3$)
	c_3	78.0 ($w=4$)	115.3 ($w=5$)	200.7 ($w=5$)	354.6 ($w=6$)	638.0 ($w=7$)
	c_4	67.3 ($w=4$)	99.3 ($w=4$)	176.3 ($w=5$)	318.0 ($w=6$)	574.0 ($w=6$)
3	c_1	131.8 ($w=2$)	179.0 ($w=2$)	305.0 ($w=2$)	557.0 ($w=2$)	1061.0 ($w=2$)
	c_2	127.7 ($w=2$)	172.5 ($w=2$)	291.9 ($w=2$)	530.9 ($w=2$)	1008.7 ($w=2$)
	c_3	117.0 ($w=4$)	173.0 ($w=5$)	301.0 ($w=5$)	531.9 ($w=6$)	957.0 ($w=7$)
	c_4	101.0 ($w=4$)	149.0 ($w=4$)	264.4 ($w=5$)	477.0 ($w=6$)	861.0 ($w=6$)
4	c_1	161.0 ($w=1$)	251.0 ($w=1$)	491.0 ($w=1$)	746.0 ($w=2$)	1256.0 ($w=2$)
	c_2	161.0 ($w=1$)	251.0 ($w=1$)	483.7 ($w=2$)	731.5 ($w=2$)	1227.0 ($w=2$)
	c_3	156.0 ($w=4$)	230.7 ($w=5$)	401.3 ($w=5$)	709.1 ($w=6$)	1276.0 ($w=7$)
	c_4	134.7 ($w=4$)	198.7 ($w=4$)	352.6 ($w=5$)	636.0 ($w=6$)	1148.0 ($w=6$)
5	c_1	181.0 ($w=1$)	274.0 ($w=1$)	522.0 ($w=1$)	1018.0 ($w=1$)	2010.0 ($w=1$)
	c_2	181.0 ($w=1$)	274.0 ($w=1$)	522.0 ($w=1$)	1018.0 ($w=1$)	1994.7 ($w=2$)
	c_3	195.0 ($w=4$)	288.3 ($w=5$)	501.7 ($w=5$)	886.4 ($w=6$)	1595.0 ($w=7$)
	c_4	168.3 ($w=4$)	248.3 ($w=4$)	440.7 ($w=5$)	795.0 ($w=6$)	1435.0 ($w=6$)
6	c_1	214.5 ($w=1$)	309.0 ($w=1$)	561.0 ($w=1$)	1065.0 ($w=1$)	2073.0 ($w=1$)
	c_2	214.5 ($w=1$)	309.0 ($w=1$)	561.0 ($w=1$)	1065.0 ($w=1$)	2073.0 ($w=1$)
	c_3	234.0 ($w=4$)	346.0 ($w=5$)	602.0 ($w=5$)	1063.7 ($w=6$)	1914.0 ($w=7$)
	c_4	202.0 ($w=4$)	298.0 ($w=4$)	528.9 ($w=5$)	954.0 ($w=6$)	1722.0 ($w=6$)
7	c_1	278.8 ($w=1$)	374.0 ($w=1$)	628.0 ($w=1$)	1136.0 ($w=1$)	2152.0 ($w=1$)
	c_2	278.8 ($w=1$)	374.0 ($w=1$)	628.0 ($w=1$)	1136.0 ($w=1$)	2152.0 ($w=1$)
	c_3	273.0 ($w=4$)	403.7 ($w=5$)	702.3 ($w=5$)	1241.0 ($w=6$)	2233.0 ($w=7$)
	c_4	235.7 ($w=4$)	347.7 ($w=4$)	617.0 ($w=5$)	1113.0 ($w=6$)	2009.0 ($w=6$)
8	c_1	406.4 ($w=1$)	502.0 ($w=1$)	757.0 ($w=1$)	1267.0 ($w=1$)	2287.0 ($w=1$)
	c_2	406.4 ($w=1$)	502.0 ($w=1$)	757.0 ($w=1$)	1267.0 ($w=1$)	2287.0 ($w=1$)
	c_3	312.0 ($w=4$)	461.3 ($w=5$)	802.7 ($w=5$)	1418.3 ($w=6$)	2552.0 ($w=7$)
	c_4	269.3 ($w=4$)	397.3 ($w=4$)	705.1 ($w=5$)	1272.0 ($w=6$)	2296.0 ($w=6$)
9	c_1	661.7 ($w=1$)	757.5 ($w=1$)	1013.0 ($w=1$)	1524.0 ($w=1$)	2546.0 ($w=1$)
	c_2	661.7 ($w=1$)	757.5 ($w=1$)	1013.0 ($w=1$)	1524.0 ($w=1$)	2546.0 ($w=1$)
	c_3	351.0 ($w=4$)	519.0 ($w=5$)	903.0 ($w=5$)	1595.6 ($w=6$)	2871.0 ($w=7$)
	c_4	303.0 ($w=4$)	447.0 ($w=4$)	793.3 ($w=5$)	1431.0 ($w=6$)	2583.0 ($w=6$)
10	c_1	1172.8 ($w=1$)	1268.8 ($w=1$)	1524.5 ($w=1$)	2036.0 ($w=1$)	3059.0 ($w=1$)
	c_2	1172.8 ($w=1$)	1268.8 ($w=1$)	1524.5 ($w=1$)	2036.0 ($w=1$)	3059.0 ($w=1$)
	c_3	390.0 ($w=4$)	576.7 ($w=5$)	1003.3 ($w=5$)	1772.9 ($w=6$)	3190.0 ($w=7$)
	c_4	336.7 ($w=4$)	496.7 ($w=4$)	881.4 ($w=5$)	1590.0 ($w=6$)	2870.0 ($w=6$)

7.4.1 Exponent Splitting

Exponent splitting (cf. [17] and [31]; the idea goes back to [73]) constructs a new power product representation by rewriting each factor as follows:

$$g_i^{e_i} = \prod_{0 \leq j < \lceil L_i/m \rceil} (g_i^{2^{jm}})^{e_i[jm+m-1 \dots jm]}$$

This leads to power products consisting of $\sum_{1 \leq i \leq k} \lceil L_i/m \rceil$ factors. Any multi-exponentiation method can be used for evaluating these power products.

It is evident that for the multi-exponentiation methods that we have looked at, exponent splitting does not help if k is already large and there are many large exponents: saving some evaluation stage squarings will not have a large overall impact in such cases. (Then, instead of using precomputed table entries for additional bases, window sizes should be increased. The evaluation stage will require more squarings, but fewer general multiplications than with exponent splitting.)

7.4.2 Lim-Lee Precomputation

To apply the Lim-Lee “comb” method [51], for every i we choose w_i such that $L_i \leq w_i m$ and precompute

$$G_i(S) := g_i^{\sum_{j \in S} 2^j}$$

for all subsets $S \subseteq \{0, m, 2m, \dots, (w_i - 1)m\}$. Note that then every exponent up to L_i bits of length can be written as

$$e_i = \sum_{0 \leq j < m} b_{j,i} \cdot 2^j$$

where each $b_{j,i}$ is an integer of the form $\sum_{j \in S} 2^j$ with S as above. Thus we can use the interleaved exponentiation algorithm from section 7.2 (where we have no need for divisions because all digits are non-negative) with the number of loop iterations reduced to m . The $b_{j,i}$ values for each iteration need not be stored in advance, they can be extracted from the e_i by tapping their bits in comb-shaped patterns; hence the nickname of this method.

A refinement of this (also from [51]) is based on the observation that the precomputed table can be reduced in size in exchange for additional evaluation stage multiplications: partition $\{0, m, 2m, \dots, (w_i - 1)m\}$ into v_i subsets $T_{i,1}, \dots, T_{i,v_i}$; now each of the above sets S can be written as $\bigcup_{1 \leq n \leq v_i} S_n$ with $S_n = S \cap T_{i,n}$, and then we have $G_i(S) = \prod_{1 \leq n \leq v_i} G_i(S_n)$. Thus it suffices to precompute $G_i(S_n)$ for all non-zero subsets $S_n \subseteq T_{i,n}$ for all n ; from this precomputed data, $G_i(S)$ can be computed in at most $v_i - 1$ multiplications when it is needed in the evaluation stage.

While Lim-Lee precomputation reduces the number of squarings, the expected number of general multiplications is larger than for the basic interleaved exponentiation method with a similarly sized precomputed table. (In the basic interleaved method, 2^{w_i-1} precomputed table entries suffice to make sure that each evaluation stage multiplication covers w_i exponent bits, and we can skip many additional zero bits thanks to the sliding window. With Lim-Lee precomputation, we need at least $2^W - 1$ precomputed table entries to be able to cover W exponent bits with each evaluation stage multiplication, and we lose the advantage of a sliding window.) Thus if k is large, using Lim-Lee precomputation is a disadvantage.

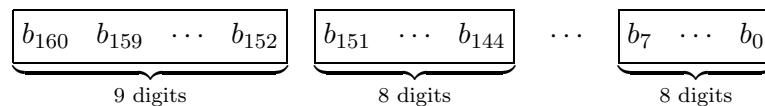
Note that it is possible to use Lim-Lee precomputation for some of the bases and standard precomputation (as in section 7.2) for others. This does not help for multi-exponentiation in these mixed cases, but precomputed data can then profitably be reused for pure Lim-Lee cases.

The Lim-Lee method can be considered an application of exponent splitting using specific multi-exponentiation algorithms suited for small exponents: for example, if $k = 1$, the simple Lim-Lee method uses “Shamir’s trick”, i.e. simultaneous exponentiation with a window size of 1. Further algorithmic variations are possible.

7.4.3 Window NAF Splitting

For groups where inversion is easy, exponent splitting as described in section 7.4.1 could be used with window-NAF based interleaved exponentiation: that is, each of the length- m exponent parts is encoded as a window NAF as described in section 6.2, and then an interleaved exponentiation using these windows NAFs is performed as described in section 7.2. With width- $(w_i + 1)$ NAFs, this computation should take about m squarings and $L_i/(w_i + 2)$ multiplications using $\lceil (L_i + 1)/m \rceil \cdot 2^{w_i - 1}$ precomputed elements. However, if m is very small, the expected number of multiplications will be noticeably higher because the estimate that the density of window NAFs is approximately $1/(w_i + 2)$ becomes accurate only if the encoded number is sufficiently long. (Window NAFs usually waste part of one window; the more individual integers must be encoded into window NAFs, the more is wasted in total.)

An improved technique that avoids this drawback is *window NAF splitting*. Instead of splitting the binary representation of an exponent e_i into partial exponents of length m and determining window NAFs for these, we first determine the window NAF of e_i and then split this new representation into parts of length m . The computation continues as above, using the interleaved exponentiation algorithm shown in section 7.2. To avoid length expansion if possible, this technique should be used with modified window NAFs (section 6.2.1) The leftmost part can be made large than the others if one more part would have to be added otherwise; e.g. for integers up to 160 bits with $m = 8$:



Most of the time, the additional digit of the leftmost part will be zero since length expansion is relatively rare (for modified window NAFs of positive integers up to a length of L_i bits with $w_i = 4$, only about one out of five cases has a non-zero digit at maximum index L_i).

With window NAF splitting, single exponentiations ($k = 1$) for L_1 -bit exponents can be performed in $m - 1$ squarings and on average about $L_1/(w_1 + 2)$ multiplications, using $\lceil (L_1 + 1)/m \rceil \cdot 2^{w_1 - 1}$ precomputed elements. If the leftmost part gets an extra digit as described above, $\lceil L_1/m \rceil \cdot 2^{w_1 - 1}$ precomputed elements are sufficient, and the number of squarings goes up to m for some cases.

This method can compete with the Lim-Lee method even when much space is available for precomputed elements (whereas exponent splitting with window-NAF based interleaved exponentiation is better than the Lim-Lee algorithm only for comparatively small precomputed tables). For example, if $L_1 = 160$, then with $m = 8$ and $w_1 = 4$ (160 precomputed elements if we allow an extra digit in the leftmost window NAF part), our exponentiation method with window NAF splitting needs about 7.2 squarings and 26.7 multiplications. The refined

version of the Lim-Lee method (with $v_1 = 2$ and $\#T_{(1,1)} = \#(T_{1,2}) = w_1/2$) can perform such 160-bit exponentiations in 13 squarings and about 26.6 multiplications using 126 precomputed elements ($m = 14$, $w_1 = 12$), or in 11 squarings and about 22.8 multiplications using 254 precomputed elements ($m = 12$, $w_1 = 14$).

It is possible to use window NAF splitting with a *flexible window size*: while generating digits using the algorithm described in section 6.2, parameter w_i can be changed. This should be done only at the beginning of a new part of the window NAF (i.e., when the number of digits generated so far is a multiple of m). For example, if in the $k = 1$, $L_1 = 160$ setting we are using $m = 8$ and allowing an extra digit in the leftmost part, the (modified) window NAF will be split into 20 parts; we can start with $w_1 = 5$ for the first 12 of these, then switch to $w_1 = 4$ for the remaining 8. Then we need $12 \cdot 2^4 + 8 \cdot 2^3 = 256$ precomputed elements and can perform exponentiations in about 7.2 squarings and $\frac{12 \cdot 8}{5+2} + \frac{8 \cdot 8}{4+2} \approx 24.4$ multiplications, which is usually (depending on the relative performance of squarings and general multiplications) better than the performance of the Lim-Lee method with 254 precomputed elements.

Chapter 8

Elliptic Curve Point Multiplication with Resistance against Side-Channel Attacks

A type of public-key cryptography that is gaining much popularity is *elliptic curve cryptography*, cryptography using the group of rational points on an elliptic curve over a finite field \mathbb{F}_q . For details on the mathematics and on implementation aspects, see e.g. [12] and [41]. An elliptic curve is specified by a *curve equation* E over variables X, Y . The relevant group $E(\mathbb{F}_q)$ consists (when using the *affine representation* of points) of those pairs (X, Y) of field elements that fulfill said equation, plus the neutral element \mathcal{O} , which is called the *point at infinity*. Several styles of *projective representations* are often used for better efficiency (appropriate representations can significantly cut down the number of field inversions, and these often are rather slow).

The group operation is commutative. It is usually written additively rather than multiplicatively, and the present chapter follows this convention. The following operations arise:

- *Point addition*, i.e. computing $P + Q$ given a pair of points P, Q .
- *Point doubling*, i.e. computing $2P$ given a point P .
- *Point inversion*, i.e. computing $-P$ given a point P .

Typically, point addition algorithms have to detect the cases where one of the points P, Q is the point at infinity or where $P = -Q$ (so that the sum of P and Q is the point at infinity) or where $P = Q$ (so that the point addition is a point doubling) and treat them specially. The algorithm for the remaining case of point addition where $P, Q \neq \mathcal{O}$ and $P \neq \pm Q$ is in general not suitable for point doubling, and thus point addition and point doubling can be considered distinct operations as long as those special cases are avoided.

Point inversion is a very quick operation (it can be implemented by a field inversion in the case of a field of odd characteristic, and by a field addition in the case of a field of characteristic two), so computing $P - Q$ is essentially the same work as computing $P + Q$.

As we switch from multiplicative to additive notation, exponentiation becomes multiplication of a point P by a scalar e , or *point multiplication* for short. The methods described in the preceding chapters can be used to compute eP .

Elliptic curves used for cryptography are usually required to have a large prime-order subgroup. We denote its order by p and assume that only one order- p subgroup exists. In this setting, cryptographic protocols typically employ only points of order p . The integer $h = \#(E(\mathbb{F}_q))/p$, the *cofactor*, is typically very small, e.g. $h \leq 4$ as required by [22].

The present chapter focuses on an additional aspect for point multiplication, namely *side-channel attacks* ([48], [49]) where adversaries try to use power consumption measurements or similar observations to derive information on secret scalars e in point multiplications eP . Point P is usually not secret, and indeed may often be chosen by the adversary. Multiplier e may either be ephemeral or serve as a long-time key.

One distinguishes between *differential side-channel attacks*, which require correlated measurements from multiple point multiplications, and *simple side-channel attacks*, which directly interpret data obtained during a single point multiplication. Since usually executions of the point addition algorithm can be distinguished from executions of the point doubling algorithm, elliptic curve cryptography is particularly vulnerable to this kind of attack when side-channel information is available to an adversary unless point multiplication is implemented with appropriate countermeasures.

Various side-channel attack countermeasures for elliptic curve point multiplication have been proposed in the literature. The contributions of the present chapter are two different methods that use a uniform pattern of point doublings and point additions (avoiding the special cases listed above) in order not to reveal information on e . Both methods are very general (instead of requiring specifically chosen elliptic curves as some other methods, they work with conventional elliptic curve arithmetic implementations and can be used with standard curves such as the ones recommended by NIST and SECG in [67] and [23], whose use is often encouraged in order to ease interoperability) and provide good efficiency (unlike earlier proposals in the literature, they are 2^w -ary rather than binary). The first method is a left-to-right method based on special representations of scalars. The second method is a right-to-left method employing a special initialization stage; in contrast with the first method, it does not use an inherently fixed table, thus reducing potential information leakage available to adversaries, and it is easily parallelizable on two-processor systems, where it provides much improved performance.

Section 8.1 presents some previous side-channel attack countermeasures for elliptic curve cryptography. Section 8.2 discusses assumptions of the security model underlying the countermeasures in the present chapter and looks into certain implementation details. Section 8.3 describes the new 2^w -ary left-to-right method, and section 8.4 describes the new 2^w -ary right-to-left method. Section 8.5 provides efficiency comparisons. Section 8.6 shows an idea for scalar randomization in side-channel attack countermeasures.

8.1 Previous Side-Channel Attack Countermeasures

Randomization can be used as a countermeasure against differential side-channel attacks. In particular, for elliptic curve cryptography, *projective randomization* is a simple and effective tool [28]: if

$$(X, Y, Z)$$

represents (in *Jacobian* projective coordinates) the point whose affine coordinates are

$$(X/Z^2, Y/Z^3),$$

then another representation of the same point that cannot be predicted by the adversary is obtained by substituting

$$(r^2X, r^3Y, rZ)$$

with a randomly chosen secret non-zero field element r . When starting from an affine representation (X, Y) , this simplifies to (r^2X, r^3Y, r) . Other countermeasures have been proposed that randomize the computation without depending on the representation of group elements:

- eP can be expressed as $e(P - Q) + eQ$ where Q is a random point.
- eP can be expressed as $(e + np)P$ or $(e - n)P + nP$ where n is a random integer. (Remember that p denotes the order of the subgroup that is used. See section 8.6 for a new proposal for such scalar randomization.)

However, while these countermeasures may provide protection against differential side-channel attacks, they are not likely to provide sufficient protection if simple side-channel attacks are possible. Note that straightforward implementations of elliptic curve systems are particularly vulnerable to simple side-channel attacks: as adding and doubling require different algorithms, the bits of e may be plainly visible in a power consumption trace if a simple binary algorithm is used for point multiplication and the adversary can tell apart point doublings from general point additions. Improved point multiplication algorithms using other scalar representations (see e.g. section 6.2) will obscure e to some degree, but may still reveal plenty of information.

This vulnerability can be avoided by implementing point multiplication in a fixed sequence of point operations that does not depend on the particular scalar. Note that it is reasonable to assume that point addition and point subtraction are uniform to the adversary as point inversion is nearly immediate (dummy inversions can be inserted to obtain the same sequence of operations for point additions as for point subtractions).

Various point multiplication methods have been proposed that use an alternating sequence of doublings and additions. The simplest approach is to use a binary point multiplication method with dummy additions inserted to avoid dependencies on scalar bits [28]; however as we will see in section 8.2.2, it may be easy for adversaries to determine which additions are dummy operations, so it is not clear that this method provides sufficient security. For odd scalars, a variant of binary point multiplication can be used where the scalar is represented in balanced binary representation (digits -1 and $+1$ without any digits of value zero other than leading zeros) [86]. Also Montgomery's binary point multiplication method [65], which maintains an invariant $Q_1 - Q_0 = P$ while computing eP using two variables Q_0, Q_1 , can be adapted for implementing point multiplication with a fixed sequence of point operations ([87], [69], [10], [43], [36]). With this approach, specific techniques can be used to speed up point arithmetic: the doubling and addition steps can be combined; y -coordinates of points may be omitted during the computation ([65], [10], [43], [36]); and on suitable hardware, parallel execution can be conveniently used for improved efficiency ([43], [36]).

All of the above point multiplication methods are binary. Given sufficient memory, efficiency can be improved by using the 2^w -ary point multiplication methods proposed in sections 8.3 and 8.4. In these methods, the uniform operation pattern in the main loop of the point multiplication algorithm has one addition after each w doublings.

Multiple side-channel attack countermeasures can be combined; the new point multiplication methods proposed in the current chapter can be used together with one or multiple of the randomization methods mentioned above. In particular, randomized projective coordinates

should be used (cf. [70]) as noted in the following method descriptions. If the scalar e is a long-time key, it should be randomized as described in section 8.6.

8.2 Security against Side-Channel Attacks

The goal to achieve security against side-channel attacks warrants some notes on the model to be used for the security analysis. While a complete model of all side-channel information is hardly ever attainable (and, in any case, would be closely tied to a specific implementation), it is possible to describe the workings of the point multiplication algorithm in enough detail to obtain reasonable assurance that information leakage will not be useful to an adversary. Before looking at the actual point multiplication methods in sections 8.3 and 8.4, we discuss desired features and how to achieve them. The most prominent item to consider are elliptic curve point operations, where certain special cases should be avoided; this is discussed in section 8.2.1. At a lower level, we have the individual field operations used to implement point operations. Section 8.2.2 shows that point multiplication implementations intended to be secure against side-channel attacks should use randomized projective coordinates, and that left-to-right point multiplication methods with a fixed table should use certain extended point representations. That section also explains why inserting dummy point additions to achieve uniform behavior appears questionable.

8.2.1 Elliptic Curve Point Operations

As different algorithms are usually required for point doubling and point addition, we assume that side-channel data reveals the sequence in which these operations take place. Thus the point multiplication algorithm should use doublings and additions in a uniform pattern independent of the specific multiplier. We also have to take into account certain special cases that must be expected to be visible through side channels when they occur (if conventional implementations of the point operations are employed):

- Point doubling $2P$ requires conditional statements for the case that P is the point at infinity or that P is a point of order two. If these cases are avoided, then, expressed in field operations, point doubling runs as a fixed routine.
- As mentioned before, point addition $P + Q$ requires conditional statements for the case that one of the points is the point at infinity, or that $P = -Q$, or that $P = Q$. For other cases, it too can be implemented as a fixed routine.

Note that these special cases do also apply when projective coordinates are used (for example, an addition $P + Q$ invokes a special case when P coincides with Q even if the projective representations differ). Details of the sequences of field operations used for point doubling and point addition depend on the underlying field (odd characteristic or characteristic 2) and the choice of point representations (e.g. either affine coordinates or one of multiple styles of projective coordinates, cf. [27]), so implementations may vary widely. The essential observation is that the respective algorithm always behaves the same as long as the above special cases are avoided.

8.2.2 Field Operations

While it is reasonable to assume that side-channel observations for, say, a field multiplication do not immediately reveal the factors involved, it would not be prudent to assume that all multiplications look the same to an adversary. This is why projective randomization is useful ([28], [70]): if, e.g., Jacobian projective coordinates are used, a representation (X, Y, Z) of the point with affine coordinates $(X/Z^2, Y/Z^3)$ can be replaced by (r^2X, r^3Y, rZ) for any field element $r \neq 0$. If r is secretly chosen at random, then, provided that $X \neq 0$ and $Y \neq 0$, none of the coordinates of the new representation will be predictable to the adversary. Point doubling or point addition using projective coordinates results in a point represented with a Z -coordinate that is the product of the Z -coordinate(s) of the input point(s) and a short polynomial involving one or more other coordinates of the input points; thus the output point is again in a randomized representation.

However, for a point with $X = 0$ or $Y = 0$, every equivalent projective representation will have an according zero coordinate. An adversary who can choose the input point P to a point multiplication algorithm may be able to exploit the existence of such special points by selecting P such that intermediate points during the point multiplication will have a zero coordinate if a guess for part of the scalar e is correct [39]. Assuming that the zero coordinate becomes visible through side channels, then if e is constant for many point multiplications, this can be used to determine all of e . Similar attacks can be possible based on intermediate field elements that appear during individual point operations [3]. A countermeasure for curves where such attacks are possible is to randomize, if point P may be chosen by the adversary, the point rather than just its projective representation, i.e. compute eP as

$$e(P + Q) - eQ$$

for a random point Q . The points Q and eQ can be stored in memory constantly (Q must be kept secret). As the full attacks require a constant e , they can also be countered by scalar randomization (see section 8.6).

When randomized projective coordinates are used to prevent the adversary from guessing the specific inputs to field operations (and thus from directly verifying, for example, which of several known points are currently being added), the adversary may still be able to recognize if the same field operation with the same input values is performed multiple times. Even if these values are not known, this may leak essential information, so it should be avoided. We now look what should be taken care of.

Many left-to-right algorithms for point multiplication eP , including the one that will be presented in section 8.3, can be outlined as follows (cf. section 6.1.1):

[Precomputation stage.] First, independently of the specific multiplier e , certain small multiples of P are computed and stored in a table.

[Evaluation stage.] Second, the product eP is evaluated as follows: a variable A is initialized to one of the table values; then, many times, A either is doubled or a table value is added to A , replacing the previous value of A . Finally, A contains the result eP .

Unless a projective randomization of the table value that has been used is performed whenever the table is accessed (which will avoid a fixed table at the price of reduced efficiency), the use of a fixed table may allow for statistical attacks. For such algorithms with a fixed table, each entry of the precomputed table should contain not only the representation of the point, but

additionally any field elements that would have to be computed each time the point is added to A . If Jacobian projective coordinates are used, this means that (X, Y, Z, Z^2, Z^3) should be stored (*Chudnovsky Jacobian* coordinates, see [18]) rather than just (X, Y, Z) (cf. the addition formulas in [12, Chapter IV.2] or [41]). If such *extended point representations* are not used for the precomputed table, the adversary may be able to guess easily which point additions involve the same table value.

There are point multiplication algorithms that can achieve a fixed pattern of doublings and additions only if dummy additions are inserted into the evaluation stage. That is, sometimes a table value is added to A , but the result is thrown away and A is left unchanged. The problem with dummy additions is that the adversary may be able to tell that their result is ignored: for Jacobian projective coordinates, each point operation requires squaring the Z -coordinate of A ; if A is not changed, then two consecutive point operations involve the very same squaring. The point multiplication algorithms presented in sections 8.3 and 8.4 achieve uniform behavior without resorting to dummy additions.

8.3 2^w -Ary Left-to-Right Method

Left-to-right point multiplication algorithms look at the digits of some representation of the scalar starting at the most significant position and proceed down to the least significant one (see section 6.1.1). A simple way to obtain a uniform sequence of doublings and additions (namely, one addition after each w doublings in the main loop of the point multiplication algorithm) in left-to-right point multiplication is to use a 2^w -ary scalar encoding and to insert a dummy addition whenever a zero digit is encountered. However, as we have seen in section 8.2.2, using dummy additions to achieve uniform behavior is questionable because it may be visible to the adversary which additions are dummy additions. Here we will look at a modified 2^w -ary left-to-right point multiplication method that does not need dummy additions for uniform behavior.

This method may fail in some cases in that an addition step may turn out to be a point doubling or may involve the point at infinity (which both requires special treatment and is potentially clearly visible through side channels). However, we will show that the probability of this happening is close to zero if multipliers are appropriately selected: randomly chosen e is safe.

Section 8.3.1 describes how to encode the multiplier e into special signed-digit representations. Section 8.3.2 discusses the multiplication algorithms implied by these representations. Section 8.3.3 shows that, unless e is ill-chosen, this point multiplication algorithm indeed limits information leakage as intended.

8.3.1 Recoding Algorithms

Let the positive integer e be given in 2^w -ary digits where $w \geq 2$ is a small integer; namely,

$$e = \sum_{i=0}^{\ell'} b'_i \cdot 2^{wi}$$

with $b'_i \in \{0, 1, \dots, 2^w - 1\}$. We demand that ℓ' be chosen minimal, i.e. $b'_{\ell'} \neq 0$. For $i > \ell'$, we define that $b'_i = 0$.

The first (basic) recoding algorithm converts this into a representation

$$e = \sum_{i=0}^{\ell} b_i \cdot 2^{wi}$$

such that

$$b_i \in \{-2^w, 1, 2, \dots, 2^w - 1\}.$$

This means that we disallow digit value 0 and instead introduce the new value -2^w . Intuitively, the recoding algorithm replaces 0 digits by -2^w and increments the more significant neighbor digit to adjust the value. A slightly more complicated variant of the multiplier recoding algorithm produces digits

$$b_i \in \{-2^w, \pm 1, \pm 2, \dots, \pm(2^{w-1} - 1), 2^{w-1}\}$$

instead. Note that the set of possible digit values still has 2^w elements. As we will see in section 8.3.2, this alternative representation leads to improved efficiency.

It is easy to see that for both recoding techniques, b_ℓ is necessarily positive (otherwise e would be negative) and that the representation of e needs to grow by at most one digit, i.e. $\ell = \ell'$ or $\ell = \ell' + 1$.

We express the recoding algorithms recursively, using auxiliary values c_i and t_i such that $0 \leq c_i \leq 2$ and $0 \leq t_i \leq 2^w + 1$: let

$$c_0 = 0,$$

and for $i = 0, \dots, \ell' + 1$, let

$$t_i = b'_i + c_i$$

where for the basic recoding algorithm we have

$$(c_{i+1}, b_i) = \begin{cases} (1, -2^w) & \text{if } t_i = 0 \\ (0, t_i) & \text{if } 0 < t_i < 2^w \\ (2, -2^w) & \text{if } t_i = 2^w \\ (1, 1) & \text{if } t_i = 2^w + 1 \end{cases}$$

and for the variant we have

$$(c_{i+1}, b_i) = \begin{cases} (1, -2^w) & \text{if } t_i = 0 \\ (0, t_i) & \text{if } 0 < t_i \leq 2^{w-1} \\ (1, -2^w + t_i) & \text{if } 2^{w-1} < t_i < 2^w \\ (2, -2^w) & \text{if } t_i = 2^w \\ (1, 1) & \text{if } t_i = 2^w + 1. \end{cases}$$

Note that $c_{i+1} \cdot 2^w + b_i = t_i$ always holds.

If $b_{\ell'+1} \neq -2^w$, then

$$e = \sum_{i=0}^{\ell'+1} b_i \cdot 2^{wi};$$

in this case, we set $\ell = \ell' + 1$. Otherwise, we have

$$e = \sum_{i=0}^{\ell'} b_i \cdot 2^{wi}$$

and $b'_\ell \neq -2^w$, and we set $\ell = \ell'$.

Observe that if $b_\ell = 1$ (and $\ell > 0$), then $b_{\ell-1} \neq -2^w$.

As a countermeasure against side-channel attacks on the recoding algorithm itself, assignments can be implemented by table lookups instead of using conditional statements.

Storing the converted representation of e requires almost no additional memory compared with the original representation: digit values can be stored modulo 2^w . If w is understood, this new representation can be stored as an ordinary integer (leading 0 digits can simply be ignored because b_ℓ is never -2^w). This integer is at most two bits longer than e ; the maximum possible length growth occurs if $\ell = \ell' + 1$ and $b_\ell = 2$.

It may be desirable to ensure that the encoded form has a predetermined maximum index ℓ . For our point multiplication application, if ℓw is large enough compared with the binary length of group order p , this is easy to do: if necessary, adjust e by adding p or a multiple of p .

If a random multiplier is required and a uniform distribution is not essential, then random bits can be used directly to generate the recoded form of e . In this case, too, it should be ensured that $b_{\ell-1} \neq -2^w$ if $b_\ell = 1$.

8.3.2 Point Multiplication Algorithm

Remember that we want to compute eP where point P should have order p , p being a large prime that divides the order of the elliptic curve $E(\mathbb{F}_q)$. For our point multiplication algorithm to work as intended, $\text{ord}(P)$ may be any positive multiple of p .

If point P can be chosen by the adversary, we must be aware that it might have incorrect order. In case that $\#E(\mathbb{F}_q) = p$, then (besides testing that P is indeed a point on the curve) we just have to verify that P is not the point at infinity, \mathcal{O} . Otherwise, we need an additional sanity check. Let h be the cofactor of the elliptic curve, i.e. the integer $\#(E(\mathbb{F}_q))/p$. Curves used for cryptography are usually selected such that h is very small ([22] requires $h \leq 4$). Thus hP can be computed quickly; if it is not \mathcal{O} , then we know that p divides $\text{ord}(P)$.

If P has incorrect order, computing eP must be rejected. Otherwise, given a representation

$$e = \sum_{i=0}^{\ell} b_i \cdot 2^{wi}$$

of e as determined by one of the recoding algorithms from section 8.3.1, eP can be computed by the procedure shown in algorithm 8.1 in the case of the basic recoding algorithm or algorithm 8.2 in the case of the variant. For both cases, what is shown is a high-level description of the algorithm: as explained in section 8.2.2, implementations should use randomized projective coordinates, and values computed in the precomputation stage should be stored in extended point representation. The algorithm for the basic case requires $2^{w-1} + \ell w$ point doublings and $2^{w-1} - 1 + \ell$ point additions; the algorithm for the variant, which makes extended use of the ease of inversion in elliptic curve groups, requires $2^{w-2} + 1 + \ell w$ point doublings and $2^{w-2} - 1 + \ell$ point additions. (When comparing the efficiency, note that for the variant a larger choice of w might be preferable.)

The obvious way to implement $A \leftarrow 2^w A$ in this procedure is w -fold iteration of the statement $A \leftarrow 2A$, but depending on the elliptic curve, more efficient specific algorithms for w -fold point doubling may be available (see [42]).

During the procedure shown in algorithm 8.1 or algorithm 8.2, a projective randomization should be performed twice. First, if the precomputed table of multiples of P is stored in

Algorithm 8.1 Compute $(\sum_{i=0}^{\ell} b_i \cdot 2^{wi})P$ where $b_i \in \{-2^w, 1, 2, \dots, 2^w - 1\}$

{Precomputation stage}
 $P_1 \leftarrow P$
for $n = 2$ to $2^w - 2$ **step 2 do**
 $P_n \leftarrow 2P_{n/2}$
 $P_{n+1} \leftarrow P_n + P$
 {now $P_n = nP, P_{n+1} = (n+1)P$ }
 $P_{-2^w} \leftarrow -2P_{2^w-1}$
 {now $P_{-2^w} = (-2^w)P$ }

{Evaluation stage}
 $A \leftarrow P_{b_\ell}$
for $j = \ell - 1$ **down to 0 do**
 $A \leftarrow 2^w A$
 $A \leftarrow A + P_{b_j}$
return A

projective coordinates, the representation of P should be randomized before the table is computed (if the table of multiples of P is stored in affine coordinates to speed up the evaluation stage by using mixed addition of affine and projective points [27], this first randomization obviously is not necessary; but using such fixed and predictable table entries may help differential side-channel attacks and thus cannot be recommended). Second, at the beginning of the evaluation stage, after the initial value has been assigned to A , the representation of A should be randomized. For best assurance against statistical attacks, a projective randomization of the table value that has just been used can be performed whenever the evaluation stage accesses the precomputed table. However, unless e is constant over many point multiplications, this should not be necessary; see section 8.6 for how a constant e can be avoided.

8.3.3 Uniformity of the Point Multiplication Algorithm

It is apparent that algorithms 8.1 and 8.2 use doublings and additions in a regular pattern. As discussed in section 8.2, to show that the procedure has uniform behavior and thus is suitable for limiting information leakage during the computation of eP , we also have to show that the following special cases of point doubling and point addition can be avoided:

- $2\mathcal{O}$
- $2A$ where $\text{ord}(A) = 2$
- $A + \mathcal{O}$ or $\mathcal{O} + B$
- $A + A$
- $A + (-A)$

We have required that $\text{ord}(P)$ be a positive multiple of p . Without loss of generality, here we may assume that $\text{ord}(P) = p$. (Otherwise, multiply P by the cofactor $h = \#(E(\mathbb{F}_q))/p$)

Algorithm 8.2 Compute $(\sum_{i=0}^{\ell} b_i \cdot 2^{wi})P$ where $b_i \in \{-2^w, \pm 1, \pm 2, \dots, \pm(2^{w-1} - 1), 2^w - 1\}$

{Precomputation stage}

$P_1 \leftarrow P$

for $n = 2$ to $2^{w-1} - 2$ **step 2 do**

$P_n \leftarrow 2P_{n/2}$

$P_{n+1} \leftarrow P_n + P$

$P_{2^{w-1}} \leftarrow 2P_{2^{w-2}}$

$P_{2^w} \leftarrow 2P_{2^{w-1}}$

{Evaluation stage}

if $b_\ell > 0$ **then**

$A \leftarrow P_{b_\ell}$

else

$A \leftarrow -P_{|b_\ell|}$

for $j = \ell - 1$ **down to** 0 **do**

$A \leftarrow 2^w A$

if $b_j > 0$ **then**

$A \leftarrow A + P_{b_j}$

else

$A \leftarrow A - P_{|b_j|}$

return A

to obtain a point of order p . In the algorithm performed for P , the above special cases can occur only when one of them would occur in the algorithm performed for hP ; in particular, $\text{ord}(A) = 2$ would imply $hA = \mathcal{O}$, so points of order 2 are not an issue if we can show that the point at infinity can be avoided.) Then, as 2^w can be expected to be much smaller than p , all precomputed points $P_{b_j} = b_j P$ in algorithms 8.1 and 8.2 will be of order p . Thus, initially we have $A \neq \mathcal{O}$; and as long as we avoid additions of the form $A + (-A)$, it will remain like this. So what is left to be checked is whether in the evaluation stage of algorithms 8.1 and 8.2 we can avoid that $A = b_j P$ or $A = -b_j P$ before an addition or subtraction takes place.

With

$$e_j = \sum_{i=j}^{\ell} b_i \cdot 2^{w(i-j)},$$

the point addition step in the evaluation stage loop of algorithm 8.1 or 8.2 performs the point addition $(2^w \cdot e_{j+1})P + b_j P$. Thus we are safe if

$$2^w \cdot e_{j+1} \not\equiv \pm b_j \pmod{p}.$$

Since $|e_j| \leq 2^{(1+\ell-j)w}$ and $|b_j| \leq 2^w$, for many indices j we have $|2^w \cdot e_{j+1}| + |b_j| < p$, meaning that reduction modulo p does not matter and it suffices to check whether $2^w \cdot e_{j+1} \neq \pm b_j$.

The largest index to consider is $j = \ell - 1$; for this case, the question is whether we can be sure that $2^w \cdot e_\ell \neq \pm b_{\ell-1}$. Indeed we can, as $e_\ell = b_\ell$ and if $b_\ell = 1$, then $b_{\ell-1} \neq -2^w$ (see section 8.3.1). It follows that $e_j \geq 2^{(\ell-j)w}$, which shows that the incongruence is satisfied for indices $j < \ell - 1$ too as long as we do not have to consider reduction modulo p .

For indices j so small such that this modular reduction is relevant, we argue that if e is sufficiently random, then the probability of picking an e such that the above incongruence

is not satisfied can be neglected: it is comparable to the probability that $e \bmod p$ can be guessed in a modest number of attempts, which can be presumed to be practically impossible (otherwise, side-channel attacks should not be a concern since e could not really be considered sufficiently secret in the first place).

8.4 2^w -Ary Right-to-Left Method

Right-to-left point multiplication algorithms look at the digits of some representation of the scalar starting at the least significant position and proceed up to the most significant one (see section 6.1.2). A regular sequence of doublings and additions (with one addition after each w doublings in the main loop of the point multiplication algorithm) can be obtained in right-to-left point multiplication by using a 2^w -ary scalar encoding and treating 0 like any other digit value. However, this approach is not fully sufficient for obtaining uniform behavior because the special cases of point addition (see section 8.2.1) would reveal information on the scalars.

The 2^w -ary right-to-left point multiplication method presented in the following is based on the observation that a special randomized initialization stage can be used to avoid the problematic special cases. The right-to-left method is easily parallelizable and can provide much improved performance on two-processor systems.

The method for computing eP is parameterized by an integer $w \geq 2$ and a digit set B consisting of 2^w integers of small absolute value such that every positive scalar e can be represented in the form

$$e = \sum_{0 \leq i \leq \ell} b_i 2^{wi}$$

using digits $b_i \in B$; for example

$$B = \{0, 1, \dots, 2^w - 1\}$$

or

$$B = \{-2^{w-1}, \dots, 2^{w-1} - 1\}.$$

A representation of e using the latter digit set can be easily determined on the fly when scanning the binary digits of e in right-to-left direction. If e is at most n bits long (i.e. $0 < e < 2^n$), $\ell = \lfloor n/w \rfloor$ is sufficient.

Let B' denote the set $\{|b| \mid b \in B\}$ of absolute values of digits, which has at least $2^{w-1} + 1$ and at most 2^w elements. The point multiplication method uses $\#(B') + 1$ variables for storing points on the elliptic curve in projective representation: one variable A_b for each $b \in B'$, and one additional variable Q .

The method works in three stages, which we call *initialization stage*, *right-to-left stage*, and *result stage*. We will first look at a high-level view of these stages before discussing the details. Let A_b^{init} denote the value of A_b at the end of the initialization stage, and let A_b^{sum} denote the value of A_b at the end of the right-to-left stage.

The *initialization stage* sets up the variables A_b ($b \in B'$) in a randomized way such that $A_b^{\text{init}} \neq \mathcal{O}$ for each b , but

$$\sum_{b \in B'} b A_b^{\text{init}} = \mathcal{O}.$$

Then the *right-to-left stage* performs computations depending on P and the digits b_i , yielding new values A_b^{sum} of the variables A_b satisfying

$$A_b^{\text{sum}} = A_b^{\text{init}} + \sum_{\substack{0 \leq i \leq \ell \\ b_i = b}} 2^{wi} P - \sum_{\substack{0 \leq i \leq \ell \\ b_i = -b}} 2^{wi} P$$

for each $b \in B'$. Finally, the *result stage* computes

$$\sum_{b \in B' \setminus \{0\}} b A_b^{\text{sum}},$$

which yields the final result eP because

$$\begin{aligned} \sum_{b \in B' \setminus \{0\}} b A_b^{\text{sum}} &= \underbrace{\sum_{b \in B' \setminus \{0\}} b A_b^{\text{init}}}_{\mathcal{O}} + \sum_{b \in B' \setminus \{0\}} b \left(\sum_{\substack{0 \leq i \leq \ell \\ b_i = b}} 2^{wi} P - \sum_{\substack{0 \leq i \leq \ell \\ b_i = -b}} 2^{wi} P \right) \\ &= \sum_{0 \leq i \leq \ell} b_i 2^{wi} P = eP. \end{aligned}$$

Section 8.4.1 discusses the initialization stage, section 8.4.2 the right-to-left stage, and section 8.4.3 the result stage. Section 8.4.4 presents some variants.

8.4.1 Initialization Stage

The initialization stage can be implemented as follows:

1. For each $b \in B' \setminus \{1\}$, generate a random point on the elliptic curve and store it in variable A_b .
2. Compute the point $-\sum_{b \in B' \setminus \{0,1\}} b A_b$ and store it in variable A_1 .
3. For each $b \in B'$, perform a projective randomization of variable A_b .

The resulting values of the variables A_b are denoted by A_b^{init} .

If the elliptic curve is fixed, precomputation can be used to speed up the initialization stage: run steps 1 and 2 just once, e.g. during personalization of a smart card, and store the resulting intermediate values A_b for future use. We denote these values by A_b^{fix} . Then only step 3 (projective randomization of the values A_b^{fix} to obtain new representations A_b^{init}) has to be performed anew each time the initialization stage is called for. The points A_b^{fix} must not be revealed; they should be protected like secret keys.

Generating a random point on an elliptic curve is straightforward. For each element X of the underlying field, there are zero, one or two values Y such that (X, Y) is the affine representation of a point on the elliptic curve. Given a random candidate value X , it is possible to compute an appropriate Y if one exists; the probability for this is approximately 1/2 by Hasse's theorem. If there is no appropriate Y , one can simply start again with a new X .

Computing an appropriate Y given X involves solving a quadratic equation, which usually (depending on the underlying field) is computationally expensive. This makes it worthwhile to use precomputation as explained above. It is also possible to reuse the values that have

remained in the variables A_b , $b \neq 1$, after a previous computation, and start at step 2 of the initialization stage.

To determine $-\sum_{b \in B' \setminus \{0,1\}} bA_b$ in step 2, it is not necessary to compute all the individual products bA_b . Algorithm 8.3 can be used instead to set up A_1 appropriately if $B' = \{0, 1, \dots, \beta\}$, $\beta \geq 2$. (Note that both loops will be skipped in the case $\beta = 2$.) This

Algorithm 8.3 Compute $A_1 \leftarrow -\sum_{b \in \{2, \dots, \beta\}} bA_b$ in the initialization stage

for $i = \beta - 1$ down to 2 **do**

$A_i \leftarrow A_i + A_{i+1}$

$A_1 \leftarrow 2A_2$

for $i = 2$ to $\beta - 1$ **do**

$A_i \leftarrow A_i - A_{i+1}$

$A_1 \leftarrow A_1 + A_{i+1}$

$A_1 \leftarrow -A_1$

algorithm takes one point doubling and $3\beta - 6$ point additions. When it has finished, the variables A_b for $1 < b < \beta$ will contain modified values, which are representations of the points originally stored in the respective variables. If sufficient memory is available, a faster algorithm can be used to compute A_1 without intermediate modification of the variables A_b for $b > 1$ (use additional variables Q_b instead; in this case, see section 8.4.3 for a possible additional improvement if point doublings are faster than point additions).

The projective randomization of the variables A_b ($b \in B'$) in step 3 has the purpose to prevent adversaries from correlating observations made during the computation of A_1 in the initialization stage with observations from the following right-to-left stage. If algorithm 8.3 has been used to compute A_1 and the points are not reused for multiple invocations of the initialization stage, then no explicit projective randomization of the variables A_b for $1 < b < \beta$ is necessary; and if $\beta > 2$, no explicit projective randomization of A_1 is necessary: the variables have automatically been converted into new representations by the point additions used to determine their final values.

8.4.2 Right-to-Left Stage

Algorithm 8.4 implements the right-to-left stage using a uniform pattern of point doublings and point additions. Initially, for each b , variable A_b contains the value A_b^{init} ; the final value is denoted by A_b^{sum} . Due to special cases that must be handled in the point addition algorithm

Algorithm 8.4 Right-to-left stage

$Q \leftarrow P$

for $i = 0$ to ℓ **do**

if $b_i \geq 0$ **then**

$A_{b_i} \leftarrow A_{b_i} + Q$

else

$A_{|b_i|} \leftarrow -((-A_{|b_i|}) + Q)$

$Q \leftarrow 2^w Q$

(see section 8.2.1), uniformity of this algorithm is violated if $A_{|b_i|}$ is a projective representation of $\pm Q$; the randomization in the initialization stage ensures that the probability of this is

close to zero. This is why in section 8.4.1 we required that precomputed values A_b^{fix} be kept secret: adversaries must not be able to choose P depending on the values A_b^{fix} , or they might be able to trigger the special cases.

If B contains no negative digits, the corresponding branch in the algorithm can be omitted. Otherwise, implementations should use dummy point inversions to achieve uniform behavior for the two conditional branches.¹

The obvious way to implement $Q \leftarrow 2^w Q$ in this algorithm is w -fold iteration of the statement $Q \leftarrow 2Q$, but depending on the elliptic curve, more efficient specific algorithms for w -fold point doubling may be available (see [42]).

In the final iteration of the loop, the assignment to Q may be skipped (the value Q is not used after the right-to-left stage has finished). With this modification, the algorithm uses ℓw point doublings and $\ell + 1$ point additions.

Observe that on two-processor systems the point addition and the w -fold point doubling in the body of the loop may be performed in parallel: neither operations depends on the other's result.

8.4.3 Result Stage

Similarly to the computation of A_1 in the initialization stage, the result stage computation

$$\sum_{b \in B' \setminus \{0\}} b A_b^{\text{sum}}$$

can be performed without computing all the individual products $b A_b^{\text{sum}}$. In the result stage, it is not necessary to preserve the original values of the variables A_b , so algorithm 8.5 (from [46, answer to exercise 4.6.3-9]) can be used if $B' = \{0, 1, \dots, \beta\}$ when initially each variable A_b contains the value A_b^{sum} . This algorithm uses $2\beta - 2$ point additions.

Algorithm 8.5 Compute $\sum_{b \in \{1, \dots, \beta\}} b A_b^{\text{sum}}$ when initially $A_b = A_b^{\text{sum}}$

for $i = \beta - 1$ **down to** 1 **do**

$A_i \leftarrow A_i + A_{i+1}$

for $i = 2$ **to** β **do**

$A_1 \leftarrow A_1 + A_i$

return A_1

Elliptic curve point arithmetic usually has the property that point doublings are faster than point additions. Then the variant given in algorithm 8.6 is advantageous. This algorithm uses $\lfloor \beta/2 \rfloor$ point doublings and $2\beta - 2 - \lfloor \beta/2 \rfloor$ point additions.

8.4.4 Variants

In the following, we discuss some variations of the right-to-left point multiplication method.

¹In the publication [61], the second branch was written as $A_{|b_i|} \leftarrow A_{|b_i|} - Q$. Katsuyuki Okeya has pointed out the problem that then if adversaries are able to predict the representations of Q and $-Q$, they may be able to distinguish between the two conditional branches even if dummy point inversions are used to let point additions look like point subtractions. Note that this problem is avoided when a projective randomization of P is performed before beginning the right-to-left stage.

Algorithm 8.6 Compute $\sum_{b \in \{1, \dots, \beta\}} bA_b^{\text{sum}}$ when initially $A_b = A_b^{\text{sum}}$ (variant)

```

for  $i = \beta$  down to 1 do
  if  $2i \leq \beta$  then
     $A_i \leftarrow A_i + A_{2i}$ 
  if  $i$  is even then
    if  $i < \beta$  then
       $A_i \leftarrow A_i + A_{i+1}$ 
     $A_i \leftarrow 2A_i$ 
  else
    if  $i > 1$  then
       $A_1 \leftarrow A_1 + A_i$ 
return  $A_1$ 

```

Projective Randomization of P

While it does not appear to be strictly necessary, it can be recommended to perform a projective randomization of P before beginning the right-to-left stage (algorithm 8.4). At small computational cost, this will further reduce the side-channel information available to potential adversaries.

Avoiding Digit 0

In the point multiplication method as described in sections 8.4.2 and 8.4.3, if $0 \in B$, the variable A_0 is essentially a dummy variable: its value does not affect the final result. Now assume that an adversary is performing a fault attack [13] by purposefully inducing computation faults. If these faults occur only during computations affecting A_0 , the result of the point multiplication will still be correct. Thus, verifying the result cannot reveal that a fault attack has taken place. Therefore it may be useful to avoid the dummy variable.

The point multiplication method can be used with a digit set B that does not include the value 0, e.g.

$$B = \{-2^w, \pm 1, \pm 2, \dots, \pm(2^w - 2), 2^w - 1\}$$

as in the variant in section 8.3.1. Compared with digit set $\{-2^{w-1}, \dots, 2^{w-1} - 1\}$, this requires modifications to the algorithms used in step 2 of the initialization stage (section 8.4.1) and in the result stage (section 8.4.3). If we assume that the initialization stage uses precomputed points A_b^{fix} , only the changes to the result stage will increase the computational cost of a point multiplication. The result stage for said digit set has to compute the sum

$$\sum_{b \in \{1, \dots, 2^{w-1}, 2^w\}} bA_b^{\text{sum}};$$

the additional cost is one point doubling and one point addition (set $A_{2^{w-1}} \leftarrow A_{2^{w-1}} + 2A_{2^w}$ before running algorithm 8.5 or 8.6).

Variant for $w = 1$

The right-to-left point multiplication method as described before works only for $w \geq 2$ because of the requirement that $A_b^{\text{init}} \neq \mathcal{O}$ for each $b \in B'$, but $\sum_{b \in B'} bA_b^{\text{init}} = \mathcal{O}$. The method can be

adapted to the case $w = 1$ with

$$B = \{0, 1\}$$

by relinquishing the latter part of the requirement; instead, save the value A_1^{init} and compute $A_1^{\text{sum}} - A_1^{\text{init}}$ in the result stage.² If A_1^{init} is just a projective randomization of a precomputed random point A_1^{fix} , there is no need to save A_1^{init} : the result stage can simply compute $A_1^{\text{sum}} - A_1^{\text{fix}}$.

Application to Modular Exponentiation

The right-to-left method with a randomized initialization stage can similarly be used for modular exponentiation. For this purpose, digit set B will only contain non-negative digits as modular inversion is an expensive operation.

8.5 Efficiency Comparison

For elliptic curve cryptography over prime fields using Jacobian projective coordinates, a point addition can be done in 16 field multiplications (here we count squarings as multiplications), and curves are usually chosen such that a point doubling can be done in 8 field multiplications [41]. The cost for a projective randomization, transforming (X, Y, Z) into (r^2X, r^3Y, rZ) , is 5 field multiplications. Converting a point (X, Y, Z) into extended point representation (X, Y, Z, Z^2, Z^3) , Chudnovsky Jacobian coordinates, takes 2 field multiplications. A point addition costs 14 field multiplications if one of the points to be added is given in Chudnovsky Jacobian coordinates and the result is needed in Jacobian coordinates, or if both points are given in Chudnovsky Jacobian coordinates and the result is needed in Chudnovsky Jacobian coordinates as well; a point doubling in Chudnovsky Jacobian coordinates representation costs 9 field multiplications [18].

We first examine the efficiency of the algorithms for performing a point multiplication eP in a small configuration with $w = 2$.

The 2^w -ary left-to-right method from section 8.3 with digit set $\{-4, -1, 1, 2\}$ (three points $P, 2P, 4P$ to precompute) uses one projective randomization followed by two point doublings for precomputation and then one projective randomization, 2ℓ point doublings, and ℓ point additions for determining the result.

Now one possibility is to use Chudnovsky Jacobian coordinates for the precomputed table. Then the cost for the precomputation stage is $2 \cdot 9 + 5 + 2 = 25$ field multiplications, the cost for the evaluation stage is $\ell \cdot 14 + 2\ell \cdot 8 + 5 = 30\ell + 5$ field multiplications, and the total cost is

$$30\ell + 30$$

field multiplications. Alternatively, to avoid concerns about keeping the table constant during one point multiplication, we can use usual Jacobian coordinates for the precomputed table and perform a projective randomization of the table value used after each except the very last point addition. Then the cost for the precomputation stage is $2 \cdot 8 + 5 = 21$, the cost for the evaluation stage is $\ell \cdot 16 + 2\ell \cdot 8 + \ell \cdot 5 = 37 \cdot \ell$, and the total cost becomes

$$37\ell + 21$$

²This variant was suggested by Tsuyoshi Takagi.

field multiplications. Assuming 160-bit scalars ($\ell = 80$), we have 2430 field multiplications for the first implementation choice or 2981 for the second.

For the 2^w -ary right-to-left method from section 8.4 with digit set $B = \{-2, -1, 0, 1\}$ (i.e. four variables A_0, A_1, A_2, Q), we assume that points $A_0^{\text{fix}}, A_1^{\text{fix}}, A_2^{\text{fix}}$ such that $A_1^{\text{fix}} = -2A_2^{\text{fix}}$ have been precomputed since generating a random point on the curve would be rather expensive. In this scenario, the initialization stage has to perform three projective randomizations, i.e. 15 field multiplications; the right-to-left stage uses 2ℓ point doublings and $\ell + 1$ point additions, i.e. $(\ell + 1) \cdot 16 + 2\ell \cdot 8 = 32\ell + 16$ field multiplications; and the result stage can be implemented in one point doubling and one point addition, i.e. $16 + 8 = 24$ field multiplications. The total cost is

$$32\ell + 55$$

field multiplications; assuming 160-bit scalars ($\ell = 80$), we have 2615 field multiplications.

With two processors, in the loop of the right-to-left stage, the two point doublings ($2 \cdot 8 = 16$ field multiplications) can be performed in parallel with the one point addition (also 16 field multiplications), and so we can remove 16ℓ field multiplications from the tally. (We ignore the small additional savings that can be achieved through parallelization in the other stages.) Only

$$16\ell + 55$$

field multiplications remain; for 160-bit scalars, this is 1335 field multiplications.

Now we consider similar scenarios with arbitrary window sizes $w \geq 2$ and arbitrary scalar bit lengths n . The left-to-right method from section 8.3 (again using the efficiency-improving variant of the recoding algorithm) with Chudnovsky Jacobian coordinates for the precomputed table uses one projective randomization, one conversion into Chudnovsky Jacobian coordinates, $2^{w-2} + 1$ point doublings, and $2^{w-2} - 1$ point additions for precomputation, i.e. $5 + 2 + (2^{w-2} + 1) \cdot 9 + (2^{w-2} - 1) \cdot 14 = 2^{w-2} \cdot 23 + 2$ field multiplications; and one projective randomization, $\lfloor n/w \rfloor \cdot w$ point doublings, and $\lfloor n/w \rfloor$ point additions for computing the result, i.e. $5 + \lfloor n/w \rfloor \cdot w \cdot 8 + \lfloor n/w \rfloor \cdot 14 = \lfloor n/w \rfloor \cdot (w \cdot 8 + 14) + 5$ field multiplications. The total cost of this is

$$\left\lfloor \frac{n}{w} \right\rfloor \cdot (w \cdot 8 + 14) + 2^{w-2} \cdot 23 + 7$$

field multiplications. The left-to-right method with additional projective randomizations to avoid a fixed table uses one projective randomization, $2^{w-2} + 1$ point doublings, and $2^{w-2} - 1$ point additions for precomputation, i.e. $5 + (2^{w-2} + 1) \cdot 8 + (2^{w-2} - 1) \cdot 16 = 2^{w-2} \cdot 24 - 3$ field multiplications; and $\lfloor n/w \rfloor$ projective randomizations, $\lfloor n/w \rfloor \cdot w$ point doublings, and $\lfloor n/w \rfloor$ point additions for computing the result, i.e. $\lfloor n/w \rfloor \cdot 5 + \lfloor n/w \rfloor \cdot w \cdot 8 + \lfloor n/w \rfloor \cdot 16 = \lfloor n/w \rfloor \cdot (w \cdot 8 + 21)$ field multiplications. The total cost of this is

$$\left\lfloor \frac{n}{w} \right\rfloor \cdot (w \cdot 8 + 21) + 2^{w-2} \cdot 24 - 3$$

field multiplications.

For arbitrary window size $w \geq 2$ and scalar bit length n , the right-to-left method from section 8.4 with a scalar encoding using digits from the set $\{-2^{w-1}, \dots, 2^{w-1} - 1\}$ (with $\ell = \lfloor n/w \rfloor$, $\beta = 2^{w-1}$) performs $2^{w-1} + 1$ projective randomizations in the initialization stage ($2^{w-1} \cdot 5 + 5$ field multiplications); $\lfloor n/w \rfloor \cdot w$ point doublings and $\lfloor n/w \rfloor + 1$ point additions in the right-to-left stage ($\lfloor n/w \rfloor \cdot w \cdot 8 + (\lfloor n/w \rfloor + 1) \cdot 16 = \lfloor n/w \rfloor \cdot (w \cdot 8 + 16) + 16$ field

Table 8.1: Number of field multiplications for a 160-bit point multiplication

w	2	3	4	5	6
L-to-R method, Chud. Jac. coordinates	2430	2067	1939	1919	1987
L-to-R method, proj. rand. to avoid fixed table	2981	2430	2213	2141	2175
R-to-L method	2615	2241	2173	2309	2709
R-to-L method, two processors	1335	1393	1533	1797	2293

multiplications); and 2^{w-2} point doublings and $3 \cdot 2^{w-2} - 2$ point additions in the result stage ($2^{w-2} \cdot 8 + (3 \cdot 2^{w-2} - 2) \cdot 16 = 2^{w-2} \cdot 56 - 32$ field multiplications). The total cost is

$$\left\lfloor \frac{n}{w} \right\rfloor \cdot (w \cdot 8 + 16) + 2^{w-2} \cdot 66 - 11$$

field multiplications.

Note that for the right-to-left method in the case with parallelization, $w = 2$ provides better performance than larger values of w (the right-to-left stage provides essentially the same amount of work to both processors if $w = 2$). Compared with the one-processor variant, we always save $\lfloor n/w \rfloor \cdot 16$ field multiplications, and

$$\left\lfloor \frac{n}{w} \right\rfloor \cdot w \cdot 8 + 2^{w-2} \cdot 66 - 11$$

field multiplications remain.

Table 8.1 compares the efficiency of the methods for various window sizes in the case of 160-bit scalars. Table 8.2 provides a similar comparison for 256-bit scalars. (Note that these efficiency comparisons do not take into account the additional cost of generating random field elements for projective randomization; only field multiplications are counted.) Table entries are printed in bold if the respective window size w provides a lower field multiplication count than smaller values of w , i.e. if w is optimal for certain bounds on memory usage.

The right-to-left method needs read-write memory for the same number of points as the left-to-right method with the same window size. (The right-to-left method needs additional read-only memory for the precomputed points A_b^{fix} .) The tables show that for the example bit-lengths, when using a single processor, the left-to-right method with Chudnovsky Jacobian coordinates is always the fastest; but remember that the table contains more field elements because Chudnovsky Jacobian coordinates have the form (X, Y, Z, Z^2, Z^3) rather than just (X, Y, Z) ; so if memory is scarce, this method variant may not be applicable. The left-to-right method with additional projective randomizations to avoid a fixed table can be faster than the right-to-left method, but will need more read-write memory to achieve this: this variant of the left-to-right method needs $w = 5$ (17 table values) to outperform the right-to-left method with $w = 4$.

8.6 Scalar Randomization

Okeya and Sakurai [71] describe a second-order power analysis attack on the fixed-table point multiplication method of section 8.3 (as published in [59]). The attack requires power consumption traces from many point multiplications using the same scalar e (and thus the same addition chain). The basis of the attack is to detect table-value reuse by observing side-channel

Table 8.2: Number of field multiplications for a 256-bit point multiplication

w	2	3	4	5	6	7
L-to-R method, Chud. Jac. coordinates	3870	3283	3043	2945	2979	3263
L-to-R method, proj. rand. to avoid fixed table	4757	3870	3485	3300	3279	3537
R-to-L method	4151	3521	3325	3373	3733	4693
R-to-L method, two processors	2103	2161	2301	2557	3061	4117

data that leaks information on the Hamming weight of representations of points (cf. [54]): to find out whether the i -th and j -th point addition use the same table value, compute for each power consumption trace the difference between (normalized) power consumption measurements at the two points of time when the respective table entries are read from memory; over sufficiently many point multiplications, the sample variance of these power consumption differences should converge to one of two values depending on whether the i -th and j -th point addition use the same table value or different table values.

No experimental results are given in [71]. If this attack is practical, similar attacks may be possible against most point multiplication methods using a constant sequence of operations as it may be possible to trace values based on their Hamming weight (i.e. determine whether the output of the i -th operation is used as input to the j -th operation). A countermeasure is to randomize the addition chain. This can be done by randomizing the scalar e : compute eP with two point multiplications and one point subtraction as

$$(e + mN + \tilde{m})P - \tilde{m}P$$

where N is the order of the elliptic curve group and m, \tilde{m} are one-time random numbers (e.g. 32 bits long).

This can also be expressed as follows:

$$(e + mN + \tilde{m})P + \tilde{m}(-P)$$

When computing this sum of products with the right-to-left point multiplication method from section 8.4, the initialization stage and result stage need to be run just once; only the right-to-left stage needs to be run twice. The final result stage will automatically yield the combined result.

(Adding a multiple of the group order was originally proposed in [48], but it leaves some bias in the least significant digits [70]. Scalar splitting in the form $eP = (e + m)P - mP$ as proposed in [25] avoids this bias, but is only sufficient if m is of the same length as e , which would double the cost of a point multiplication. By combining these two ideas, we avoid the bias while keeping the overhead low.)

Bibliography

- [1] ABDALLA, M., BELLARE, M., AND ROGAWAY, P. DHAES: An encryption scheme based on the Diffie-Hellman problem. Submission to IEEE P1363a. <http://grouper.ieee.org/groups/1363/P1363a/Encryption.html>, 1998.
- [2] ABDALLA, M., BELLARE, M., AND ROGAWAY, P. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In *Progress in Cryptology – CT-RSA 2001* (2001), D. Naccache, Ed., vol. 2020 of *Lecture Notes in Computer Science*, pp. 143–158.
- [3] AKISHITA, T., AND TAKAGI, T. Zero-value point attacks on elliptic curve cryptosystem. In *Information Security – ISC 2003* (2003), Lecture Notes in Computer Science. To appear.
- [4] AMERICAN NATIONAL STANDARDS INSTITUTE (ANSI). Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA). ANSI X9.62, 1998.
- [5] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying hash functions for message authentication. In *Advances in Cryptology – CRYPTO '96* (1996), N. Koblitz, Ed., vol. 1109 of *Lecture Notes in Computer Science*, pp. 1–15.
- [6] BELLARE, M., DESAI, A., JOKIPII, E., AND ROGAWAY, P. A concrete security treatment of symmetric encryption. In *38th Annual Symposium on Foundations of Computer Science (FOCS '97)* (1997), IEEE Computer Society, pp. 394–403.
- [7] BELLARE, M., KILIAN, J., AND ROGAWAY, P. The security of cipher block chaining. In *Advances in Cryptology – CRYPTO '94* (1994), Y. G. Desmedt, Ed., vol. 839 of *Lecture Notes in Computer Science*, pp. 341–358.
- [8] BELLARE, M., KILIAN, J., AND ROGAWAY, P. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences* 61 (2000), 362–399.
- [9] BELLARE, M., AND ROGAWAY, P. Random oracles are practical: A paradigm for designing efficient protocols. In *First Annual Conference on Computer and Communications Security* (1993), ACM, pp. 62–73.
- [10] BIER, É., AND JOYE, M. Weierstraß elliptic curves and side-channel attacks. In *Public Key Cryptography – PKC 2002* (2002), D. Naccache and P. Paillier, Eds., vol. 2274 of *Lecture Notes in Computer Science*, pp. 335–345.

-
- [11] BLACK, J., HALEVI, S., KRAWCZYK, H., KROVETZ, T., AND ROGAWAY, P. UMAC: Fast and secure message authentication. In *Advances in Cryptology – CRYPTO ’99* (1999), M. Wiener, Ed., vol. 1666 of *Lecture Notes in Computer Science*, pp. 216–233.
- [12] BLAKE, I. F., SEROUSSI, G., AND SMART, N. P. *Elliptic Curves in Cryptography*, vol. 265 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1999.
- [13] BONEH, D., DEMILLO, R. A., AND LIPTON, R. J. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology* 14 (2001), 101–119.
- [14] BOS, J., AND COSTER, M. Addition chain heuristics. In *Advances in Cryptology – CRYPTO ’89* (1989), G. Brassard, Ed., vol. 435 of *Lecture Notes in Computer Science*, pp. 400–407.
- [15] BOSMA, W. Signed bits and fast exponentiation. Department of Mathematics, University of Nijmegen, Report No. 9935, 1999.
- [16] BRANDS, S. *Rethinking Public Key Infrastructures and Digital Certificates – Building in Privacy*. MIT Press, 2000.
- [17] BRICKELL, E. F., GORDON, D. M., MCCURLEY, K. S., AND WILSON, D. B. Fast exponentiation with precomputation. In *Advances in Cryptology – EUROCRYPT ’92* (1993), R. A. Rueppel, Ed., vol. 658 of *Lecture Notes in Computer Science*, pp. 200–207.
- [18] BROWN, M., HANKERSON, D., LÓPEZ, J., AND MENEZES, A. Software implementation of the NIST elliptic curves over prime fields. In *Progress in Cryptology – CT-RSA 2001* (2001), D. Naccache, Ed., vol. 2020 of *Lecture Notes in Computer Science*, pp. 250–265.
- [19] CANETTI, R. Personal communication, 2003.
- [20] CANETTI, R., GOLDREICH, O., AND HALEVI, S. The random oracle methodology, revisited. In *30th ACM Symposium on Theory of Computing (STOC)* (1998), pp. 209–218.
- [21] CANETTI, R., GOLDREICH, O., AND HALEVI, S. The random oracle methodology, revisited. E-print cs.CR/0010019, 2000. Available from <http://arXiv.org/abs/cs/0010019>.
- [22] CERTICOM RESEARCH. Standards for efficient cryptography – SEC 1: Elliptic curve cryptography. Version 1.0, 2000. Available from <http://www.secg.org/>.
- [23] CERTICOM RESEARCH. Standards for efficient cryptography – SEC 2: Recommended elliptic curve cryptography domain parameters. Version 1.0, 2000. Available from <http://www.secg.org/>.
- [24] CHAUM, D. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM* 24 (1981), 84–88.
- [25] CLAVIER, C., AND JOYE, M. Universal exponentiation algorithm – a first step towards provable SPA-resistance. In *Cryptographic Hardware and Embedded Systems – CHES 2001* (2001), Ç. K. Koç, D. Naccache, and C. Paar, Eds., vol. 2162 of *Lecture Notes in Computer Science*, pp. 300–308.

-
- [26] COHEN, H. Analysis of the flexible window powering algorithm. <http://www.math.u-bordeaux.fr/%7Ecohen/window.dvi>, 1999.
- [27] COHEN, H., ONO, T., AND MIYAJI, A. Efficient elliptic curve exponentiation using mixed coordinates. In *Advances in Cryptology – ASIACRYPT '98* (1998), K. Ohta and D. Pei, Eds., vol. 1514 of *Lecture Notes in Computer Science*, pp. 51–65.
- [28] CORON, J.-S. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems – CHES '99* (1999), Ç. K. Koç and C. Paar, Eds., vol. 1717 of *Lecture Notes in Computer Science*, pp. 292–302.
- [29] COTTRELL, L. Mixmaster & remailer attacks. <http://web.archive.org/web/19961112194057/http://www.obscura.com/%7Elok%i/remailer/remailer-essay.html>, 1996.
- [30] CRAMER, R., AND SHOUP, V. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. Manuscript, <http://shoup.net/papers/>, 2001.
- [31] DE ROOIJ, P. Efficient exponentiation using precomputation and vector addition chains. In *Advances in Cryptology – EUROCRYPT '94* (1995), T. Helleseth, Ed., vol. 950 of *Lecture Notes in Computer Science*, pp. 389–399.
- [32] DIFFIE, W., AND HELLMAN, M. E. Multiuser cryptographic techniques. In *Proceedings of AFIPS National Computer Conference* (1976), pp. 109–112.
- [33] DIFFIE, W., AND HELLMAN, M. E. New directions in cryptography. *IEEE Transactions on Information Theory* 22, 6 (1976), 644–654.
- [34] DIMITROV, V. S., JULLIEN, G. A., AND MILLER, W. C. Complexity and fast algorithms for multiexponentiation. *IEEE Transactions on Computers* 49 (2000), 141–147.
- [35] ELGAMAL, T. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory* 31 (1985), 469–472.
- [36] FISCHER, W., GIRAUD, C., KNUDSEN, E. W., AND SEIFERT, J.-P. Parallel scalar multiplication on general elliptic curves over \mathbb{F}_p hedged against non-differential side-channel attacks. Cryptology ePrint Archive Report 2002/007, 2002. Available from <http://eprint.iacr.org/>.
- [37] GOLDWASSER, S., AND MICALI, S. Probabilistic encryption. *Journal of Computer and System Sciences* 28 (1984), 270–299.
- [38] GORDON, D. M. A survey of fast exponentiation methods. *Journal of Algorithms* 27 (1998), 129–146.
- [39] GOUBIN, L. A refined power-analysis attack on elliptic curve cryptosystems. In *Public Key Cryptography – PKC 2003* (2003), Y. G. Desmedt, Ed., vol. 2567 of *Lecture Notes in Computer Science*, pp. 199–211.
- [40] HAMDY, S., AND MÖLLER, B. Security of cryptosystems based on class groups of imaginary quadratic orders. In *Advances in Cryptology – ASIACRYPT 2000* (2000), T. Okamoto, Ed., vol. 1976 of *Lecture Notes in Computer Science*, pp. 234–247.

- [41] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS (IEEE). IEEE standard specifications for public-key cryptography. IEEE Std 1363-2000, 2000.
- [42] ITOH, K., TAKENAKA, M., TORII, N., TEMMA, S., AND KURIHARA, Y. Fast implementation of public-key cryptography on a DSP TMS320C6201. In *Cryptographic Hardware and Embedded Systems – CHES '99* (1999), Ç. K. Koç and C. Paar, Eds., vol. 1717 of *Lecture Notes in Computer Science*, pp. 61–72.
- [43] IZU, T., AND TAKAGI, T. A fast parallel elliptic curve multiplication resistant against side channel attacks. In *Public Key Cryptography – PKC 2002* (2002), D. Naccache and P. Paillier, Eds., vol. 2274 of *Lecture Notes in Computer Science*, pp. 280–296.
- [44] JAKOBSSON, M., AND JUELS, A. An optimally robust hybrid mix network. In *20th Annual ACM Symposium on Principles of Distributed Computing (PODC 2001)* (2001), ACM Press, pp. 284–292.
- [45] JOYE, M., AND TYMEN, C. Compact encoding of non-adjacent forms with applications to elliptic curve cryptography. In *Public Key Cryptography – PKC 2001* (2001), K. Kim, Ed., vol. 1992 of *Lecture Notes in Computer Science*, pp. 353–364.
- [46] KNUTH, D. E. *The Art of Computer Programming – Vol. 2: Seminumerical Algorithms (2nd ed.)*. Addison-Wesley, 1981.
- [47] KNUTH, D. E. *The Art of Computer Programming – Vol. 2: Seminumerical Algorithms (3rd ed.)*. Addison-Wesley, 1998.
- [48] KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology – CRYPTO '96* (1996), N. Koblitz, Ed., vol. 1109 of *Lecture Notes in Computer Science*, pp. 104–113.
- [49] KOCHER, P. C., JAFFE, J., AND JUN, B. Differential power analysis. In *Advances in Cryptology – CRYPTO '99* (1999), M. Wiener, Ed., vol. 1666 of *Lecture Notes in Computer Science*, pp. 388–397.
- [50] KROVETZ, T., BLACK, J., HALEVI, S., HEVIA, A., KRAWCZYK, H., AND ROGAWAY, P. UMAC: Message authentication code using universal hashing. Internet-Draft `draft-krovetz-umac-01.txt`, <http://www.cs.ucdavis.edu/~rogaway/umac/>, 2000.
- [51] LIM, C. H., AND LEE, P. J. More flexible exponentiation with precomputation. In *Advances in Cryptology – CRYPTO '94* (1994), Y. G. Desmedt, Ed., vol. 839 of *Lecture Notes in Computer Science*, pp. 95–107.
- [52] LIPMAA, H., ROGAWAY, P., AND WAGNER, D. Comments to NIST concerning AES modes of operation: CTR-mode encryption. <http://csrc.nist.gov/encryption/modes/workshop1/papers/lipmaa-ctr.pdf>, 2000.
- [53] MENEZES, A. J., VAN OORSCHOT, P. C., AND VANSTONE, S. A. *Handbook of Applied Cryptography*. CRC Press, 1997.
- [54] MESSERGES, T. S. Using second-order power analysis to attack DPA resistant software. In *Cryptographic Hardware and Embedded Systems – CHES 2000* (2000), Ç. K. Koç and C. Paar, Eds., vol. 1965 of *Lecture Notes in Computer Science*, pp. 238–251.

- [55] MILLER, V. S. Use of elliptic curves in cryptography. In *Advances in Cryptology – CRYPTO ’85* (1986), H. C. Williams, Ed., vol. 218 of *Lecture Notes in Computer Science*, pp. 417–428.
- [56] Mixmaster anonymous remailer software. <http://sourceforge.net/projects/mixmaster/>.
- [57] MIYAJI, A., ONO, T., AND COHEN, H. Efficient elliptic curve exponentiation. In *International Conference on Information and Communications Security – ICICS ’97* (1997), Y. Han, T. Okamoto, and S. Qing, Eds., vol. 1334 of *Lecture Notes in Computer Science*, pp. 282–290.
- [58] MÖLLER, B. Algorithms for multi-exponentiation. In *Selected Areas in Cryptography – SAC 2001* (2001), S. Vaudenay and A. M. Youssef, Eds., vol. 2259 of *Lecture Notes in Computer Science*, pp. 165–180.
- [59] MÖLLER, B. Securing elliptic curve point multiplication against side-channel attacks. In *Information Security – ISC 2001* (2001), G. I. Davida and Y. Frankel, Eds., vol. 2200 of *Lecture Notes in Computer Science*, pp. 324–334.
- [60] MÖLLER, B. Securing elliptic curve point multiplication against side-channel attacks, addendum: Efficiency improvement. <http://www.informatik.tu-darmstadt.de/TI/Mitarbeiter/moeller/ecc-sca-isc01.pdf>, 2001.
- [61] MÖLLER, B. Parallelizable elliptic curve point multiplication method with resistance against side-channel attacks. In *Information Security – ISC 2002* (2002), A. H. Chan and V. Gligor, Eds., vol. 2433 of *Lecture Notes in Computer Science*, pp. 402–413.
- [62] MÖLLER, B. Improved techniques for fast exponentiation. In *Information Security and Cryptology – ICISC 2002* (2003), P. J. Lee and C. H. Lim, Eds., vol. 2587 of *Lecture Notes in Computer Science*, pp. 298–312.
- [63] MÖLLER, B. Provably secure public-key encryption for length-preserving chaumian mixes. In *Topics in Cryptology – CT-RSA 2003* (2003), M. Joye, Ed., vol. 2612 of *Lecture Notes in Computer Science*, pp. 244–262.
- [64] MÖLLER, U., AND COTTRELL, L. Mixmaster protocol version 2. <http://www.eskimo.com/~rowdenw/crypt/Mix/draft-moeller-v2-01.txt>, 2000.
- [65] MONTGOMERY, P. L. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation* 48 (1987), 243–264.
- [66] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). Digital Signature Standard (DSS). FIPS PUB 186-2, 2000.
- [67] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). Digital Signature Standard (DSS). FIPS PUB 186-2, 2000.
- [68] OHKUBO, M., AND ABE, M. A length-invariant hybrid mix. In *Advances in Cryptology – ASIACRYPT 2000* (2000), T. Okamoto, Ed., vol. 1976 of *Lecture Notes in Computer Science*, pp. 178–191.

-
- [69] OKEYA, K. Method of calculating multiplication by scalars on an elliptic curve and apparatus using same. European Patent EP1160661, 2001.
- [70] OKEYA, K., AND SAKURAI, K. Power analysis breaks elliptic curve cryptosystems even secure against the timing attack. In *Progress in Cryptology – INDOCRYPT 2000* (2000), B. K. Roy and E. Okamoto, Eds., vol. 1977 of *Lecture Notes in Computer Science*, pp. 178–190.
- [71] OKEYA, K., AND SAKURAI, K. A second-order DPA attack breaks a window-method based countermeasure against side channel attacks. In *Information Security – ISC 2002* (2002), A. H. Chan and V. Gligor, Eds., vol. 2433 of *Lecture Notes in Computer Science*, pp. 389–401.
- [72] PEDERSEN, T., AND PFITZMANN, B. Fail-stop signatures. *SIAM Journal on Computing* 26 (1997), 291–330.
- [73] PIPPENGER, N. On the evaluation of powers and related problems (preliminary version). In *17th Annual Symposium on Foundations of Computer Science* (1976), IEEE Computer Society, pp. 258–263.
- [74] RACKOFF, C. W., AND SIMON, D. R. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology – CRYPTO '91* (1992), J. Feigenbaum, Ed., vol. 576 of *Lecture Notes in Computer Science*, pp. 433–444.
- [75] REITWIESNER, G. W. Binary arithmetic. *Advances in Computers* 1 (1960), 231–308.
- [76] RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21, 2 (1978), 120–126.
- [77] SAKAI, Y., AND SAKURAI, K. Algorithms for efficient simultaneous elliptic scalar multiplication with reduced joint Hamming weight representation of scalars. In *Information Security – ISC 2002* (2002), A. H. Chan and V. Gligor, Eds., vol. 2433 of *Lecture Notes in Computer Science*, pp. 484–499.
- [78] SCHROEPEL, R., ORMAN, H., O'MALLEY, S., AND SPATSCHECK, O. Fast key exchange with elliptic curve systems. In *Advances in Cryptology – CRYPTO '95* (1995), D. Coppersmith, Ed., vol. 963 of *Lecture Notes in Computer Science*, pp. 43–56.
- [79] SHANNON, C. E. Communication theory of secrecy systems. *Bell System Technical Journal* 28 (1949), 656–715.
- [80] SHOUP, V. A proposal for an ISO standard for public key encryption. Version 2.1, December 20, 2001. <http://shoup.net/papers/>.
- [81] SOLINAS, J. A. An improved algorithm for arithmetic on a family of elliptic curves. In *Advances in Cryptology – CRYPTO '97* (1997), B. S. Kaliski, Jr., Ed., vol. 1294 of *Lecture Notes in Computer Science*, pp. 357–371.
- [82] SOLINAS, J. A. Efficient arithmetic on Koblitz curves. *Designs, Codes and Cryptography* 19 (2000), 195–249.

-
- [83] STRAUS, E. G. Problems and solutions: Addition chains of vectors. *American Mathematical Monthly* 71 (1964), 806–808.
- [84] THURBER, E. G. On addition chains $l(mn) \leq l(n) + b$ and lower bounds for $c(r)$. *Duke Mathematical Journal* 40 (1973), 907–913.
- [85] TSIOUNIS, Y., AND YUNG, M. On the security of ElGamal-based encryption. In *Public Key Cryptography – PKC '98* (1998), H. Imai and Y. Zheng, Eds., vol. 1431 of *Lecture Notes in Computer Science*, pp. 117–134.
- [86] VADEKAR, A., AND LAMBERT, R. J. Timing attack resistant cryptographic system. Patent Cooperation Treaty (PCT) Publication WO 00/05837, 2000.
- [87] VANSTONE, S. A., AND GALLANT, R. P. Power signature attack resistant cryptography. Patent Cooperation Treaty (PCT) Publication WO 00/25204, 2000.
- [88] VERNAM, G. S. Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Journal of the American Institute of Electrical Engineers XLV* (1926), 109–115.
- [89] WEGMAN, M. N., AND CARTER, J. L. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences* 22 (1981), 265–279.
- [90] YAO, A. C.-C. On the evaluation of powers. *SIAM Journal on Computing* 5 (1976), 100–103.
- [91] YEN, S.-M., LAIH, C.-S., AND LENSTRA, A. K. Multi-exponentiation. *IEE Proceedings – Computers and Digital Techniques* 141 (1994), 325–326.

Index

- adaptive chosen-ciphertext attack, 28, 33, 47
- Adv, 21, 35
- advantage, 21
 - CCA, 29, 34, 49
 - CPA, 21
 - pseudo-random bit string generator, 35
 - unlinkability, 47
- AdvCCA, 29, 34, 49
- AdvCPA, 21
- AdvForge, 34
- affine representation, 87
- almost strongly universal hash function, 31
- asymmetric cryptography, 18
- asymptotic security, 23

- basic interleaved exponentiation method, 76
- bit string oracle, 35

- CCA advantage, 29, 34, 49
- challenge, 34, 35
 - ciphertext, 21, 29, 48
- chosen-ciphertext attack, 28
 - adaptive, 28, 33, 47
- chosen-plaintext attack, 21
- Chudnovsky Jacobian projective representation, 92
- CipherLen, 30
- ciphertext, 17, 18, 29
 - challenge, 21, 29, 48
- cofactor, 88
- collision, 24
- computational security, 20
- concrete security, 21
- counter mode, 32
- CPA advantage, 21
- cryptography
 - asymmetric, 18
 - public-key, 18
 - symmetric, 17

- decryption oracle, 28, 29, 33, 34, 48, 49
- DH, 19, 55
- DHAES, 27, 32
- DHIES, 27
- Diffie-Hellman, 19, 55
- digital signature, 18
- domain parameters, 20, 55
- DSA, 55, 82

- ECDSA, 55, 82
- ElGamal
 - encryption, 20, 29
 - signatures, 55
- elliptic curve cryptography, 56, 87
- encryption oracle, 21, 29, 46
- evaluation stage, 60, 71, 72, 95, 96
- exponentiation, 55
 - fractional windows, 64
 - interleaved, 71, 75
 - basic, 76
 - window-NAF based, 77
 - left-to-right, 60
 - Lim-Lee, 84
 - multi-exponentiation, 71
 - right-to-left, 61
 - Shamir's trick, 71
 - signed fractional windows, 64
 - signed windows, 62
 - simultaneous, 71
 - sliding windows, 62
 - unsigned fractional windows, 67
 - with precomputation, 55

- fail-stop signature schemes, 20
- find stage, 28, 29, 48
- flexible window size, 86
- fractional windows, 64

- generic attack, 24
- guess stage, 28, 29, 49
- hash function, 19, 24
 - almost strongly universal, 31
 - collision, 24
- HMAC, 31
- hybrid encryption, 27
- IND-CCA, 29
- IND-CPA, 21
- indistinguishability
 - under chosen-ciphertext attack, 29
 - under chosen-plaintext attack, 21
- information-theoretic security, 18
- initialization stage, 97
- interactive encryption oracle, 48
- interleaved exponentiation, 71, 75
- Jacobian projective representation, 88
- KEM, 18, 30
- key, 17
 - public, 18, 28, 30
 - secret, 18, 28, 30
- key derivation function, 31
- key encapsulation mechanism, 18, 30
- key encapsulation oracle, 34
- key generation algorithm, 28, 30
- key generation oracle, 21, 29, 33, 46, 47
- left-to-right exponentiation, 60
- left-to-right stage, 60, 71, 72
- length-expanding decryption, 40
- length-preserving mix, 39
- Lim-Lee exponentiation, 84
- LSB, 57
- LSB_m , 57
- MAC, 31
- MAC oracle, 34
- message authentication code, 31
- mix, 39
- modified signed fractional window representation, 65
- modified width- $(w + 1)$ NAF, 63
- modified window NAF, 63
- multi-exponentiation, 55, 71
- NAF, 62
- negligible, 23
- non-adjacent form, 62
- one-time message authentication code, 31
- one-time pad, 17
- oracle
 - bit string, 35
 - decryption, 28, 29, 33, 34, 48, 49
 - encryption, 21, 29, 46
 - interactive, 48
 - key encapsulation, 34
 - key generation, 21, 29, 33, 46, 47
 - MAC, 34
 - random, 24
- OTP, 17
- plaintext, 17, 18, 29
- point
 - addition, 87
 - at infinity, 87
 - doubling, 87
 - inversion, 87
 - multiplication, 87
- precomputation, 55
- precomputation stage, 60, 71, 72, 95, 96
- precomputed table, 60
- primitives, 22, 30
- projective randomization, 88
- projective representation, 87
 - Chudnovsky Jacobian, 92
 - Jacobian, 88
- pseudo-random bit string generator, 31
- public key, 18, 28, 30
- public-key cryptography, 18
- public-key encryption, 18
- public-key encryption scheme, 28
- random oracle model, 24
- reduction-based security proof, 22
- representation
 - affine, 87
 - projective, 87
 - Chudnovsky Jacobian, 92
 - Jacobian, 88
- result stage, 61, 98
- right-to-left exponentiation, 61
- right-to-left stage, 61, 98

RSA, 55

secret key, 18, 28, 30

security parameter, 23

Shamir's trick, 71

side-channel attacks, 88

signed fractional windows, 64

signed windows, 62

simultaneous exponentiation, 71

sliding windows, 62

source routing,, 39

STREAM, 31

symmetric cryptography, 17

UHASH, 31

unlinkability, 45

unsigned fractional windows, 67

Vernam encryption, 17

width- $(w + 1)$ NAF, 62
 modified, 63

window NAF, 62
 modified, 63

window NAF splitting, 85

window size, 62, 72
 flexible, 86

window-NAF based interleaved exponenti-
 ation method, 77

wNAF, 62