BATCH VERIFICATION OF ELLIPTIC CURVE AND EDWARDS CURVE DIGITAL SIGNATURES

Sabyasachi Karati

BATCH VERIFICATION OF ELLIPTIC CURVE AND EDWARDS CURVE DIGITAL SIGNATURES

Thesis submitted in partial fulfillment
of the requirements for the award of the degree

of

Doctor of Philosophy

by

Sabyasachi Karati

Under the supervision of

Dr. Abhijit Das

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

2014

APPROVAL OF THE VIVA-VOCE BOARD

Certified that the thesis entitled **"Batch Verification of Elliptic Curve and Edwards Curve Digital Signatures"** submitted by **Sabyasachi Karati** to the Indian Institute of Technology, Kharagpur, for the award of the degree of Doctor of Philosophy has been accepted by the external examiners and that the student has successfully defended the thesis in the viva-voce examination held today.

Dr. Arobinda Gupta
Professor
Computer Science and Engineering
IIT Kharagpur

Dr. Debdeep Mukhopadhyay
Associate Professor
Computer Science and Engineering
IIT Kharagpur

Dr. Shamik Sural
Professor
School of Information Technology
IIT Kharagpur

Dr. Abhijit Das
Associate Professor
Computer Science and Engineering
IIT Kharagpur
PhD Supervisor

Dr. C. Pandu Rangan
Professor
Computer Science and Engineering
IIT Madras
External Examiner

Dr. Dipanwita Roychaudhury
Professor
Computer Science and Engineering
IIT Kharagpur
DSC Chairman

Date:

CERTIFICATE

This is to certify that the thesis entitled **"Batch Verification of Elliptic Curve and Edwards Curve Digital Signatures"**, submitted by **Sabyasachi Karati** to the Indian Institute of Technology, Kharagpur, for the partial fulfillment of the award of the degree of Doctor of Philosophy, is a record of bona fide research work carried out by him under our supervision and guidance.

The thesis in our opinion, is worthy of consideration for the award of the degree of Doctor of Philosophy in accordance with the regulations of the Institute. To the best of our knowledge, the results embodied in this thesis have not been submitted to any other University or Institute for the award of any other Degree or Diploma.

Dr. Abhijit Das
Associate Professor
CSE, IIT Kharagpur

Date:

DECLARATION

I certify that

(a) The work contained in the thesis is original and has been done by myself under the general supervision of my supervisors.

(b) The work has not been submitted to any other Institute for any degree or diploma.

(c) I have followed the guidelines provided by the Institute in writing the thesis.

(d) I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

(e) Whenever I have used materials (data, theoretical analysis, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

(f) Whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Sabyasachi Karati

ACKNOWLEDGEMENTS

In the beginning, post joining IIT Kharagpur I was not sure of what my future would be like. I really enjoyed my time at IIT Kharagpur. I had made many friends in the process. The best thing that happened in those years was that I got introduced to elliptic-curve cryptography through my M.Tech. project. I found this topic so interesting and I felt so passionate about it, that I enrolled myself into the PhD program to continue my research on elliptic-curve cryptography.

I would first like to thank all my family members for their endless support and understanding throughout this journey. My mother, Sandhya Karati, has been my pillar of support and instrumental in shaping me for which I am ever grateful.

The person whom I would like to thank next is my supervisor Dr. Abhijit Das, for whom I have the deepest regards. He has been the most significant person in this journey of mine. His influence has shaped my perceptive towards life. I have enjoyed all the time that I have spent with him and I am very much grateful to him for that. He is my friend, philosopher and guide for life. My love for public-key cryptography has been his biggest contribution. When I leave IIT Kharagpur, I will miss his teachings, his care and so many other things for which I am forever indebted.

I would also like to thank Prof. Dipanwita Roychaudhury and Dr. Debdeep Mukhopadhyay for inspiring me and critically evaluating me, which helped me in my research a lot. I also would like to thank Bhargav Bellur, Debojyoti Bhattacharya and Aravind Iyer for helping me during the initial years of my PhD. I thank all the faculty members and staff members of this department for their support and assistance.

Last but not the least I would like to thank all my friends. I would not be able mention all their names as it would not fit. I will mention a few without whom my life at IIT Kharagpur would be void. I would like to thank Anup Kumar Bhattacharya, Binanda Sengupta, Satrajit Ghosh, Sudakshina Datta, Shamit Ghosh, Abhrajit Sengupta, Joy Chandra Mukherjee, Shuvo Bardhan, Dhiman Saha, Anirban Bhattacharya and many others for their beautiful friendship. I will treasure all the time that I have spent with all of you.

I express my sincere thanks to all of you.

Sabyasachi Karati

ABSTRACT

In this thesis, several algorithms for the batch verification of standard ECDSA signatures are introduced. The first of these algorithms is based upon the naive idea of taking square roots in the underlying field. In order to improve the efficiency beyond what can be achieved by the naive algorithm, two new algorithms are initially proposed, which replace square-root computations by symbolic manipulations. We then use elliptic-curve summation polynomials to design ECDSA batch-verification algorithms which are significantly faster than our batch-verification algorithms based on symbolic manipulations. Experiments carried out on NIST prime curves demonstrate a maximum speedup of above six over individual verification if all the signatures in the batch belong to the same signer, and a maximum speedup of about two if the signatures in the batch belong to different signers, both achieved by a fast variant of our batch-verification algorithm based on elliptic-curve summation polynomials. We also establish a theoretical connection between our symbolic-computation and summation-polynomial algorithms. To the best of our knowledge, these are the first algorithms to address the problem of batch verification of standard ECDSA signatures.

We propose three randomization methods for our batch-verification algorithms in order to prevent several types of attacks. The first method is based on Montgomery ladders, and the second on computing square roots in the underlying field. Both these methods use numeric arithmetic only. Our third proposal exploits symbolic computations leading to a seminumeric algorithm. We theoretically and experimentally prove that for standard ECDSA signatures, our seminumeric randomization algorithm in tandem with the summation-polynomial-based batch-verification algorithm gives the best speedup over individual verification. If each ECDSA signature contains an extra bit to uniquely identify the correct y-coordinate of the elliptic-curve point appearing in the signature, then the second (numeric) randomization method followed by the naive batch-verification algorithm yields the best performance gains. Randomization significantly brings down the performance gains achieved by batch verification. For standard ECDSA signatures, our experiments reveal a maximum reduced speedup close to two for half-length

randomizers and for all signatures in a batch coming from the same signer.

We theoretically prove that all the proposed algorithms offer the same security as the straightforward batch verification of ECDSA* signatures in which the x-coordinates of the elliptic-curve points are replaced by the entire points.

Our batch-verification algorithms are also ported to NIST Koblitz curves defined over finite fields of characteristic two. We also make a comparative study of our algorithms for the Edwards curve digital signature algorithm (EdDSA) over a medium-sized prime field. We report our experimental results both with and without randomization.

All our algorithms are practical only for small ($\leqslant 10$) batch sizes, because their running times and space requirements are exponential in the batch size. It remains an open problem whether the batch-verification problem for standard ECDSA signatures can be solved in less than exponential (possibly even in polynomial) time.

Keywords: Digital Signatures, Elliptic Curves, Koblitz Curve, Edwards Curve, ECDSA, EdDSA, Batch Verification, Scalar Multiplication, Symbolic Computation, Linearization, Multivariate Polynomial, Summation Polynomial, Randomization, Montgomery Ladder

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Cryptography is the study of algorithms for achieving several security goals including privacy and authentication [19]. By privacy, we mean secure communication over a public and insecure communication channel. Cryptographic encryption algorithms ensure that unauthorized third parties cannot extract any information from encrypted messages. However, the intended recipients of the messages possess some secret key materials which enable them to decrypt and recover the original messages. On the other hand, authentication protects messages from unauthorized tampering during transmission. Authentication protocols also ensure that the messages originate from genuine users.

Digital signatures are widely used to implement authentication protocols. In 1976, Diffie and Hellman [19] first propose the basic concept of digital signatures while addressing the authentication problem. A *digital signature* is the digital equivalent of hand-written signatures. The designated signer can create a digital signature on a message using some secret key. Other entities having no knowledge of this key cannot forge a signature on behalf of the designated signer. Anybody can, however, verify the signature because signature verification uses only public information.

Massive deployment of the Internet in several critical applications like online banking, marketing, payment and other electronic transactions calls for frequent use of digital signatures. Digital-signature schemes typically use exponentiation-based modular arithmetic algorithms which are somewhat demanding in terms of timing and space resources. In modern desktop machines and laptop computers, these timing

and space overheads do not impose serious practical restrictions. However, the use of digital signatures is not limited to such platforms alone. We need to address the applicability of cryptographic primitives in several types of resource-constrained devices like in the following applications.

- *Sensor networks*: A sensor network consists of a large number of computing nodes with restricted computation and communication capabilities and limited memory and battery life. When used in military and medical applications, public-key algorithms like digital signatures pose a serious problem.
- *Vehicular ad hoc networks*: In a VANET, vehicles broadcast periodic beacons for assistive driving. The authenticity of the broadcaster is ensured by digital signatures. Processors in a vehicle are not as primitive as in a sensor node. Nevertheless, in order to ensure cost effectiveness, we do not expect very high-end computing nodes in the vehicles. The main issue here is response to the beacons in real time. When many signatures need to be verified in a small amount of time as in a congested road, public-key algorithms may be a serious overhead, particularly in terms of computing time.
- *Hand-held devices*: Many security-critical applications need to run on handheld devices like mobile phones and tablets which vary widely in terms of system configuration. Low-end devices may lack enough computing power and/or memory to run expensive public-key algorithms.
- *Internet of things*: Embedded devices may also need to authenticate. Signature verification may incur practical computing overheads in low-end devices.

Applications like those just listed require three desirable properties of digital-signature algorithms:

- The computing overhead for signature generation and verification should be as low as possible.
- The key size should be small in order to reduce storage overhead.
- Signature sizes should be small in order to reduce communication bandwidth.

In view of these requirements, the elliptic curve digital signature algorithm (ECDSA) turns out to be the preferred choice among widely used signature schemes [3, 18, 73].

When multiple signatures need to be verified, the idea of *batch verification* turns out to be a useful one. This involves verifying multiple signatures together in a time less than the total time for individual verification of all these signatures. ECDSA signatures, although a favorite in many security-critical applications as mentioned above, offers significant resistance to batch verification.

This thesis primarily deals with the issues associated with the batch verification of ECDSA signatures. To the best extent of our knowledge, we propose the first algorithms to work for standardized ECDSA signatures. The novelty in our proposal is the use of symbolic computation and summation polynomials in the context of batch verification. We hydrothermally and experimentally establish the practicality of our proposed algorithms.

1.1 Overview and Motivation

The concept of digital signatures is first proposed in [19] by Diffie and Hellman. The first practically applicable signature scheme RSA is proposed by Rivest, Shamir and Adleman in 1978 [60]. The security of the RSA algorithm is based on the hardness of the factorization of a composite number which is a product of two large primes. In 1979, Rabin proposes another signature scheme based on the modular square-root problem [57] which is computationally equivalent to the factorization problem. Goldwasser and Micali first introduce the notion of probabilistic trapdoor functions [23]. In 1985, ElGamal proposes a new type of digital signature scheme based on the discrete logarithm problem in prime fields [20]. The ElGamal signature scheme is probabilistic in nature. The Digital Signature Algorithm (DSA) [46] is proposed by National Institute of Standards and Technology (NIST) in 1991. DSA is a variant of the ElGamal digital signature scheme. Later in 2001, the Elliptic Curve Digital Signature Algorithm (ECDSA) is proposed by Johnson, Menezes and Vanstone [33]. Many other digital signature schemes are also proposed. Most of these digital signatures can be divided into two groups based on the underlying intractable problems:

- RSA-type digital-signature schemes, the security of which is based on factoring large composite integers.
- ElGamal-type digital-signature schemes, the security of which is based on the discrete logarithm problem in a finite field or in an elliptic-curve group.

The verification primitive of an ElGamal-like signature requires at least two finite-field exponentiations (for DSA) or two elliptic-curve scalar multiplications (for ECDSA). Each such modular exponentiation or scalar multiplication is the most time-consuming operation compared to the other field operations.

All these signature schemes suffer from one common problem. Usually, signature verification is much slower than the signing procedure. Many applications have to verify multiple signatures in real time. Naccache et al. introduce a solution to this problem [45] in 1994. They propose the concept of *batch verification*, that is the verifier simultaneously verifies signatures on a batch of electronic documents. An interactive batch verification procedure is proposed for DSA signatures in [45]. In 1997, the concept of batch RSA is introduced by Fiat [21]. Harn, in 1998, proposes an efficient scheme for the batch verification of RSA signatures [27]. In this scheme (also see [29]), multiple signatures signed by the same private key can be verified simultaneously. Harn's scheme uses only one exponentiation for batches of any size. Harn's scheme does not adapt to the case of signatures from multiple signers.

These proposed batch-verification procedures are not directly applicable to ECDSA signatures. But the key sizes of ECDSA signatures are much smaller than the key sizes of RSA and DSA signatures. In Table 1.1 [51], the (L,N) pair presents the bit lengths of public and private keys of the DSA signer, k is the bit length of the modulus in RSA, and f is the group order of the base point of the elliptic-curve group. From Table 1.1, we can see that in order to achieve 256 bits of security, ECDSA needs only 512 bit keys. On the other hand, corresponding DSA and RSA key sizes are $(15360,512)$ bits and 15360 bits. So there has been a growing interest in ECDSA. Also smaller key sizes make ECDSA signature verification much faster then RSA and DSA verification.

ECDSA*, a modification of ECDSA introduced by Antipa et al. [2], permits an easy adaptation of Naccache et al's batch-verification protocol for DSA. Cheon and Yi [14] study batch verification of ECDSA* signatures, and report speedup factors

Table 1.1: Comparable strengths of common signature algorithms

Bits of Security	DSA minimum (L,N)	RSA minimum k	ECDSA minimum f
80	(1024,160)	1024	160
112	(2048,224)	2048	224
128	(3072,256)	3072	256
192	(7680,384)	7680	384
256	(15360,512)	15360	512

of up to 7 for same signer and 4 for different signers. However ECDSA* is not accepted as a standard signature scheme like DSA, RSA or ECDSA [46]. Thus, the use of ECDSA* is unacceptable, particularly in applications where interoperability is of important concern. Moreover, ECDSA* approximately doubles the size of the signature compared to ECDSA, without any increase in security. Consequently, batch verification of original ECDSA signatures turns out to be a practically important open research problem. Bellare et al. show in [4] that it is necessary to randomize the batch-verification procedure in order to prevent forgery attacks. To the best of our knowledge, no significant results in the batch verification of ECDSA signatures and associated randomization issues have been reported in the literature.

The performance gains achieved by batch verification come at a cost. We forfeit the ability to identify individual faulty signatures. This means that if batch verification fails, we need to resort to individual verification. Indeed, any batch-verification scheme turns out to be useful only when most signatures are authentic. Another disadvantage of batch-verification schemes is that they verify an aggregate of the signatures. A batch may be verified as a whole even when it contains faulty signatures. The use of randomization makes such incidents practically improbable.

1.2 Contributions

In this thesis, we propose several algorithms to verify batches of *standard* ECDSA signatures on NIST prime and Koblitz curves. The first algorithm we introduce

(henceforth denoted as Algorithm N) is based upon a naive approach of taking square roots in the underlying field. As the field size increases, square-root computations become quite costly. We modify Algorithm N by replacing square-root calculations by symbolic manipulations. This leads to two novel ECDSA batch-verification algorithms, called S1 and S2. Algorithm S1 is not as practical as Algorithm S2, but it provides the theoretical and practical foundations for arriving at Algorithm S2. For a wide range of field and batch sizes, Algorithm S2 convincingly outperforms the naive Algorithm N. Both S1 and S2 are probabilistic algorithms in the Monte Carlo sense, that is, they may occasionally fail to verify correct signatures. We analytically establish that for randomly generated signatures, the failure probability is extremely low. Using a one-level divide-and-conquer strategy, we arrive at faster variants S1′ and S2′ of Algorithms S1 and S2.

We then use elliptic-curve summation polynomials to design a theoretically and practically faster ECDSA batch-verification algorithm denoted as Algorithm SP. A modification of Algorithm S2′ leads to another Algorithm S3 based on symbolic manipulations. We establish that Algorithms SP and S3 essentially perform the same work, and turn out to be the fastest of all the batch-verification algorithms we study. Replacing the resultant computation in the last steps in Algorithms SP and S3 by a gcd calculation lets us gain some extra speedup.

We then look into the issue of randomization of our batch-verification algorithms in order to avoid several attacks. We work out three methods to achieve this: a numeric method, a method based on Montgomery ladders, and a seminumeric method. Randomization reduces the performance gains achieved by batch verification. Nevertheless, we can achieve non-negligible speedup even at the meaningfully highest security level.

We finally port our batch-verification and randomization methods to the EdWards Curve Digital Signature Algorithm (EdDSA).

Our algorithms are useful when all signatures in a batch belong to the same signer. Moreover, these algorithms are exponential in the batch size and so are practical only for small batches (of size $\leqslant 10$). Designing batch-verification algorithms to work for larger batch sizes and to gracefully handle the case of different signers turns out to be the most relevant open problems arising out of our study.

1.3 Thesis Organization

The rest of the dissertation is organized as follows.

Chapter 2 provides a survey of related published research works on common digital signature schemes and their batch-verification algorithms.

Chapter 3 describes our naive and symbolic-computation algorithms (N, S1, S2), and their faster variants (S1′, S2′). In this chapter, we provide detailed analysis of the algorithms, and the experimental results for prime curves and Koblitz curves.

Chapter 4 introduces our algorithms related to summation polynomials (SP, S3). A theoretical and experimental study is made on these algorithms, and a connection between these two algorithms is established. The practically improved gcd-based variants are also studied.

Chapter 5 deals with the randomization of the proposed algorithms. We introduce the seminumeric randomization procedure, and compare its performance with the known procedures based upon numeric and Montgomery-ladder computations.

Chapter 6 describes the adaptation of our ECDSA batch-verification algorithms to the Edwards curve digital signature algorithm (EdDSA).

Chapter 7 summarizes the work done, and concludes the thesis after mentioning some possible extensions of our work.

1.4 Chapter Summary

This chapter introduces the ECDSA batch-verification problem, which the rest of the thesis deals with. The motivations and applications of this problem are enumerated. The contributions made in the thesis are also summarized.

Chapter 2

Batch Verification of Common Digital-Signature Algorithms

In this chapter, we review the basic digital signatures like RSA and DSA public-key signature schemes (which are accepted as standards in DSS [46]), and also their batch-verification procedures. At the end of this chapter, we introduce ECDSA and the problems of batch verification using the procedure for DSA. We show how a non-standard and practically inefficient variant ECDSA* can overcome these problems. Another non-standard variant ECDSA$^\#$ is practically much better than ECDSA*, and solve the problems mentioned above like ECDSA* does. We finally mention an improved individual verification scheme for standard ECDSA signatures. This scheme applies to the case when multiple signatures from the same signer are available.

2.1 Digital Signature

The notion of *digital signatures* is one of most fundamental and useful inventions of public-key cryptography. A digital-signature scheme makes it possible that any user can *sign* a message that can be *verified* later by anyone. Each user can create a pair of *public* and *private* keys on their own, where it is infeasible to compute the private key from the available knowledge of the public key. Any user (say Alice) can sign a message using her private key and any one else (say Bob) can verify the signature using Alice's public key. If the signature gets verified, then Bob can be sure that the

message originated from Alice and has not been altered. Also, later Alice cannot deny the signature because no one but her can generate that signature.

A *signature scheme* starts with a set of predefined system parameters derived from the security parameter. The scheme should specify a tuple of algorithms (G,S,V) described as follows. Each of these algorithms has access to the system parameters.

- G is a probabilistic polynomial-time *key-generation* algorithm. It takes the security parameter as input and outputs a key pair (Pri,Pub), where Pri is called the private key and Pub is called the public key.
- S is a probabilistic polynomial-time *signing* algorithm. On the input of the private key Pri and a message m, it generates a signature s on the message.
- V is a probabilistic polynomial-time *verification* algorithm. Given the public key Pub, the digital signature s and a message m, V returns true (1) or false (0) to indicate whether the signature is valid or not.

Digital signature schemes are almost always used in conjunction with a fast cryptographic hash function. The hash function (SHA-1 [50] is the most used one, the recent standard Keccak [52] can also be used) takes a message of arbitrary length and produces a message digest e of a specified size (160 bits is a popular choice). The message digest is then used to generate the digital signature for the message. The use of a hash function in a signature scheme is depicted in Figure 2.1.

2.2 RSA Signature Scheme

RSA, named after its inventors **R**ivest, **S**hamir and **A**dleman, is proposed in 1978 shortly after the discovery of public-key cryptography. RSA is the first practically applicable signature scheme. This algorithm derives its security from the intractable problem of factoring large composite integers.

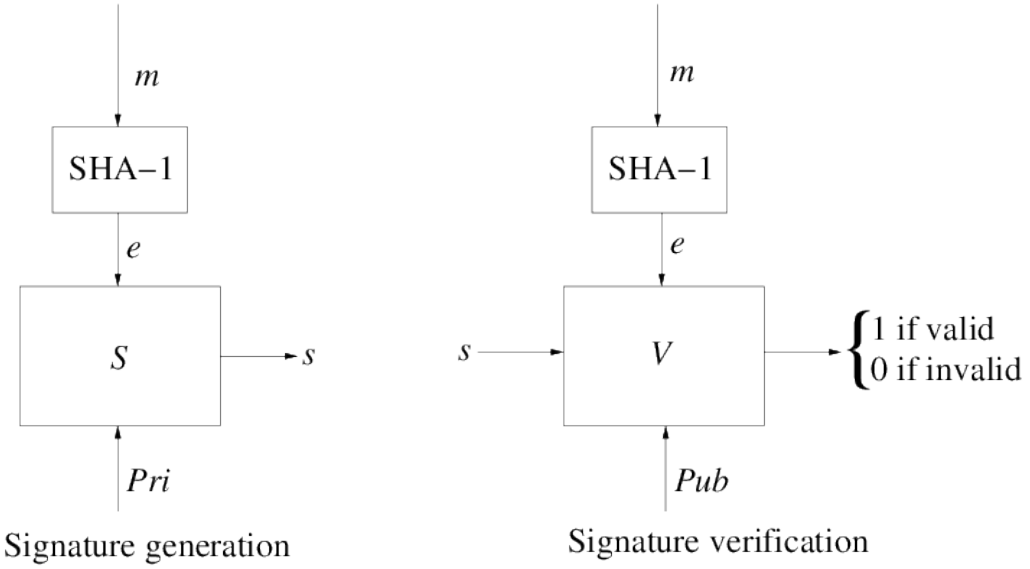

Figure 2.1: Signature generation and verification

2.2.1 RSA Key Generation

An RSA key pair can be generated using Algorithm 2.1. The public key consists of a pair of integers (n,e) where the RSA modulus n is a product of two randomly generated (and secret) primes p and q of the same bit length. The encryption exponent e is an integer satisfying $1 < e < \phi$ and gcd(e, ϕ) = 1, where $\phi = (p-1)(q-1)$. The private key d, also known as the decryption exponent, is the integer satisfying $1 < d < \phi$ and $ed \equiv 1 \pmod{\phi}$. It is proved that the problem of determining the private key d from the public key (n,e) is polynomial-time equivalent to factoring n.

2.2.2 RSA Signature Generation and Verification

The RSA signing and verifying procedures are shown in Algorithms 2.2 and 2.3. The signer of a message m first computes its message digest $h = H(m)$ using a secure cryptographic hash function H, where h serves as a short fingerprint of m. The signer uses his private key d to compute the $e-th$ root s of h modulo n: $s \equiv h^d \pmod{n}$. Upon the input of the signed message (m,s), a verifier recomputes the message digest $h = H(m)$ and recovers a message digest $h' \equiv s^e \pmod{n}$ from the signature. The signature is **accepted** if and only if $h = h'$.

Algorithm 2.1 RSA Key Pair Generation

INPUT : Security parameter l.

OUTPUT : RSA public key (n,e) and private key d.

1. Generate two large distinct random primes p and q of bit-length $\frac{l}{2}$.
2. Compute $n = pq$ and $\phi = (p-1)(q-1)$.
3. Select a random integer e with $1 < e < \phi$ and $\gcd(e,\phi) = 1$.
4. Use an extended gcd algorithm to compute the unique integer d satisfying $1 < d < \phi$ and $ed \equiv 1 \pmod{\phi}$.
5. Return (n,e,d).

Algorithm 2.2 RSA Signature Generation

INPUT : RSA exponent n, RSA private key d, message m.

OUTPUT : Signature s.

1. Compute $h = H(m)$, where H is a secure cryptographic hash function.
2. Compute $s \equiv h^d \pmod{n}$.
3. Return (s).

Algorithm 2.3 RSA Signature Verification

INPUT : RSA public key (n,e), message m, signature s.

OUTPUT : Acceptance or rejection of the signature.

1. Compute $h = H(m)$, where H is a secure cryptographic hash function.
2. Compute $h' \equiv s^e \pmod{n}$.
3. If $h = h'$ then *Accept*, else *Reject*.

The *correctness* of the verification procedure is based on that:

$$h' \equiv s^e \equiv (h^d)^e \equiv h^{ed} \equiv h \pmod{n}, \tag{2.1}$$

since $ed \equiv 1 \pmod{\phi}$. This proof holds for $h \in \mathbb{Z}_n^*$, and can be extended to elements of $\mathbb{Z}_n \setminus \mathbb{Z}_n^*$ too.

2.2.3 RSA Batch Verification

The concept of *Batch RSA* was introduced by Fiat in 1997 [21]. In this scheme, signatures on t messages under different small public keys can be generated using an effort much less than t full exponentiations. Since we deal with batch verification, this work is not directly relevant to our context.

In 1998, Harn first proposes an efficient scheme to verify batches of RSA signatures. In Harn's scheme, multiple signatures signed by the same private key can be verified simultaneously, instead of individual verification of the signatures. This batch verification performs only one exponentiation operation.

Assume that Alice signs $m_1, m_2, \ldots, m_t$ using her private key to get the signatures $s_1, s_2, \ldots, s_t$. The product $(\prod_{i=1}^{t} s_i)$ is subjected to the verification primitive to get:

$$\begin{aligned}\left(\prod_{i=1}^{t} s_i\right)^e &= \left(\prod_{i=1}^{t} h(m_i)^d\right)^e \pmod{n} \\ &= \prod_{i=1}^{t} h(m_i)^{de} \pmod{n} \\ &= \prod_{i=1}^{t} h(m_i) \pmod{n}\end{aligned}$$

If this condition is satisfied, all the signatures $s_1, s_2, \ldots, s_t$ are considered to be verified.

The problem with this scheme is that if an adversary chooses faulty signatures $s'_1, s'_2, \ldots, s'_t$ satisfying

$$\prod_{i=1}^{t} s'_i = \prod_{i=1}^{t} s_i \pmod{n},$$

then all the faulty signatures are accepted in the batch. Such attacks can be largely eliminated by randomizing the batch-verification process. We choose l-bit random

integers $\xi_1, \xi_2, \ldots, \xi_t$ and verify whether the following modified condition holds:

$$\left(\prod_{i=1}^{t} s_i^{\xi_i}\right)^e \equiv \prod_{i=1}^{t} h(m_i)^{\xi_i} \pmod{n}.$$

Bellare et al. [4] analyze that l-bit randomizers offer l-bit security. However, the additional exponentiations would significantly bring down the performance gains achieved by batch verification. If factoring n offers k-bit security in view of the best known factoring algorithms, then it suffices to take $l = k$. For example, 3072-bit RSA is supposed to offer 128-bit security, so 128-bit randomizers can be used.

2.3 Digital Signature Algorithm (DSA)

In 1984, ElGamal proposes the first digital-signature scheme based on the discrete logarithm problem. Subsequently, many ElGamal-like signatures are introduced, like Schnorr signatures [62, 63] and Nyberg-Rueppel signatures [64]. In August 1991, the US National Institute of Standards and Technology (NIST) proposes the Digital Signature Algorithm (DSA). The DSA has become a US Federal Information Processing Standard (FIPS 186) called the Digital Signature Standard (DSS). DSA is again a variant of the ElGamal scheme.

2.3.1 DSA Key-Pair Generation

In DSA, a key pair is associated with a set of public domain parameters (p,q,g). Here, p is a prime, q is a prime divisor of $(p-1)$ and $g \in [1, p-1]$ has order q, that is, $k = q$ is the smallest positive integer satisfying $g^k \equiv 1 \pmod{p}$. A long-term key pair (x,y) of an entity is generated as follows. The private key x is an integer selected uniformly at random from the interval $[1, q-1]$, and the corresponding public key is $y \equiv g^x \pmod{p}$. The problem of determining x, given the domain parameters (p,q,g) and y, is the *discrete logarithm problem*. We summarize the DSA domain-parameter generation and key-pair generation procedures in Algorithms 2.4 and 2.5.

Algorithm 2.4 DSA Domain Parameter Generation

INPUT : Security parameters l.

OUTPUT : DSA domain parameters (p,q,g).

1. Select a t-bit prime q and an l-bit prime p such that q divides $(p-1)$.
2. Select an element g of order q:
   (a) Select a random $h \in [1, p-1]$ and compute $g \equiv h^{\frac{p-1}{q}} \pmod{p}$.
   (b) If $g = 1$, then go to step 2(a).
3. Return (p,q,g).

Algorithm 2.5 DSA key Pair Generation

INPUT : DSA domain parameters (p,q,g).

OUTPUT : Public key y and private key x.

1. Select $x \in_R [1, q-1]$
2. Compute $y \equiv g^x \pmod{p}$.
3. Return (x,y).

2.3.2 DSA Signature Generation and Verification

We summarize DSA signing and verifying procedures in Algorithms 2.6 and 2.7. An entity A with private key x signs a message by selecting a random integer k (the session key) from the interval $[1, q-1]$, and computing $T \equiv g^k \pmod{p}$, $r \equiv T \pmod{q}$, and

$$s \equiv k^{-1}(h + xr) \pmod{q}, \tag{2.2}$$

where $h = H(m)$ is a message digest. The signature on m is the pair (r,s).

To verify the signature (r,s), Eqn (2.2) is first rewritten as:

$$k \equiv s^{-1}(h + xr) \pmod{q}. \tag{2.3}$$

Algorithm 2.6 DSA Signature Generation

INPUT : DSA domain parameters (p,q,g), and private key x, message m.

OUTPUT : Signature (r,s)

1. Select $k \in_R [1, q-1]$
2. Compute $T \equiv g^k \pmod{p}$.
3. Compute $r \equiv T \pmod{q}$. If $r = 0$, go to step 1.
4. Compute $h = H(m)$ (Use of SHA-1 as H is recommended by [47]).
5. Compute $s \equiv (k^{-1}(H(m) + xr)) \pmod{q}$.
6. Return (r,s).

Algorithm 2.7 DSA Signature Verification

INPUT : DSA domain parameters (p,q,g), public key y, message m, signature (r,s).

OUTPUT : Acceptance or rejection of the signature.

1. Verify that $0 < r,s < q$. If either condition is violated, then the signature is **rejected**.
2. Compute $h = H(m)$.
3. Compute $w \equiv s^{-1} \pmod{q}$.
4. Compute $u_1 \equiv hw \pmod{q}$ and $u_2 \equiv rw \pmod{q}$.
5. Compute $T \equiv g^{u_1} y^{u_2} \pmod{p}$.
6. Compute $v \equiv T \pmod{q}$.
7. If $v = r$, then the signature is **verified**, else the signature is **rejected**.

Raising g to both sides of Eqn (2.3) yields the congruence

$$T \equiv g^{hs^{-1}} y^{rs^{-1}} \pmod{p}, \tag{2.4}$$

The verifier computes T and then checks whether $r \equiv T \pmod{q}$.

2.3.3 Batch Verification Procedure of DSA Signatures

We verify a signature (r,s) on a message m by checking whether

$$\begin{aligned} r &\equiv ((g^{u_1}y^{u_2}) \pmod p) \pmod q \\ &\equiv \left((g^{H(m)w}y^{rw}) \pmod p\right) \pmod q \\ &\equiv \left((g^{H(m)s^{-1}}y^{rs^{-1}}) \pmod p\right) \pmod q. \end{aligned}$$

The above equation extends to t messages $m_1, m_2, \ldots, m_t$ with signatures (r_1,s_1), (r_2,s_2), ..., (r_t,s_t) as follows:

$$\prod_{i=1}^{t} r_i \equiv \left(g^{\Sigma_{i=1}^{t} H(m_i)s_i^{-1}} y^{\Sigma_{i=1}^{t} r_i s_i^{-1}} \pmod p\right) \pmod q.$$

This is the straightforward batch-verification equation for DSA. Like RSA, this equation suffers from certain attacks which can be largely eliminated by randomizing the last equation. We shortly describe these issues in connection with Harn's analogous batch-verification algorithm on a slight variant of DSA.

Naccache et al.'s DSA-Type Interactive Batch-Transaction Protocol

In Eurocrypt'94, Naccache et al. present a batch-transaction protocol for fast DSA verification [45]. The signer generates t signatures through interaction with the verifier, and then the verifier validates all these t signatures at once based on a batch-verifying criterion.

For each message m_i to sign, $i = 1,2,\ldots,t$, the following steps are performed:

1. The signer chooses a random integer k_i, $1 < k_i < q$.
2. The signer computes $\lambda_i \equiv g^{k_i} \pmod p$, and sends λ_i to the verifier.
3. The verifier replies with an e-bit random message b_i.
4. The signer sends $s_i \equiv {k_i}^{-1}(H(m_i||b_i) + \lambda_i x) \pmod q$ to the verifier.

The verifier uses the batch-verification criterion:

$$\prod_{i=1}^{t} \lambda_i \equiv g^{\sum_{i=1}^{t} s_i^{-1} H(m_i||b_i)} y^{\sum_{i=1}^{t} s_i^{-1} \lambda_i}.$$

Forgery Attack on Naccache et al.'s Protocol

Batch verification succeeds if and only if an attackers can find w, v, s_1, λ_1, s_2, λ_2, ..., s_t, λ_t, from the following congruences:

$$\begin{aligned} \sum_{i=1}^{t} s_i^{-1} H(m_i||b_i) &\equiv w \pmod{q}, \\ \sum_{i=1}^{t} s_i^{-1} \lambda_i &\equiv v \pmod{q}, \\ \prod_{i=1}^{t} \lambda_i &\equiv g^w y^v \pmod{p}. \end{aligned}$$

The equations can be solved easily. For example, the attacker does the following steps:

1. Choose two random integers as w, v.
2. Choose $\lambda_1, \lambda_2, \ldots, \lambda_t$ satisfying the last equation.
3. Find $s_1, s_2, \ldots, s_t$ from the first two equations (linear in each s_i^{-1}).

Obviously, $s_1, \lambda_1, s_2, \lambda_2, \ldots, s_t, \lambda_t$ so chosen satisfy Naccache et al.'s batch-verifying criterion.

Harn's DSA-Type Batch-Verifying Algorithm

In 1998, Harn proposes an algorithm to simultaneously verify t DSA-type signatures on t different messages signed by the same private key [26]. The algorithm shares the same parameters and similar signing and verification equations as in the original DSA. For each message m to be signed, the signer chooses a random integer k, $1 < k < q$, computes $r \equiv (g^k \pmod{p}) \pmod{q}$ and $s \equiv rk - H(m)x \pmod{q}$. The signature (r, s) on the message m can be verified by checking whether

$$r \equiv \left(g^{sr^{-1}} y^{H(m)r^{-1}} \pmod{p}\right) \pmod{q}.$$

Let (r_1, s_1), (r_2, s_2),..., (r_t, s_t) be signatures on t messages m_1, m_2, ..., m_t signed by the same private key x. These t signatures satisfy the following t verification equations:

$$\begin{aligned} r_1 &\equiv \left(g^{s_1 r_1^{-1}} y^{H(m_1)r_1^{-1}} \pmod{p}\right) \pmod{q} \\ r_2 &\equiv \left(g^{s_2 r_2^{-1}} y^{H(m_2)r_2^{-1}} \pmod{p}\right) \pmod{q} \\ &\vdots \\ r_t &\equiv \left(g^{s_t r_t^{-1}} y^{H(m_t)r_t^{-1}} \pmod{p}\right) \pmod{q} \end{aligned}$$

Multiplying these t equations, Harn obtains the batch-verifying criterion:

$$\prod_{i=1}^{t} r_i \equiv \left(g^{\sum_{i=1}^{t} s_i r_i^{-1}} y^{\sum_{i=1}^{t} H(m_i) r_i^{-1}} \pmod{p}\right) \pmod{q}.$$

Weakness of Harn's Algorithm

An attacker can forge a batch of signatures and can make false signatures valid. For doing that, the attacker sets the faulty signatures $s_i' = s_i + a_i r_i$ satisfying

$$\sum_{i=1}^{t} a_i = 0.$$

Individual signatures satisfy

$$r_i \not\equiv \left(g^{s_i r_i^{-1}} y^{H(m_i)r_i^{-1}} \pmod{p}\right) \pmod{q},$$

but the batch passes the verification equation as a whole.

Example

The attacker forges three signatures (s_1', r_1), (s_2', r_2), (s_3', r_3) with $a_1 = 2$, $a_2 = 3$, and

$a_3 = -5$. Batch verification gives

$$\begin{aligned}
& \left(g^{s'_1 r_1^{-1} + s'_2 r_2^{-1} + s'_3 r_3^{-1}} y^{H(m_1) r_1^{-1} + H(m_2) r_2^{-1} + H(m_3) r_3^{-1}} \pmod p\right) \pmod q \\
\equiv \; & \left(g^{(s_1+2r_1) r_1^{-1} + (s_2+3r_2) r_2^{-1} + (s_3-5r_3) r_3^{-1}} \right. \\
& \qquad \left. y^{H(m_1) r_1^{-1} + H(m_2) r_2^{-1} + H(m_3) r_3^{-1}} \pmod p\right) \pmod q \\
\equiv \; & \left(g^{s_1 r_1^{-1} + s_2 r_2^{-1} + s_3 r_3^{-1}} y^{H(m_1) r_1^{-1} + H(m_2) r_2^{-1} + H(m_3) r_3^{-1}} \pmod p\right) \pmod q \\
\equiv \; & r_1 r_2 r_3 \pmod q
\end{aligned}$$

This attacks on Harn's scheme can be significantly eliminated by using randomizers $\xi_1, \xi_2, \ldots, \xi_t$. We now combine the t individual verification equations as:

$$\prod_{i=1}^{t} r_i^{\xi_i} \equiv \left(g^{\sum_{i=1}^{t} \xi_i s_i r_i^{-1}} y^{\sum_{i=1}^{t} \xi_i H(m_i) r_i^{-1}} \pmod p\right) \pmod q.$$

A successful forgery now requires

$$\sum_{i=1}^{t} \xi_i a_i = 0.$$

For sufficiently long $\xi_1, \xi_2, \ldots, \xi_t$ chosen randomly during verification, that is, after the signatures are prepared by the attacker, this condition holds with negligibly small probability. Indeed, if we use l-bit randomizers, the probability of having $\sum_{i=1}^{t} \xi_i a_i = 0$ is 2^{-l}.

2.4 Elliptic Curve Digital Signature Algorithm (ECDSA)

The *Elliptic Curve Digital Signature Algorithm* (ECDSA) is the elliptic curve analog to the Digital Signature Algorithm (DSA). It is the most widely standardized elliptic-curve-based signature scheme, appearing in the ANSI X9.62, FIPS 186-2, IEEE 1363-2000 and ISO/IEC 15946 standards [1, 30, 31, 32, 49] and also in several draft standards. See [9, 16, 25] to know about the arithmetic of elliptic curves.

The hardness of the *Elliptic Curve Discrete Logarithm Problem* (ECDLP) [16] is essential for the security of the ECDSA scheme. The elliptic-curve parameters for cryptographic schemes should be carefully chosen so that the ECDLP cannot

be solved in less than fully exponential time. At present, the fastest algorithms for solving general instances of the ECDLP are called square-root methods [55, 56, 68]. For special classes of curves, faster—even polynomial-time—algorithms are known [37, 61, 65, 66].

2.4.1 ECDSA Domain Parameters

For simplicity, we now restrict to curves defined over prime fields. ECDSA is based upon some parameters common to all entities participating in a network.

q = Order of the prime field $\mathbb{F}_q$.

E = An elliptic curve $y^2 = x^3 + ax + b$ defined over the prime field $\mathbb{F}_q$.

P = A random non-zero base point in $E(\mathbb{F}_q)$.

n = The order of P, typically a prime.

h = The cofactor $\dfrac{|E(\mathbb{F}_q)|}{n}$.

We refer the reader to [25] for the detailed description of the procedures for choosing these parameters.

2.4.2 Assumptions

For the time being, we assume that $h = 1$, that is, $E(\mathbb{F}_q)$ is a cyclic group, and P is a generator of $E(\mathbb{F}_q)$. This is indeed the case for certain elliptic curves standardized by NIST. By Hasse's theorem, we have $|n - q - 1| \leqslant 2\sqrt{q}$. If $n \geqslant q$, an element of $\mathbb{Z}_n$ has a unique representation in $\mathbb{Z}_q$. On the other hand, if $n < q$, an element of $\mathbb{Z}_n$ has at most two representations in $\mathbb{Z}_q$. The density of elements of $\mathbb{Z}_n$ having two representations in $\mathbb{Z}_q$ is $\leqslant 2/\sqrt{q}$ which is close to zero for large values of q.

In an ECDSA signature (r, s), the values r and s are known modulo n. However, r corresponds to an elliptic-curve point and should be known modulo q. If r corresponds to a random point on E, it uniquely identifies an element of $\mathbb{F}_q$ with probability close to 1. In view of this, we ignore the effect of issues associated with the ambiguous representation stated above, in the rest of this thesis.

Note that the ambiguities arising out of $h > 1$ and/or $q > n$ can be practically solved by appending only a few extra bits to standard ECDSA signatures [2, 14]. Consequently, our assumptions are neither too restrictive nor too impractical.

2.4.3 The ECDSA Algorithm

Algorithm 2.8 computes the public key Q and the private key d of a signer. The ECDSA signature (r,s) on a message M is generated by Algorithm 2.9. Algorithm 2.10 verifies the ECDSA signature (r,s) on a message M.

Algorithm 2.8 ECDSA Key-pair Generation

INPUT: Domain Parameters.

OUTPUT: Public key Q, private key d.

1. The private key $= d \in_R [1, n-1]$.
2. The public key $Q = dP \in E(\mathbb{F}_q)$.
3. Return (d, Q).

Algorithm 2.9 ECDSA Signature Generation

INPUT: Domain Parameters, message M and signer's private key d.

OUTPUT: ECDSA signature (r,s).

1. $k =$ A randomly chosen element in the range $[1, n-1]$ (the session key).
2. $R = kP$.
3. $r = x(R)$ (the x-coordinate of R) reduced modulo n.
4. $s = k^{-1}(H(M) + dr) \pmod{n}$, where H is a cryptographic hash function like SHA-1 of [50].
5. Return (r,s).

Algorithm 2.10 ECDSA Signature Verification

INPUT: Domain Parameters, message M, signature (r,s) and signer's public key Q.

OUTPUT: Accept/Reject.

1. $w \equiv s^{-1} \pmod{n}$.
2. $u \equiv H(M)w \pmod{n}$.
3. $v \equiv rw \pmod{n}$.
4. $R = uP + vQ \in E(\mathbb{F}_q)$. (2.5)
5. Accept the signature if and only if $x(R) = r \pmod{n}$.

2.4.4 ECDSA Batch Verification

For t signed messages (M_i, r_i, s_i), $i = 1, 2, \ldots, t$, we have

$$\sum_{i=1}^{t} R_i = \left(\sum_{i=1}^{t} u_i\right) P + \sum_{i=1}^{t} v_i Q_i. \tag{2.6}$$

If all the signatures belong to the same signer, we have $Q_1 = Q_2 = \cdots = Q_t = Q$ (say), and the last equation simplifies to:

$$\sum_{i=1}^{t} R_i = \left(\sum_{i=1}^{t} u_i\right) P + \left(\sum_{i=1}^{t} v_i\right) Q. \tag{2.7}$$

The basic idea is to compute the two sides of Eqn (2.6) or Eqn (2.7), and check for the equality. Use of these equations reduces the number of scalar multiplications from $2t$ to $[2, t+1]$, where 2 corresponds to the case where all the signatures belong to same signer, and $t+1$ corresponds to the case where the t signers are distinct from one another. However, only the x-coordinates of R_i are known from the signatures. In general, there are two y-coordinates corresponding to a given x-coordinate, but computing these y-coordinates requires taking square roots modulo q, which is a time-consuming operation. Moreover, there is nothing immediately available in the signatures to remove the resulting ambiguity in these two values of y. Finally, computing all R_i using Eqn (2.5) misses the basic idea of batch verification, since after this expensive computation, there is only an insignificant amount of effort left to complete individual verifications of all the t signatures.

2.4.5 ECDSA*

ECDSA*, a modification of ECDSA introduced by Antipa et al. [2], adapts readily to the above batch-verification idea. In ECDSA*, the x-coordinate r is replaced by the entire point R in the signature. The ECDSA* signature (R,s) on a message M is computed by Algorithm 2.11. Algorithm 2.12 verifies the ECDSA* signature (R,s) on a message M.

Algorithm 2.11 ECDSA* Signature Generation

INPUT: Domain Parameters, message M and signer's private key d.

OUTPUT: ECDSA signature (R,s).

1. k = A randomly chosen element in the range $[1,n-1]$ (the session key).
2. $R = kP$.
3. $s \equiv k^{-1}(H(M)+x(R)d) \pmod n$ (where H is a hash function).
4. Return (R,s).

Algorithm 2.12 ECDSA* Signature Verification

INPUT: Domain Parameters, message M, signature (R,s) and signer's public key Q.

OUTPUT: Accept/Reject.

1. $w \equiv s^{-1} \pmod n$.
2. $u \equiv H(M)w \pmod n$.
3. $v \equiv rw \pmod n$, where $r = x(R)$.
4. $R' \equiv uP + vQ \in E(\mathbb{F}_q)$.
5. Accept the signature if and only if $R' = R$.

Batch verification of ECDSA* signatures is a straightforward adaptation of the procedure of [45]. One needs to check whether Eqn (2.6) (or Eqn (2.7)) holds. Since the entire points R_i are present in the signatures, the left side of this equation can be computed efficiently and unambiguously. But ECDSA* is not accepted as a standard and results in increased signature sizes, so an algorithm that efficiently performs batch

verification in presence of only the partial information r_i (only the x-coordinates of R_i) turns out to be practically useful.

2.4.6 ECDSA$^{\#}$

In this section, we introduce another non-standard variant of ECDSA, where we include one extra bit in the ECDSA signature (r,s) to identify R uniquely. We henceforth call this algorithm as ECDSA$^{\#}$. In general, the two possible y-coordinates of R are $\pm y$, where $y^2 \equiv r^3 + ar + b \pmod{q}$. We assume that $y \neq 0$. There are many ways by which we can discriminate between the two roots. For example, one of $\pm y$ must be odd, and the other even. Another possibility is that one of $\pm y$ is less than $q/2$, and the other larger than $q/2$. Only one bit suffices to identify whether $+y$ or $-y$ is the y-coordinate of R.

An ECDSA$^{\#}$ signature (r,s,id) on a message M is generated by Algorithm 2.13. Algorithm 2.14 verifies this signature. The square-root computation in Steps 1 and 2 of the verification algorithm is unnecessary. We can use the standard ECDSA verification Algorithm 2.10. The verification presented as Algorithm 2.14 is suited to the context of batch verification.

Algorithm 2.13 ECDSA$^{\#}$ Signature Generation

INPUT: Domain Parameters, message M and signer's private key d.

OUTPUT: ECDSA signature (r,s,id).

1. k = A randomly chosen element in the range $[1, n-1]$ (the session key).
2. $R = kP$.
3. $r = x(R)$ (the x-coordinate of R) reduced modulo n.
4. $s \equiv k^{-1}(H(M) + dr) \pmod{n}$ (where H is a hash function).
5. If $\left(y(R) < \frac{q}{2}\right)$, then $id = 0$, else $id = 1$.
6. Return (r,s,id).

Algorithm 2.14 ECDSA$^{\#}$ Signature Verification

INPUT: Domain Parameters, message M, signature (r,s,id) and signer's public key Q.

OUTPUT: Accept/Reject.

1. Compute the two square roots of r^3+ar+b.
2. Use id to uniquely identify the correct square root y, and set $R=(r,y)$.
3. $w \equiv s^{-1} \pmod{n}$.
4. $u \equiv H(M)w \pmod{n}$.
5. $v \equiv rw \pmod{n}$.
6. $R' = uP+vQ \in E(\mathbb{F}_q)$.
7. Accept the signature if and only if $R'=R$.

2.4.7 The Limits of Individual ECDSA Verification

The computation of $uP+vQ$ is the main overhead of ECDSA signature verification. To speed up individual verification, we need to reduce the computation time of this. Let u and v be two scalars of length l bits. Each individual scalar multiplication performs l point doubling and an average of $l/2$ point additions. Therefore, two individual scalar multiplications require $2l$ point doubling and l point additions. There are several ways this can be improved upon.

One possibility is to use multiple (double) scalar multiplication. We precompute the point $P+Q$, and run only one double-and-add loop to consume the bits of u and v together. After each unconditional doubling, we look at the next bits of u and v. If both are zero, we move on to the next iteration. Otherwise, we add P, Q or $P+Q$ according as whether the bits are $(1,0)$, $(0,1)$ or $(1,1)$, respectively. A double scalar multiplication carries out l point doubling and $3l/4$ points additions on an average. So the total count of point operations reduces from $3l$ to $1.75l$—a significant saving.

Another idea is to use fixed-base scalar multiplication.[1] Since P is a system-

[1] This is suggested by an anonymous referee of one of our papers on ECDSA batch verification.

wide constant, we can precompute and store 2^iP for $i = 0,1,2,\ldots,l$. But then, the computation of uP does not require any point doubling at all, that is, only $l/2$ point additions suffice. The point Q is, on the other hand, not a constant and varies dynamically from signature to signature. If, however, several signatures having a common Q are available, then it may be worthwhile to precompute 2^iQ for $i = 0,1,2,\ldots,l$. By doing so, we can speed up the computation of vQ. The amortized overhead of the precomputation depends on how many signatures have the common Q. If this count is four or more, then the precomputation seems to pay off.

We can combine the above two ideas to arrive at double fixed-base scalar multiplication. In addition to the precomputation tables of 2^iP and 2^iQ, we now require an additional precomputation table storing $2^i(P+Q)$. We now require only $3l/4$ point additions, and no point doubling. The total cost of preparing the precomputations tables for 2^iQ and $2^i(P+Q)$ is $2l$ point doubling. If t signatures have the common Q, then the overhead of precomputation per signature verification is $2l/t$. For example, if $t = 6$, then the total average cost of each verification is $\frac{3l}{4} + \frac{2l}{6} = \frac{13l}{12} \approx 1.083l$ (much better than $1.75l$ as in double multiplication without the precomputation tables).

The main drawback of the fixed-base approach is the requirement of a large amount of memory to store the precomputation tables. The table for 2^iP permanently resides in the memory. The tables for 2^iQ and $2^i(P+Q)$ are dynamically generated upon encountering a new Q, and discarded after all signatures involving this Q are verified. Since our applications typically deal with resource-constrained processors, such a huge memory may be unavailable, If so, the plain double multiplication is the best choice, since it needs the precomputation of only one point $P+Q$.

In this thesis, all our reported batch-verification speedup figures are relative to individual signature verification. We use double multiplication both with and without precomputation tables to set the best running times of individual verification.

2.4.8 The Setting of Our Work on ECDSA Batch Verification

Before we present our algorithms for ECDSA batch verification, we highlight the basic assumptions we make about the domains of applicability of these algorithms.

1. Our main goal is to verify *standard* ECDSA signatures in batches. By standard ECDSA signatures, we mean that we do not have any extra information to identify the correct square root of $r^3 + ar + b \pmod{q}$. For the sake of completeness, we also study the batch-verification algorithms for the non-standard variant ECDSA$^{\#}$ of ECDSA. We, however, do not deal with ECDSA*.

2. For all the ECDSA signatures, P is fixed as it is a domain parameter. But Q is assumed fixed only in a batch and varies across different batches. Consequently, precomputations on Q should be considered as a part of the overhead associated with individual or batch verification.

3. We deal with resource-constrained environments, so any speedup in the verification process is helpful. Moreover, the storage of massive precomputed data may be infeasible. Some of our initially proposed algorithms are demanding in terms of memory. But our best proposals are not so.

4. Our basic aim is to investigate whether square-root computations can be replaced by faster techniques. ECDSA$^{\#}$ using square-root computations scales to arbitrarily large batch sizes. For standard ECDSA and even for ECDSA$^{\#}$ with small batches, we seek for faster alternatives.

5. In the next two chapters, we deal with non-randomized batch verification. All issues pertaining to randomization are dealt with separately in Chapter 5.

2.5 Chapter Summary

This chapter presents a survey of batch-verification algorithms for RSA, DSA and ECDSA. Two non-standard variants of ECDSA are introduced. An idea to speed up individual verification with common public key is explained. The chapter ends with the basic assumptions about the settings of the work reported in the rest of the thesis.

Chapter 3

ECDSA Batch Verification Using Symbolic Computations

In this chapter, we propose three algorithms to verify *standard* ECDSA signatures in batches. Our algorithms apply to all cases of ECDSA signatures sharing the same curve parameters, although we obtain good speedup figures when all the signatures in the batch come from the same signer. Our algorithms are effective only for small batch sizes (like $t \leqslant 8$). The first algorithm we introduce (henceforth denoted as Algorithm N) is based upon a naive approach of taking square roots in the underlying field. As the field size increases, square-root computations become quite costly. We modify Algorithm N by replacing square-root calculations by symbolic manipulations. We propose two ECDSA batch-verification algorithms, called S1 and S2, based upon symbolic manipulations. Algorithm S1 is not very practical. But Algorithm S1 is discussed in this chapter, for it provides the theoretical and practical foundations for arriving at Algorithm S2. For a wide range of field and batch sizes, Algorithm S2 convincingly outperforms the naive Algorithm N. Both S1 and S2 are probabilistic algorithms in the Monte Carlo sense, that is, they may occasionally fail to verify correct signatures. We analytically establish that for randomly generated signatures, the failure probability is extremely low.

3.1 Algorithm N and N′

ECDSA verification is based upon the equality:

$$R \;=\; uP + vQ \in E(\mathbb{F}_q). \tag{3.1}$$

Let there be t signed messages (M_i, r_i, s_i), $i = 1, 2, \ldots, t$. We have

$$\sum_{i=1}^{t} R_i = \left(\sum_{i=1}^{t} u_i \right) P + \sum_{i=1}^{t} v_i Q_i. \tag{3.2}$$

If all the signatures belong to the same signer, we have $Q_1 = Q_2 = \cdots = Q_t = Q$ (say), and the last equation simplifies to:

$$\sum_{i=1}^{t} R_i = \left(\sum_{i=1}^{t} u_i \right) P + \left(\sum_{i=1}^{t} v_i \right) Q. \tag{3.3}$$

The basic idea is to compute the two sides of Eqn (3.2) or Eqn (3.3), and check for the equality. For a given x-coordinate there exist two y-coordinates which satisfy Eqn (3.1). Computation of these y-coordinates requires taking square roots modulo q which is a time-consuming operation. Moreover, there is an ambiguity in these two values of y. Nevertheless, a naive algorithm for the batch verification of original ECDSA signatures can be conceived of, as illustrated in Algorithm 3.1. This is an obvious way of solving the ECDSA batch-verification problem, but we have not found any previous mention of this algorithm in the literature. There are (usually) 2^t choices of the square roots y_i of $r_i^3 + ar_i + b$ for all $i = 1, 2, \ldots, t$. If any of these combinations of square roots satisfies Eqn (3.2), we accept the batch of signatures. Step 6 turns out to be a costly operation. Moreover, Step 7 needs to check (at most) $m = 2^t$ possible conditions for batch verification, and is also quite costly unless t is small.

By using one extra bit of information in an ECDSA signature, ECDSA$^{\#}$ can unambiguously identify the *correct* square root of $r_i^3 + ar_i + b$, and thereby avoid the $\Theta(2^t)$ overhead associated with Algorithm N. In that case, the Step 7 of Algorithm 3.1 needs to be updated appropriately. This updated (and efficient) version of the naive algorithm will henceforth be denoted by **Algorithm N′**. Despite this updating, there is apparently nothing present in ECDSA signatures, that provides a support for quickly *computing* the correct square root. The basic aim of this chapter is to develop algorithms to reduce the overhead associated with square-root calculations. In effect,

Algorithm 3.1 ECDSA Batch-verification Algorithm N

INPUT: Domain Parameters, messages $M_1, M_2, \ldots, M_t$, corresponding signatures $(r_1, s_1), (r_2, s_2), \ldots, (r_t, s_t)$ and public keys $Q_1, Q_2, \ldots, Q_t$ of the signers.

OUTPUT: Accept/Reject all the signatures

1. Compute $w_i = s_i^{-1} \pmod{n}$ for all $i = 1, 2, \ldots, t$.
2. Compute $u_i = H(M_i)w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
3. Compute $v_i = r_i w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
4. Compute $R' = (\sum_{i=1}^{t} u_i)P + \sum_{i=1}^{t} v_i Q_i \in E(\mathbb{F}_q)$.

   Club together the points Q_i from same signers during the computation of R'. For example, if all the signatures belong to the same signer, compute R' as $(\sum_{i=1}^{t} u_i)P + (\sum_{i=1}^{t} v_i)Q$.
5. For each $i = 1, 2, \ldots, t$, if $r_i^3 + ar_i + b$ is neither zero nor a quadratic residue modulo q, reject the i-th signature, and remove it from the batch.
6. For $i = 1, 2, \ldots, t$, compute the square roots of $r_i^3 + ar_i + b$ modulo q.
7. For each square root y_i of $r_i^3 + ar_i + b$ for all $i = 1, 2, \ldots, t$, do the following:

   If $R' = \sum_{i=1}^{t} (r_i, y_i)$, accept all the signatures.
8. Reject all the signatures.

we are converting ECDSA signatures to ECDSA* signatures. In that sense, our work is not competing with but complementary to the earlier works on ECDSA* [2, 14].

3.2 Algorithm S1

In this section, we present a new algorithm to convert Eqn (3.2) or Eqn (3.3) to a form which eliminates the problems associated with the lack of knowledge of the y-coordinates of R_i. We compute the right side of Eqn (3.2) or Eqn (3.3) as efficiently as possible. The left side is not computed explicitly, but symbolically in the unknown values $y_1, y_2, \ldots, y_t$ (the y-coordinates of $R_1, R_2, \ldots, R_t$). By solving a system of linear equations over $\mathbb{F}_q$, we obtain enough information to verify the batch of t signatures

simultaneously. This new algorithm, called Algorithm S1, turns out to be faster than Algorithm N for small batch sizes (typically for $t \leqslant 4$) and for large underlying fields.

3.2.1 Symbolic Computation of $R = \sum_{i=1}^{t} R_i$

Let $R_i = (x_i, y_i)$. The x-coordinates $x_i = x(R_i)$ are available from the signatures, namely, $x_i = r_i$ or $x_i = r_i + n$. The second case pertains to the condition $n < q$ and has a very low probability. So we plan to ignore this case, and take $x_i = x(R_i) = r_i$. It is indeed easy to detect when the reduced x-coordinate r_i has two representatives in $\mathbb{F}_q$, and if so, we repeat Algorithm S1 for both these values.

Although the y-coordinate $y_i = y(R_i)$ is unknown to us, we know the value

$$y_i^2 = r_i^3 + ar_i + b \pmod{q} \tag{3.4}$$

for all $i = 1, 2, \ldots, t$, since $R_i = (r_i, y_i)$ is a point on the curve E.

Let $P_1 = (h_1, k_1)$ and $P_2 = (h_2, k_2)$ be two non-zero points on E with $P_1 \neq \pm P_2$. The sum $P_1 + P_2$ is another point (h, k) on E computed as follows:

$$\lambda = (k_2 - k_1)/(h_2 - h_1), \tag{3.5}$$

$$h = \lambda^2 - h_1 - h_2, \tag{3.6}$$

$$k = \lambda(h_1 - h) - k_1. \tag{3.7}$$

Applying this formula repeatedly lets us have the following representation of the point $R = \sum_{i=1}^{t} R_i$:

$$R = \left(\frac{g_x(y_1, y_2, \ldots, y_t)}{h_x(y_1, y_2, \ldots, y_t)}, \frac{g_y(y_1, y_2, \ldots, y_t)}{h_y(y_1, y_2, \ldots, y_t)} \right), \tag{3.8}$$

where g_x, g_y, h_x and h_y are polynomials in $\mathbb{F}_q[y_1, y_2, \ldots, y_t]$. In view of Eqn (3.4), we may assume that these polynomials have y_i-degrees $\leqslant 1$ for all $i = 1, 2, \ldots, t$. This implies that the denominator $h_x(y)$ is of the form $u(y_2, y_3, \ldots, y_t)y_1 + v(y_2, y_3, \ldots, y_t)$. Multiplying both g_x and h_x by $u(y_2, y_3, \ldots, y_t)y_1 - v(y_2, y_3, \ldots, y_t)$ and making use of Eqn (3.4), we can eliminate y_1 from the denominator. Repeating this successively for $y_2, y_3, \ldots, y_t$ allows us to represent the point R as a pair of polynomial expressions:

$$R = (R_x(y_1, y_2, \ldots, y_t), R_y(y_1, y_2, \ldots, y_t)) \tag{3.9}$$

with the polynomials R_x and R_y linear individually with respect to all y_i. It is useful to clear the denominator after every symbolic addition instead of only once after the entire sum $R = \sum_{i=1}^{t} R_i$ is computed symbolically. A pseudocode for the symbolic-addition procedure is supplied as Algorithm 3.3 which uses Algorithm 3.2 for clearing the denominator of rational functions.

Algorithm 3.2 Denominator Clearing in a Rational Function

INPUT: Rational function f/g with $f, g \in \mathbb{F}_q[y_j, y_{j+1}, \ldots, y_k]$, and x-coordinates $r_j, r_{j+1}, \ldots, r_k$.

OUTPUT: A polynomial in $\mathbb{F}_q[y_j, y_{j+1}, \ldots, y_k]$ equal to f/g.

STEPS

For $i = j, j+1, \ldots, k$, repeat the following steps:

1. Express $g = uy_i + v$ with $u, v \in \mathbb{F}_q[y_{i+1}, y_{i+2}, \ldots, y_k]$.
2. If $u = 0$ (that is, if y_i does not appear in g), go to the top of the loop.
3. Set $g = u^2 \times (r_i^3 + ar_i + b) - v^2$.
4. For $i' = i+1, i+2, \ldots, k$, substitute $y_{i'}^2$ by $r_{i'}^3 + ar_{i'} + b$ in g.
5. Set $f = f \times (uy_i - v)$.
6. For $i' = j, j+1, \ldots, k$, substitute $y_{i'}^2$ by $r_{i'}^3 + ar_{i'} + b$ in f.

Return f/g. (After the above loop, $f \in \mathbb{F}_q[y_j, y_{j+1}, \ldots, y_k]$, and $g \in \mathbb{F}_q$.)

3.2.2 Properties of R_x and R_y

Theorem 3.2.1. *R_x consists of only even-degree monomials, and R_y consists of only odd-degree monomials in the variables $y_1, y_2, \ldots, y_t$.*

Proof. We proceed by induction on the batch size $t \geqslant 1$. If $t = 1$ (this amounts to individual verification), we have $R_x = r_1$ and $R_y = y_1$, for which the theorem evidently holds.

Algorithm 3.3 Symbolic Addition of Elliptic-curve Points

INPUT: x-coordinates $r_j, r_{j+1}, \ldots, r_k \in \mathbb{F}_q$.

OUTPUT: Symbolic sum $(R_x, R_y) = \sum_{i=j}^{k} (r_i, y_i)$ with $R_x, R_y \in \mathbb{F}_q[y_j, y_{j+1}, \ldots, y_k]$, $(r_i, y_i) \in E(\mathbb{F}_q)$.

STEPS

1. If $j = k$, return (r_j, y_j).
2. Set $\tau = \lfloor (j+k)/2 \rfloor$.
3. Recursively compute $(R_x^{(1)}, R_y^{(1)}) = \sum_{i=j}^{\tau} R_i$.
4. Recursively compute $(R_x^{(2)}, R_y^{(2)}) = \sum_{i=\tau+1}^{k} R_i$.
5. Compute $\lambda = (R_y^{(2)} - R_y^{(1)})/(R_x^{(2)} - R_x^{(1)})$ as a rational function in $y_j, y_{j+1}, \ldots, y_k$.
6. Apply denominator clearing (Algorithm 3.2) on λ.
7. Set $R_x = \lambda^2 - R_x^{(1)} - R_x^{(2)}$.
8. For $i = j, j+1, \ldots, k$, substitute y_i^2 by $r_i^3 + ar_i + b$ in R_x.
9. Set $R_y = \lambda \times (R_x^{(1)} - x) - R_y^{(1)}$.
10. For $i = j, j+1, \ldots, k$, substitute y_i^2 by $r_i^3 + ar_i + b$ in R_y.
11. Return (R_x, R_y).

So assume that $t \geqslant 2$. We compute $R = \sum_{i=1}^{t} R_i$ as $R' + R''$ with $R' = \sum_{i=1}^{\tau} R_i$ and $R'' = \sum_{i=\tau+1}^{t} R_i$ for some τ in the range $1 \leqslant \tau \leqslant t-1$. Let $R' = (R'_x, R'_y)$ and $R'' = (R''_x, R''_y)$. The inductive assumption is that all non-zero terms of R'_x and R''_x are of even degrees (in $y_1, \ldots, y_\tau$ and $y_{\tau+1}, \ldots, y_t$, respectively), and all non-zero terms of R'_y and R''_y are of odd degrees.

We first symbolically compute $\lambda = (R''_y - R'_y)/(R''_x - R'_x)$ as a rational function. Clearing the variables y_i from the denominator multiplies both the numerator and the denominator of λ by polynomials of non-zero terms having even degrees. Every substitution of y_i^2 by the field element $r_i^3 + ar_i + b$ reduces the y_i-degree of certain terms by 2, so the parity of the degrees in these terms is not altered. Finally,

λ becomes a polynomial with each non-zero term having odd degree. But then, $R_x = \lambda^2 - R'_x - R''_x$ is a polynomial with each non-zero term having even degree, whereas $R_y = \lambda(R'_x - R_x) - R'_y$ is a polynomial with each non-zero term having odd degree. Further substitutions of y_i^2 by $r_i^3 + ar_i + b$ to simplify R_x and R_y preserve these degree properties. □

We have established that R_x is a polynomial with each non-zero term having even total degree, whereas R_y is a polynomial with each non-zero term having odd total degree.

From the right side of Eqn (3.2) or Eqn (3.3), we compute the x- and y-coordinates of R as

$$R = (\alpha, \beta)$$

for some $\alpha, \beta \in \mathbb{F}_q$. This gives us two multivariate equations to start with:

$$R_x(y_1, y_2, \ldots, y_t) = \alpha, \tag{3.10}$$

$$R_y(y_1, y_2, \ldots, y_t) = \beta. \tag{3.11}$$

3.2.3 Solving the Multivariate Equations

We treat Eqns (3.10) and (3.11) as linear equations in the monomials y_i, $y_i y_j$, $y_i y_j y_k$, and so on. R_x contains non-zero terms involving only the even-degree monomials, that is, $y_i y_j$, $y_i y_j y_k y_l$, and so on. Throughout the rest of this chapter, we denote $m = 2^t$. There are exactly $\mu = 2^{t-1} - 1 = \frac{m}{2} - 1$ such monomials. We name these monomials as $z_1, z_2, \ldots, z_\mu$, and take out the constant term from R_x to rewrite Eqn (3.10) as

$$\rho_{1,1} z_1 + \rho_{1,2} z_2 + \cdots + \rho_{1,\mu} z_\mu = \alpha_1. \tag{3.12}$$

If we square both sides of this equation, and use Eqn (3.4) to eliminate all squares of variables, we obtain another linear equation:

$$\rho_{2,1} z_1 + \rho_{2,2} z_2 + \cdots + \rho_{2,\mu} z_\mu = \alpha_2. \tag{3.13}$$

By repeated squaring, we generate a total of μ linear equations in $z_1, z_2, \ldots, z_\mu$. We then solve the resulting system and obtain the values of $z_1, z_2, \ldots, z_\mu$.

If the system is not of full rank, we make use of Eqn (3.11) as follows. Each non-zero term in R_y has odd degree. However, the equation $R_y^2 = \beta^2$ (along with the substitution given by Eqn (3.4)) leads to a linear equation in the even-degree monomials $z_1, z_2, \ldots, z_\mu$ only. Repeated squaring of this equation continues to generate a second sequence of linear equations in $z_1, z_2, \ldots, z_\mu$.

We expect to be able to obtain μ linearly independent equations from these two sequences.

3.2.4 A Strategy for Faster Equation Generation

There are indeed other ways of generating new linear equations in $z_1, z_2, \ldots, z_\mu$. Let

$$\rho_1 z_1 + \rho_2 z_2 + \cdots + \rho_\mu z_\mu = \gamma \tag{3.14}$$

be an equation already generated, and let $f(z_1, z_2, \ldots, z_\mu)$ be any $\mathbb{F}_q$-linear combination of the monomials $z_1, z_2, \ldots, z_\mu$. Simplification of the equation

$$(\rho_1 z_1 + \rho_2 z_2 + \cdots + \rho_\mu z_\mu) f(z_1, z_2, \ldots, z_\mu) = \gamma f(z_1, z_2, \ldots, z_\mu)$$

using Eqn (3.4) again yields a linear equation in $z_1, z_2, \ldots, z_\mu$. The particular choice $f(z_1, z_2, \ldots, z_\mu) = z_i$ with a small degree of z_i leads to a faster generation of a new equation than squaring Eqn (3.14). Our experiments indicate that we can generate a full-rank linearized system by monomial multiplications and a few squaring operations. Moreover, only Eqn (3.10) suffices to generate a uniquely solvable linearized system.

Examples

First, consider $t = 2$. In this case, R_x contains only one unknown value $y_1 y_2$, and the equation $R_x = \alpha$ can be immediately solved to obtain $y_1 y_2$.

For $t = 3$, we have 3 unknown monomials $y_1 y_2$, $y_1 y_3$ and $y_2 y_3$. Three independent linear equations are needed to solve for these values.

For $t = 4$, we have 7 unknown monomials $y_1 y_2$, $y_1 y_3$, $y_1 y_4$, $y_2 y_3$, $y_2 y_4$, $y_3 y_4$ and

$y_1y_2y_3y_4$. We need seven linearly independent equations in these variables.

Table 3.1: List of monomials in R_x for batch sizes $t \leqslant 6$

Batch size (t)	**No. of monomials (μ)**	**Monomials** $(z_1, z_2, \ldots, z_\mu)$
2	1	y_1y_2
3	3	y_1y_2, y_1y_3, y_2y_3
4	7	y_1y_2, y_1y_3, y_1y_4, y_2y_3, y_2y_4, y_3y_4, $y_1y_2y_3y_4$
5	15	y_1y_2, y_1y_3, y_1y_4, y_1y_5, y_2y_3, y_2y_4, y_2y_5, y_3y_4, y_3y_5, y_4y_5, $y_1y_2y_3y_4$, $y_1y_2y_3y_5$, $y_1y_2y_4y_5$, $y_1y_3y_4y_5$, $y_2y_3y_4y_5$
6	31	y_1y_2, y_1y_3, y_1y_4, y_1y_5, y_1y_6, y_2y_3, y_2y_4, y_2y_5, y_2y_6, y_3y_4, y_3y_5, y_3y_6, y_4y_5, y_4y_6, y_5y_6, $y_1y_2y_3y_4$, $y_1y_2y_3y_5$, $y_1y_2y_3y_6$, $y_1y_2y_4y_5$, $y_1y_2y_4y_6$, $y_1y_2y_5y_6$, $y_1y_3y_4y_5$, $y_1y_3y_4y_6$, $y_1y_3y_5y_6$, $y_1y_4y_5y_6$, $y_2y_3y_4y_5$, $y_2y_3y_4y_6$, $y_2y_3y_5y_6$, $y_2y_4y_5y_6$, $y_3y_4y_5y_6$, $y_1y_2y_3y_4y_5y_6$

Table 3.1 lists the even-degree monomials for all the values of t in the range $2 \leqslant t \leqslant 6$. Table 3.2 proposes possible sequences of monomial multiplication and squaring for generating the linearized systems.

3.2.5 Retrieving the Unknown y-coordinates

The final step in Algorithm S1 involves the determination of the y-coordinates y_i of the points R_i. Multiplying both sides of Eqn (3.11) by y_1 gives an equation of the form

$$\beta y_1 = \varepsilon_0 + \varepsilon_1 z_1 + \varepsilon_2 z_2 + \cdots + \varepsilon_\mu z_\mu.$$

Substitution of the values of z_i available from the previous stage gives y_1 (provided that $\beta \neq 0$). Subsequently, the values y_i for $i = 2, 3, \ldots, t$ can be obtained by dividing the known value of y_1y_j by y_1 provided that $y_1 \neq 0$. Even if $y_1 = 0$, we can multiply Eqn (3.11) by y_2 to solve for y_2. If $y_2 \neq 0$, we are allowed to compute $y_i = (y_2y_i)/y_2$ for $i \geqslant 3$. If $y_2 = 0$ too, we compute y_3 by directly using Eqn (3.11), and so on. The only condition that is necessary to solve for all y_i values uniquely is $\beta \neq 0$, where β is the y-coordinate of the point on the right side of Eqn (3.2) (or Eqn (3.3)).

We finally check whether Eqn (3.4) is valid for all $i = 1, 2, \ldots, t$. If so, all the signatures are verified simultaneously. If one or more of these equations fail(s) to hold, batch verification fails.

Algorithm 3.4 describes our batch-verification Algorithm S1. In short, Algorithm S1 uniquely reconstructs the points R_i with $x(R_i) = r_i$. The computations do not involve taking modular square roots in $\mathbb{F}_q$. We also avoid computing the points $R'_i = u_iP + v_iQ_i$ needed in individual verification. The final check in Step 13 guarantees that the reconstructed points really lie on the curve. In the next section, we prove that the reconstruction process succeeds with very high probability. Moreover, for small batch sizes, the reconstruction process is efficient. The only cost we have to pay is a loss of our ability to identify individual faulty signatures. When the batch-verification condition of Step 13 fails, we have to repeat the algorithm on sub-batches or resort to individual verifications.

3.2.6 Analysis of Algorithm S1

Time and Space Complexity

The total number of monomials handled during the equation-generation and equation-solving stages is $\mu = 2^{t-1} - 1 = \frac{m}{2} - 1$, where $m = 2^t$, which grows exponentially with t. Determination of the Eqns (3.10) and (3.11) needs $t-1$ symbolic additions involving rational functions with at most $\Theta(m)$ non-zero terms. Each symbolic point addition is followed by at most t uses of Eqn (3.4). Therefore, the symbolic derivation of R requires $\mathrm{O}(mt^2)$ operations in the field $\mathbb{F}_q$. Each equation $R_x = \alpha$ contains $\frac{\mu}{2}$ monomials containing y_i, $i = 1,2,\ldots,t$. To square this equation we need $\mathrm{O}(\mu \ln \mu \ln\ln \mu)$ field multiplications [28, 54]. The square contains $\leqslant \binom{\mu/2}{2} \approx \frac{\mu^2}{2^3}$ monomials containing y_i^2, $i = 1,2,\ldots,t$. Therefore, the removal of all the quadratic terms in the squared equation takes $\mathrm{O}(m^2t)$ field operations, that is, the generation of the $\mu \times \mu$ linearized system requires a total of $\sum_{i=1}^{\mu}(\mu \ln \mu \ln\ln \mu + \mu^2 t) = \mathrm{O}(m^3 t)$ field operations. Finally, Gaussian elimination on an $\mu \times \mu$ system demands $\Theta(m^3)$ field operations. Retrieving individual y_i values calls for $\mathrm{O}(mt^2)$ (usually $\mathrm{O}(mt)$) field operations. The running time of Algorithm S1 is dominated by the linear system-creation stage. Evidently, Algorithm S1 becomes impractical except only for small values of t.

It is worthwhile to investigate the running time of the naive Algorithm N. First, this algorithm needs to compute t modular square roots in the field $\mathbb{F}_q$. Each such

Algorithm 3.4 ECDSA Batch-Verification Algorithm S1

INPUT: Domain Parameters, messages $M_1, M_2, \ldots, M_t$, corresponding signatures $(r_1, s_1), (r_2, s_2), \ldots, (r_t, s_t)$ and public keys $Q_1, Q_2, \ldots, Q_t$ of the signers.

OUTPUT: Accept/Reject all the signatures.

1. Compute $w_i = s_i^{-1} \pmod{n}$ for all $i = 1, 2, \ldots, t$.
2. Compute $u_i = H(M_i)w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
3. Compute $v_i = r_i w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
4. Compute $R' = (\sum_{i=1}^{t} u_i)P + \sum_{i=1}^{t} v_i Q_i \in E(\mathbb{F}_q)$.

   Club together the points Q_i from same signers during the computation of R'. For example, if all the signatures belong to the same signer, compute R' as $(\sum_{i=1}^{t} u_i)P + (\sum_{i=1}^{t} v_i)Q$.
5. If R' is the point at infinity or if $y(R') = 0$, resort to individual verification, else proceed as follows.
6. Let $R_i = (r_i, y_i)$ with variables y_i for all $i = 1, 2, \ldots, t$.
7. Compute $R = (R_x, R_y) = \sum_{i=1}^{t} R_i$ symbolically using the substitutions $y_i^2 = r_i^3 + ar_i + b$ whenever necessary. The polynomials R_x and R_y are linear with respect to each y_i.
8. Generate equations $R_x = \alpha$ and $R_y = \beta$, where $R' = (\alpha, \beta)$.
9. Create $\mu - 1 = 2^{t-1} - 2$ other equations by repeated squaring of $R_x = \alpha$ or by repeatedly multiplying this equation by even-degree monomials in $y_1, y_2, \ldots, y_t$. Make the substitutions $y_i^2 = r_i^3 + ar_i + b$ whenever needed.
10. Solve the linearized system to find out the values of all even-degree monomials of $y_1, y_2, \ldots, y_t$. If the system is not uniquely solvable, resort to individual verification.
11. Multiply both sides of $R_y = \beta$ by y_1 and solve for y_1.
12. Compute $y_i = y_1 y_i / y_1$ from the monomials $y_1 y_i$ for all $i = 2, 3, \ldots, t$. (If $y_1 = 0$, solve for $y_2, y_3, \ldots, y_t$ as in Step 11.)

13. Accept all the signatures if and only if $y_i^2 = r_i^3 + ar_i + b \pmod q$ for all $i = 1, 2, \ldots, t$.

square-root computation (for example, by the Tonelli-Shanks algorithm [69]) involves an exponentiation in $\mathbb{F}_q$. Subsequently, one needs to check at most $m = 2^t = 2(\mu + 1)$ conditions, with each check involving the computation of the sum of t points on the curve. Therefore, the total running time of Algorithm N is $O((\sigma + m)t)$, where σ is the time for computing one square root in $\mathbb{F}_q$. Thus, Algorithm S1 outperforms Algorithm N only in situations where σ is rather large compared to m. This happens typically when the batch size t is small and the field size q is large.

The major memory requirement for Algorithm S1 is the storage of the coefficient matrix of the linearized system. This is a $\mu \times \mu$ matrix, that is, $\Theta(\mu^2) = \Theta(2^{2t})$ field elements need to be stored. The naive method, on the other hand, needs the storage of $\Theta(t)$ square roots (field elements) only.

A Bound on the Matrix M of the Linearized System

Let M denote the $\mu \times \mu$ coefficient matrix of the linearized system. In order to compute the number of roots $(r_1, r_2, \ldots, r_t)$ of $\det M = 0$, we treat $r_1, r_2, \ldots, r_t$ as symbols, and need to calculate an upper bound on the degree δ of each individual r_i. Without loss of generality, we compute an upper bound on the degree δ of r_1 in $\det M = 0$. To this effect, we first look at the expressions for R_x and R_y which are elements of $\mathbb{F}_q(r_1, r_2, \ldots, r_t)[y_1, y_2, \ldots, y_t]$. We can write $R_x = g_x/h$ and $R_y = g_y/h$, where g_x, g_y are polynomials in $\mathbb{F}_q[r_1, r_2, \ldots, r_t, y_1, y_2, \ldots, y_t]$, and the common denominator h is a polynomial in $\mathbb{F}_q[r_1, r_2, \ldots, r_t]$. Let η_t denote the maximum of the r_1-degrees in g_x, g_y and h. We first recursively derive an upper bound for η_t.

In Algorithm S1, we compute $R = R' + R''$ with $R' = (R'_x, R'_y) = \sum_{i=1}^{\tau} R_i$ and $R'' = (R''_x, R''_y) = \sum_{i=\tau+1}^{t} R_i$, where $\tau = \lceil t/2 \rceil$. The r_1-degree of R' is η_τ, whereas the r_1-degree of R'' is 0. The initial r_1-degree of $\lambda = (R''_y - R'_y)/(R''_x - R'_x)$ is at most η_τ. Clearing y_1 from the denominator of λ changes the r_1-degree to $2\eta_\tau + 3$. Subsequent eliminations of $y_2, \ldots, y_t$ finally reduces λ with a y-free denominator. The maximum r_1-degree of this expression for λ is $2^{t-1}(2\eta_\tau + 3)$. Therefore, λ^2 has r_1-

degree no more than $2^t(2\eta_\tau+3)$. Subsequent computations of $R_x=\lambda^2-R'_x-R''_x$ and $R_y=\lambda(R'_x-R_x)-R'_y$ indicate that

$$\eta_t \leqslant (2^t+2^{t-1})(2\eta_\tau+3)+2\eta_\tau \leqslant (2^t+2^{t-1})(2\eta_\tau+3)+2\eta_\tau$$

with $\tau=\lceil t/2\rceil$. Solving this recurrence gives the upper bound $\eta_t \leqslant 2^{2t+3\lceil\log_2 t\rceil+2}$.

Now, we follow a sequence of squaring and monomial multiplication to convert $R_x=\alpha$ to a set of linear equations. If Δ_i is the r_1-degree of the i-th equation, we have

$$\begin{aligned}\Delta_1 &= \eta_t,\\ \Delta_i &\leqslant 2\Delta_{i-1}+3 \text{ for } i\geqslant 2.\end{aligned}$$

The recurrence relation pertains to the case of squaring. One easily checks that $\Delta_i \leqslant (\eta_t+3)2^{i-1}$ for all $i\geqslant 1$. Finally, we consider $\det M=0$. The r_1-degree of this equation is

$$\delta \leqslant \Delta_1+\Delta_2+\cdots+\Delta_\mu \leqslant (\eta_t+3)(2^\mu-1) \leqslant \left(2^{2t+3\lceil\log_2 t\rceil+2}+3\right)\left(2^{2^{t-1}-1}-1\right).$$

Notice that this is potentially a very loose upper bound for δ. In general, we avoid squaring. Multiplication by a monomial can increase the r_1-degree by 3 if the monomial contains y_1. If the monomial does not contain y_1, the r_1-degree does not increase at all. Nevertheless, this loose upper bound is good enough in the present context.

Number of Roots of $\det M=0$

Let us write the equation $\det M=0$ as $D(r_1,r_2,\ldots,r_t)=0$, where the r_i-degree of the multivariate polynomial D is $\leqslant\delta$ for each i. We assume that D is not identically zero. We plan to show that the maximum number $B^{(t)}$ of roots of D is $\leqslant t\delta q^{t-1}$. To that effect, we first write D as a polynomial in r_t:

$$\begin{aligned}D(r_1,r_2,\ldots,r_t) &= D_\delta(r_1,r_2,\ldots,r_{t-1})r_t^\delta+D_{\delta-1}(r_1,r_2,\ldots,r_{t-1})r_t^{\delta-1}+\cdots+\\ &\quad D_1(r_1,r_2,\ldots,r_{t-1})r_t+D_0(r_1,r_2,\ldots,r_{t-1}).\end{aligned}$$

Since D is not identically zero, at least one D_i is not identically zero. If $(r_1,r_2,\ldots,r_{t-1})$ is a common root of each D_i, appending any value of r_t gives a root of D. The maximum number of common roots of $D_0,D_1,\ldots,D_\delta$ is $B^{(t-1)}$. On the other hand,

if $(r_1, r_2, \ldots, r_{t-1})$ is not a common root of all D_i, there are at most δ values of r_t satisfying $D(r_1, r_2, \ldots, r_t) = 0$. We, therefore, have

$$B^{(t)} \leqslant B^{(t-1)}q + (q^{t-1} - B^{(t-1)})\delta = (q-\delta)B^{(t-1)} + \delta q^{t-1}. \tag{3.15}$$

Moreover, we have

$$B^{(1)} \leqslant \delta. \tag{3.16}$$

By induction on t, one can show that $B^{(t)} \leqslant t\delta q^{t-1}$. This bound is rather tight, particularly for $\delta \ll q$ (as it happens in our cases of interest). A polynomial D satisfying equalities in (3.15) and (3.16) can be constructed as $D(r_1, r_2, \ldots, r_t) = \Delta(r_1)\Delta(r_2)\cdots\Delta(r_t)$, where Δ is a square-free univariate polynomial of degree δ, that splits over $\mathbb{F}_q$. By the principle of inclusion and exclusion (or by explicitly solving the recurrence (3.15)), we obtain the total number of roots of this D as

$$\begin{aligned} & \delta t q^{t-1} - \binom{t}{2}\delta^2 q^{t-1} + \binom{t}{3}\delta^3 q^{t-3} - \cdots + (-1)^{t-1}\delta^t \\ = \;& q^t - (q-\delta)^t \\ = \;& \delta(q^{t-1} + (q-\delta)q^{t-2} + (q-\delta)^2 q^{t-3} + \cdots + (q-\delta)^{t-1}). \end{aligned}$$

If $\delta \ll q$, this count is very close to $t\delta q^{t-1}$. It remains questionable whether our equation $\det M = 0$ actually encounters this worst-case situation, but this does not matter, at least in a probabilistic sense.

Unique Solvability of the Linearized System

In Step 10 of Algorithm 3.4, we solve a linearized $\mu \times \mu$ system in order to obtain the values of the even-degree monomials in the unknown y-coordinates $y_1, y_2, \ldots, y_t$. Let us call these monomials $z_1, z_2, \ldots, z_\mu$ and the coefficient matrix M. In order that the linearized system is uniquely solvable, we require $\det M \neq 0$. We now investigate how often this condition is satisfied, and also how we can force this condition to hold in most cases.

For a moment, let us treat the x-coordinates $r_1, r_2, \ldots, r_t$ as symbols. But then the failure condition $\det M = 0$ can be rephrased in terms of a multivariate polynomial equation in $r_1, r_2, \ldots, r_t$. Let us denote this equation as $D(r_1, r_2, \ldots, r_t) = 0$. If D is identically zero, then any values of $r_1, r_2, \ldots, r_t$ constitute a root of D. We explain shortly how this situation can be avoided.

Assume that D is not identically zero. Let δ be the maximum degree of each individual r_i in D. We have proved that

$$\delta \leqslant \left(2^{2t+3\lceil \log_2 t\rceil+2}+3\right)\left(2^{2^{t-1}-1}-1\right) \approx 2^{2^{t-1}+2t+3\lceil \log_2 t\rceil+1}.$$

If we restrict our attention to the values $t \leqslant 6$, we have $\delta \leqslant 2^{54}$. We know that the maximum number of roots of D is bounded below $t\delta q^{t-1}$. The total number of t-tuples $(r_1, r_2, \ldots, r_t)$ over $\mathbb{F}_q$ is q^t. Therefore, a randomly chosen tuple $(r_1, r_2, \ldots, r_t)$ is a root of D with probability $\leqslant t\delta q^{t-1}/q^t = t\delta/q$. If we use the inequalities $t \leqslant 6$, $\delta \leqslant 2^{54}$ and $q \geqslant 2^{160}$, we conclude that this probability is less than 2^{-103}. Therefore, if D is not the zero polynomial, we can solve for $z_1, z_2, \ldots, z_\mu$ uniquely with very high probability.

What remains is to propose a way to avoid the condition $D = 0$. We start with any t randomly chosen ECDSA signatures with r-values $r_1, r_2, \ldots, r_t$. We then choose any sequence of squaring and multiplication by z_i (Step 9 in Algorithm 3.4) in order to arrive at a linear system in $z_1, z_2, \ldots, z_\mu$. If the corresponding coefficient matrix M is not invertible, we discard the chosen sequence of squaring and multiplication. This is because $\det M = 0$ implies that either D is the zero polynomial or the chosen $r_1, r_2, \ldots, r_t$ constitute a root of a non-zero D. The second case is extremely unlikely. With high probability, we, therefore, conclude that the chosen sequence of squaring and multiplication gives $D = 0$ identically. We change the sequence, and repeat the above process until we come across the situation where $r_1, r_2, \ldots, r_t$ do not constitute a root of the non-zero polynomial equation $D(r_1, r_2, \ldots, r_t) = 0$. This implies that D is not identically zero, and randomly chosen $r_1, r_2, \ldots, r_t$ satisfy $D(r_1, r_2, \ldots, r_t) = 0$ with very low probability. We keep this sequence for all future invocations of our batch-verification algorithm, since this will work almost always.

Some sequences of squaring and multiplication, that work for all NIST prime curves are listed in Table 3.2. In the table, S stands for a squaring step, whereas a monomial (like y_2y_4) stands for multiplication by that monomial. In all these cases, we use only Eqn (3.10), whereas Eqn (3.11) is used only for the unique determination of individual y_i values. These sequences depend upon t alone, but not on the NIST curves. For curves other than the NIST curves, this method is expected to work equally well. Indeed, we may consider $D(r_1, r_2, \ldots, r_t)$ as a polynomial in $\mathbb{Z}[r_1, r_2, \ldots, r_t]$. If D is not identically zero, then it is identically zero modulo only a finite number of primes (the common prime divisors of the coefficients of D).

Table 3.2: Sequences to generate linearized systems for NIST prime curves

t	**Sequence in Step 9 of Algorithm 3.4**
2	No squaring or multiplication needed
3	y_1y_2, y_1y_3
4	$y_1y_2, y_1y_3, y_1y_4, y_2y_3, y_3y_4, y_1y_4$
5	$y_1y_2, y_1y_3, y_1y_4, y_1y_5, y_2y_3, y_2y_4, y_4y_5, y_1y_2, y_1y_3, y_1y_4, y_1y_5, y_1y_2, y_2y_4, y_2y_3$
6	$y_1y_2, y_1y_3, y_1y_4, y_1y_5, y_1y_6, y_2y_3, y_2y_4, y_2y_5, y_1y_2, y_3y_4, y_3y_5, y_1y_5, y_1y_6, y_1y_2y_3y_6, y_1y_5,$ $y_1y_4, y_1y_3, y_1y_2y_3y_6, y_1y_2, y_1y_3, y_1y_4, y_1y_5, y_2y_5, y_2y_3, S, y_2y_6, y_4y_6, y_3y_6, y_5y_6, y_1y_5$

Security Analysis

In Algorithm S1, we reconstruct the points R_i with x-coordinates $x(R_i) = r_i$ by forcing the condition $R = \sum_{i=1}^{t} R_i = \sum_{i=1}^{t} R'_i = R'$, where $R'_i = u_iP + v_iQ_i$. Suppose that an adversary too can force the condition $R = R'$. The adversary must also reveal the x-coordinates $r_1, r_2, \ldots, r_t$ as parts of ECDSA signatures. Given these x-coordinates and the condition $R = R'$, there exists (with high probability) a unique solution for the corresponding y-coordinates $y_1, y_2, \ldots, y_t$ of $R_1, R_2, \ldots, R_t$. This solution can be computed by the adversary, for example, using Algorithm S1 (or by taking modular square roots in $\mathbb{F}_q$ as in Algorithm N). So long as t is restricted to small constant values (like $t \leqslant 6$), the adversary requires only moderate computing resources for determining $y_1, y_2, \ldots, y_t$ uniquely. This implies that although the adversary needs to reveal only the x-coordinates r_i, (s)he essentially *knows* the full points R_i. But these points $R_1, R_2, \ldots, R_t$ satisfy the standard batch-verification condition for ECDSA*. That is, if the adversary can fool Algorithm S1, (s)he can fool the standard ECDSA* batch-verification algorithm too. It, therefore, follows that Algorithm S1 is no less secure than the standard batch-verification algorithm for ECDSA*. Conversely, if an adversary can fool any ECDSA* batch-verification algorithm, (s)he can always fool any ECDSA batch-verification algorithm, since ECDSA signatures are only parts of corresponding ECDSA* signatures. To sum up, Algorithm S1 is as secure as ECDSA* batch verification. For the security of the standard batch-verification algorithm for ECDSA* (or DSA), we refer the reader to [45].

The above argument is based upon the unique solvability of $y_1, y_2, \ldots, y_t$. If this is not the case, our algorithm resorts to individual verification. Moreover, we have argued that this unwelcome situation can be made extremely improbable.

An analysis of the security of Algorithm N is also worth including here. Suppose that an adversary can pass one of the $m = 2^t$ checks in Algorithm N along with disclosing $r_1, r_2, \ldots, r_t$. The correct choices y_i of the square roots of $r_i^3 + ar_i + b$ (that is, those choices corresponding to the successful check) constitute a case of fulfillment of the ECDSA* batch-verification criterion. Consequently, Algorithm N too is as secure as standard ECDSA* batch verification.

Cases of Failure for Algorithm S1

Our Monte Carlo batch-verification Algorithm S1 may fail for a few reasons. We now argue that these cases of failure are probabilistically very rare.

1. Taking $x_i = r_i$ blindly is a possible cause of failure for Algorithm S1. As discussed earlier, this situation has a very low probability. Furthermore, it is easy to identify when this situation occurs. In case of ambiguity in the values of x_i, we can repeat Algorithm S1 for all possible candidate tuples $(x_1, x_2, \ldots, x_t)$. If the points R_i are randomly chosen in $E(\mathbb{F}_q)$, most of these x_i values are unambiguously available to us, and there should not be many repeated runs (if any) of Algorithm S1. Repeated runs, if necessary, may be avoided, because doing so goes against the expected benefits achievable by batch verification.

2. Although we are able to identify good sequences of squaring and multiplication in Step 9 in order to force the determinant polynomial $D(r_1, r_2, \ldots, r_t)$ to be not identically zero, roots of this polynomial may appear in some cases of ECDSA signatures. We have seen that if $r_1, r_2, \ldots, r_t$ are randomly chosen, the probability of this situation is no more than 2^{-103}. However, an adversary may supply roots of D as $r_1, r_2, \ldots, r_t$, thereby forcing our algorithm to resort to individual verification. Clearly, the adversary gains nothing in this case, but our algorithm forfeits the desired speedup.

3. The derivation of Eqn (3.8) is based upon the point-addition formula on the curve E. The doubling formula (corresponding to the case $P_1 = P_2$) has a different value for λ. Point additions of the form $U + (-U)$ cannot also be handled by the addition formula. So long as we work symbolically using the unknown quantities $y_1, y_2, \ldots, y_t$, it is impossible to predict when the two

points being added turn out to be equal or opposite. If $R_1, R_2, \ldots, R_t$ are randomly chosen from $E(\mathbb{F}_q)$, the probability of this occurrence is extremely low. However, an attacker may force the case of such degenerate sums as suggested by Bernstein et al. [7]. Failure of batch verification on a collection of signatures anyway calls for a treatment of the signatures in smaller groups and/or individually. In doing so, one detects the error associated with Algorithm S1. Another alternative is to randomize the batch-verification process.

4. Algorithm S1 fails if R' is the point at infinity or lies on the x-axis ($\beta = 0$). In that case, one should resort to individual verification. For randomly chosen session keys, this case occurs with a very small probability (nearly $4/q$).

3.3 Algorithm S2

The linearization stage in Algorithm S1 (requiring $\mathrm{O}(m^3 t)$ field operations) and the subsequent Gaussian-elimination stage (requiring $\mathrm{O}(m^3)$ field operations) are rather costly, m being already an exponential function of the batch size t. Our second symbolic-manipulation algorithm S2 avoids these two stages altogether.

Algorithm S1 uniquely solves for the monomials $z_1, z_2, \ldots, z_\mu$ using the equation $R_x = \alpha$ only. At this point, there are only two possible solutions for the y_i values: $(y_1, y_2, \ldots, y_t)$ and $(-y_1, -y_2, \ldots, -y_t)$. This *sign* ambiguity is eliminated by using the other equation $R_y = \beta$. As mentioned in connection with the security analysis of Algorithm N, the exact determination of these signs is not important. In other words, we would be happy even if we can determine each y_i correctly up to multiplication by ± 1. This, in turn, implies that if we have any multivariate equation (linear in y_i) of the form $uy_i + v = 0$ (where u, v are polynomials in $y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_t$), we do not mind multiplying this equation by $uy_i - v$ so that $\pm y_i$ satisfy $u^2 y_i^2 - v^2 = 0$. But $y_i^2 = r_i^3 + ar_i + b$, so we have $u_i^2(r_i^3 + ar_i + b) - v_i^2 = 0$, an equation in which y_i is *eliminated*. This observation leads to Algorithm S2.

Like Algorithm S1, we first symbolically compute $R = \sum_{i=1}^{t} R_i$, and arrive at Eqns (3.10) and (3.11). Then, we consider only the multivariate equation $R_x - \alpha = 0$ linear individually in each y_i. We first eliminate y_1, and with substitutions given by

Eqn (3.4) for $i = 2,3,\ldots,t$, we arrive at a multivariate equation in $y_2, y_3, \ldots, y_t$, again linear in each of these variables. We eliminate y_2 from this equation, and arrive at a multivariate equation in $y_3, y_4, \ldots, y_t$. We repeat this process until all variables $y_1, y_2, \ldots, y_t$ are eliminated. If the polynomial after all these eliminations reduces to zero, we know that the original equation $R_x = \alpha$ is consistent with respect to $y_i^2 = r_i^3 + ar_i + b$ for all $i = 1,2,\ldots,t$.

We may likewise eliminate $y_1, y_2, \ldots, y_t$ from the equation $R_y - \beta = 0$ too, but this is not necessary, because it suffices to know y_i uniquely up to multiplication by ± 1.

Algorithm 3.5 summarizes the steps of this improved Algorithm S2.

3.3.1 Implementation Issues

Some comments on efficient implementations of the elimination stage (Step 10) are now in order. First, we are not using Eqn (3.11) at all in Algorithm S2. Consequently, it is not necessary to compute the polynomial R_y. However, in the stage of symbolic-computation (Step 8), we need to compute all intermediate y-coordinates, since they are needed in the final value of R_x. The computation of only the last y-coordinate R_y may be avoided. Still, this saves quite some amount of effort ($\mathrm{O}(mt)$ field operations, to be precise). This saving does not affect the theoretical complexity of Step 8 in the big-Oh notation, but its practical effects are noticeable.

The second issue is that the polynomials u and v computed in Step 10 have some nice properties. Throughout this step, ϕ and v are polynomials with each non-zero term having even degree, whereas u is a polynomial with each non-zero term having odd degree. In particular, when the first $t-2$ y-coordinates are eliminated, we have $\phi = uy_{t-1}y_t + v$ with $u, v \in \mathbb{F}_q$. Elimination of y_{t-1} eliminates y_t too, so an explicit elimination of y_t is not necessary, that is, it suffices to let the loop of Step 10 run for $i = 1,2,\ldots,t-1$ only.

Algorithm 3.5 ECDSA Batch-verification Algorithm S2

INPUT: Domain Parameters, messages $M_1, M_2, \ldots, M_t$, corresponding signatures $(r_1, s_1), (r_2, s_2), \ldots, (r_t, s_t)$ and public keys $Q_1, Q_2, \ldots, Q_t$ of the signers.

OUTPUT: Accept/Reject all the signatures.

1. Compute $w_i = s_i^{-1} \pmod{n}$ for all $i = 1, 2, \ldots, t$.
2. Compute $u_i = H(M_i)w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
3. Compute $v_i = r_i w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
4. Compute $R' = (\sum_{i=1}^{t} u_i)P + \sum_{i=1}^{t} v_i Q_i \in E(\mathbb{F}_q)$.

   Club together the points Q_i from same signers during the computation of R'. For example, if all the signatures belong to the same signer, compute R' as $(\sum_{i=1}^{t} u_i)P + (\sum_{i=1}^{t} v_i)Q$.
5. If R' is the point at infinity or if $y(R') = 0$, resort to individual verification, else proceed as follows.
6. For each $i = 1, 2, \ldots, t$, if $r_i^3 + ar_i + b$ is neither zero nor a quadratic residue modulo q, reject the i-th signature, and remove it from the batch.
7. Let $R_i = (r_i, y_i)$ with variables y_i for all $i = 1, 2, \ldots, t$.
8. Compute $R = (R_x, R_y) = \sum_{i=1}^{t} R_i$ symbolically using the substitutions $y_i^2 = r_i^3 + ar_i + b$ whenever necessary.
9. Let $\phi = R_x - \alpha$.
10. For $i = 1, 2, 3, \ldots, t-1$, eliminate y_i from ϕ as in the following steps:

    If ϕ is identically zero, resort to individual verification.
    Otherwise, write $\phi = uy_i + v$ where u, v are polynomials in $y_{i+1}, \ldots, y_t$.
    If u is the zero polynomial, resort to individual verification.
    Set $\phi = u^2(r_i^3 + ar_i + b) - v^2$.
    Substitute y_j^2 in ϕ by $r_j^3 + ar_j + b$ for $j = i+1, \ldots, t$.
11. Accept all the signatures if and only if $\phi = 0$.

3.3.2 Reconstruction of $y_1, y_2, \ldots, y_t$

This step is unnecessary (and is not included) in Algorithm S2, but is explained here for theoretical interest only. Suppose that a set of t ECDSA signatures with the x-coordinates $r_1, r_2, \ldots, r_t$ passes Algorithm S2. We remember all ϕ values of the form $\phi = uy_i + v$ in Step 10. We then work back to obtain two solutions for the y-coordinates $y_1, y_2, \ldots, y_t$. Just before y_{t-1} is eliminated, we have $\phi = uy_{t-1}y_t + v$ with $u, v \in \mathbb{F}_q$. Since ϕ will eventually reduce to zero, some square roots of $r_{t-1}^3 + ar_{t-1} + b$ and $r_t^3 + ar_t + b$ satisfy $\phi = 0$. If y_{t-1}, y_t are two such square roots (computed by the Tonelli-Shanks algorithm [69]), the other solution of $\phi = 0$ at this point is $-y_{t-1}, -y_t$.

Now, let us assume that we have computed two solutions $y_{i+1}, y_{i+2}, \ldots, y_t$ and $-y_{i+1}, -y_{i+2}, \ldots, -y_t$. We now look at the elimination step for y_i, that is, $\phi = uy_i + v$ with $u, v \in \mathbb{F}_q[y_{i+1}, y_{i+2}, \ldots, y_t]$. Since $\phi = 0$ gives $y_i = -v/u$ with v a polynomial with non-zero terms of even degrees and with u a polynomial with non-zero terms of odd degrees, the substitution of the two sets of values of $y_{i+1}, \ldots, y_t$ gives exactly two values $\pm y_i$.

Here, we have assumed that u evaluates to a non-zero value for the computed values of $y_{i+1}, \ldots, y_t$. If, however, we have $u(y_{i+1}, \ldots, y_t) = 0$, we must also have $v(y_{i+1}, \ldots, y_t) = 0$ too, since otherwise we cannot reduce ϕ to zero after the remaining eliminations of $y_{i+1}, \ldots, y_t$. In this case, plugging in any value of y_i^2 allows the algorithm to succeed, that is, the information $y_i^2 = r_i^3 + ar_i + b$ is not utilized during the elimination of y_i. Therefore, if this situation occurs, we resort to individual verification. Since u and v are polynomials of degrees much smaller than q, this situation occurs with vanishingly small probabilities.

Eventually, we compute two solutions $y_1, y_2, \ldots, y_t$ and $-y_1, -y_2, \ldots, -y_t$ of $R_x = \alpha$. The sign ambiguity is eliminated by using the other batch-verification condition $R_y = \beta$, provided that $\beta \neq 0$. The case $\beta = 0$ is handled by Step 5 of Algorithm S2.

The running time of this reconstruction procedure is $\mathrm{O}(mt^2 + \sigma)$. Here, σ is the time for computing a square root in $\mathbb{F}_q$. It follows that this reconstruction procedure is practical in all cases where running the algorithms N and S1 is practical.

The condition $\det M = 0$ (Step 10 of Algorithm 3.4) was necessary for Algorithm S1

to work. Such a condition is not needed for Algorithms N and S2. Moreover, Steps 11 and 12 of Algorithm S1 (determination of individual y_i values) are cryptographically unimportant, since $R_x = \alpha$ already identifies exactly two solutions for the reconstructed points. If these steps are omitted, the batch-acceptance criterion of Step 13 has to be changed. We compute z_i^2 for all $i = 1, 2, \ldots, \mu$, and match these values against appropriate products of $r_j^3 + ar_j + b$. In fact, it suffices to consider only the monomials z_i of degree 2. Note, however, that the unique determination of all y_i values takes only an insignificant amount of time compared to Steps 7–9 in Algorithm S1. Therefore, it does not practically matter to make a choice between whether we carry out these steps or not.

3.3.3 Analysis of Algorithm S2

Time and Space Complexity

The symbolic computation of (R_x, R_y) involves $\mathrm{O}(mt^2)$ field operations (similar to Algorithm S1). Subsequently, we start with the polynomial $\phi = R_x - \alpha$ with at most $\mu = \frac{m}{2}$ non-zero terms. Elimination of y_i requires computing the squares u^2 and v^2, carrying out the polynomial arithmetic $u^2(r_i^3 + ar_i + b) - v^2$, and $t - i$ substitutions of y_j^2 by $r_j^3 + ar_j + b$. During the elimination of y_i, both u and v contain $\frac{\mu}{2^i}$ monomials. The computation of each of u^2 and v^2 by fast polynomial multiplication requires $\frac{\mu}{2^i} \ln \frac{\mu}{2^i} \ln\ln \frac{\mu}{2^i}$ field operations [28, 54]. Moreover, u and v conatin $\frac{\mu}{2^{i+1}}$ monomials containing y_j, where $j > i$. Therefore, each of u^2 and v^2 conatins $\binom{\frac{\mu}{2^{i+1}}}{2} \approx \frac{\mu^2}{2^{2i+3}}$ monomials containing y_j^2, and thus the substitution of y_j^2 (for each $j > i$) requires $\frac{\mu^2}{2^{2i+3}}$ field operations. Consequently, the total cost of the elimination phase is

$$
\begin{aligned}
& 2\sum_{i=1}^{t-2}\left[\frac{\mu}{2^i}\ln\frac{\mu}{2^i}\ln\ln\frac{\mu}{2^i} + (t-i)\frac{\mu^2}{2^{2i+3}}\right] \\
= \; & 2\sum_{i=1}^{t-2}\left[\frac{2^{t-1}}{2^i}\ln\frac{2^{t-1}}{2^i}\ln\ln\frac{2^{t-1}}{2^i} + (t-i)\frac{\left(2^{t-1}\right)^2}{2^{2i+3}}\right] \\
= \; & 2\sum_{i=1}^{t-2}\left[2^{t-i-1}\ln 2^{t-i-1}\ln\ln 2^{t-i-1} + (t-i)2^{2t-2i-5}\right] \\
< \; & \sum_{i=1}^{t-2}\left[(t-i)2^{t-i}\ln(t-i) + (t-i)2^{2t-2i}\right] = \sum_{i=2}^{t-1}\left[i2^i\ln(i) + i2^{2i}\right]
\end{aligned}
$$

The inequality $2(i2^i \ln(i) + i2^{2i}) < ((i+1)2^{(i+1)} \ln((i+1)) + i2^{2(i+1)})$ holds for $i \geqslant 1$. Therefore, the reduction of ϕ requires $2(t2^t \ln(t) + t2^{2t}) = \mathrm{O}(mt \ln t + m^2 t) = \mathrm{O}(m^2 t)$ field operations, where $m = 2^t$. This is significantly better than the $\mathrm{O}(m^3 t)$ operations needed by Algorithm S1. Moreover, Algorithm S2 outperforms Algorithm N for a wide range of t and q, since the condition $(\sigma + m)t \gg m^2 t$ is more often satisfied than the condition $(\sigma + m)t \gg m^3 t$.

For Algorithm S2, the major storage requirement is that for the polynomial ϕ. This multivariate polynomial has $m/2$ non-zero terms, so the space complexity is that of $\Theta(m) = \Theta(2^t)$ field elements.

Security Analysis

We establish the equivalence between the security of Algorithm S2 and the security of standard ECDSA* batch verification, as we have done for the earlier algorithms (N and S1). Suppose that an adversary reveals the x-coordinates $r_1, r_2, \ldots, r_t$ in ECDSA signatures which pass the batch-verification procedure of Algorithm S2. The procedure described in Section 3.3.2 indicates that there are exactly two solutions $(y_1, y_2, \ldots, y_t)$ and $(-y_1, -y_2, \ldots, -y_t)$ consistent with $\phi = 0$ (Step 9 of Algorithm S2) and $y_i^2 = r_i^3 + ar_i + b$ for $i = 1, 2, \ldots, t$. One of these solutions corresponds to the ECDSA* signatures based upon the disclosed values $r_1, r_2, \ldots, r_t$. It is that solution that would pass $R_y = \beta$. To sum up, the adversary can forge the standard ECDSA* batch-verification algorithm. Moreover, this forging procedure which essentially involves the unique reconstruction of the points $R_i = (r_i, y_i)$ is practical for any adversary with only a moderate amount of computing resources, so long as t is restricted to small values (the only cases where we can apply S2).

Derivation of the Failure Probabilities p_i during Elimination

Like Algorithm S1, we first symbolically compute $R = \sum_{i=1}^{t} R_i$, and arrive at Eqns (3.10) and (3.11). Then, we set $\phi = R_x - \alpha$ in step 9 of Algorithm S2. If ϕ is identically zero, then for any values of $y_1, y_2, \ldots, y_t$, batch verification succeeds without using the Eqns (3.4) in the elimination phase at all. This situation occurs if all the coefficients of all the monomials (and also the constant term) in ϕ are zero. We name the monomials

(of even total degrees, including that of degree zero) as $z_1, z_2, \ldots, z_\mu$, where $\mu = 2^{t-1}$. We write this situation as

$$\rho_1 z_1 + \rho_2 z_2 + \cdots + \rho_\mu z_\mu = 0. \tag{3.17}$$

with each $\rho_i = 0$. For the moment, we treat the x-coordinates $r_1, r_2, \ldots, r_t$ as symbols. Each ρ_i in Eqn (3.17) is a polynomial in $\mathbb{F}_q[r_1, r_2, \ldots, r_t]$. Let δ' be the maximum degree of each individual r_j in each ρ_i. We have already derived that δ' is bounded from above by $\Delta_1 = \eta_t \leqslant 2^{2t+3\lceil \log_2 t \rceil + 2}$. If we restrict our attention to the values $t \leqslant 8$, we see that $\delta' \leqslant 2^{27}$. Let the tuple $(r_1, r_2, \ldots, r_t)$ be a root of ρ_i. We estimate that there are $\leqslant t\delta' q^{t-1}$ such tuples. The total number of t-tuples over $\mathbb{F}_q$ is q^t. Therefore, a randomly chosen tuple $(r_1, r_2, \ldots, r_t)$ is a root of ρ_i with probability $\leqslant t\delta' q^{t-1}/q^t = t\delta'/q$. Now, the total number of ρ_i's is μ. Therefore, the probability that a randomly chosen tuple $(r_1, r_2, \ldots, r_t)$ over $\mathbb{F}_q$ is a root of all the ρ_i's is $p_1 \leqslant (t\delta'/q)^\mu$. For $t \leqslant 8$, $\delta' \leqslant 2^{27}$ and $q \geqslant 2^{160}$, we have $p_1 \leqslant 2^{-16510}$.

Even if ϕ is not identically zero at the beginning of the elimination phase, it should never become so before all of $y_1, y_2, \ldots, y_t$ are eliminated. Let p_i denote the probability that ϕ becomes identically zero before the elimination of y_i. We have calculated p_1 above. Here, we calculate p_i for $i = 2, 3, \ldots, t-1$. Let δ'_i be the total degree in all r_j's in ϕ just before the elimination of y_i. We have $\delta'_i = 2\delta'_{i-1} + 3 \approx 2\delta'_i = 2^{t-i}\delta'$. Moreover, at this point, the number of even-degree monomials in $y_i, y_{i+1}, \ldots, y_t$ in ϕ is $2^{t-i} = \mu/2^{i-1}$. Therefore, like the expression for p_1, we derive that

$$p_i \leqslant (t\delta'_i/q)^{2^{t-i}} = (t\delta'_i/q)^{\frac{\mu}{2^{i-1}}}.$$

The probability that ϕ becomes identically zero just before the elimination of y_i, but never earlier, is $(1-p_1)(1-p_2)\cdots(1-p_{i-1})p_i$. Therefore, the probability that ϕ becomes identically zero in any one of the $t-1$ elimination rounds is

$$\pi \leqslant \sum_{i=1}^{t-1} \left[p_i \prod_{j=1}^{i-1} (1-p_j) \right].$$

For practical ranges of parameter values, all p_i are very close to zero, so we can approximate $1-p_i$ by 1, and conclude that

$$\pi \approx \sum_{i=1}^{t-1} p_i.$$

Moreover, p_{t-1} is the most dominating term in the above summation, so we have

$$\pi \approx p_{t-1} \leqslant (t\delta'_{t-1}/q)^2 \approx (t2^{t-2}\delta'/q)^2 \leqslant 2^{6t+8\lceil \log_2 t \rceil + 2}/q^2.$$

Cases of Failure for Algorithm S2

Algorithm S2 may fail for a variety of reasons. Most of these reasons are identical to those associated with Algorithm S1 (see points 1, 3 and 4 in Section 3.2.6). However, unlike Algorithm S1, Algorithm S2 does not generate or solve the linearized system. It instead uses a separate elimination idea. Here, we require ϕ to be never identically equal to zero before all variables $y_1, y_2, \ldots, y_t$ are eliminated. This is motivated by that we require all of the t y-coordinates to play active roles in the elimination phase. If p_i denotes the probability that ϕ becomes identically zero during the elimination of y_i, then the probability of failure of Algorithm S2 inside the loop of Step 10 is

$$\begin{aligned} & p_1 + (1-p_1)p_2 + (1-p_1)(1-p_2)p_3 + \cdots + (1-p_1)(1-p_2)\cdots(1-p_{t-2})p_{t-1} \\ \approx\ & p_1 + p_2 + \cdots + p_{t-1} \\ \approx\ & p_{t-1} \end{aligned}$$

(recall that y_{t-1} and y_t are eliminated together). For $q \geqslant 2^{160}$ and $t \leqslant 8$, this failure probability is $\leqslant 2^{-246}$.

3.4 Efficient Variants of S1 and S2

In Algorithm S1, we generate a system of linearized equations in $\frac{m}{2} - 1 = 2^{t-1} - 1$ monomials (Steps 7–9). The resulting equation is solved in Step 10 which turns out to a costly step of Algorithm S1, demanding $\Theta(m^3)$ field operations.

In Algorithm S2, on the other hand, Step 8 turns out to be a time-consuming step. This step calls for $\Theta(mt^2)$ field operations. Step 10 calls for $\Theta(m^2t)$ field operations. Any improvement in these steps speeds up Algorithm S2.

In this section, we explain a strategy to reduce the total number of monomials in Algorithms S1 and S2. So far, we have been symbolically computing the point $R = \sum_{i=1}^{t} R_i$, and equating the symbolic sum to $R' = (\alpha, \beta)$. This results in polynomial expressions with $\Theta(2^{t-1})$ (that is, $\Theta(m)$) non-zero terms.

Now, let $\tau = \lceil t/2 \rceil$. We symbolically compute the two sums:

$$R^{(1)} = \sum_{i=1}^{\tau} R_i \ \text{ and } \ R^{(2)} = R' - \sum_{i=\tau+1}^{t} R_i. \tag{3.18}$$

The polynomial expressions involved in $R^{(1)}$ and $R^{(2)}$ contain only $\Theta(2^\tau)$, that is, $\Theta(\sqrt{m})$ non-zero terms. Computing these two symbolic sums, therefore, requires $\Theta(2^\tau\tau^2)$, that is, $\Theta(\sqrt{m}t^2)$ field operations which is significantly smaller than the $\Theta(mt^2)$ field operations associated with the symbolic computation of the complete sum $\sum_{i=1}^{t} R_i$. The condition $R = R'$ is equivalent to the condition $R^{(1)} = R^{(2)}$. Using this new condition helps us in speeding up the subsequent steps too.

The symbolic sum $R^{(1)}$ can be computed using Algorithm 3.3. For computing $R^{(2)}$, one may first compute the symbolic sum $\sum_{i=\tau+1}^{t} R_i$ by Algorithm 3.3, and subsequently add the opposite of this point to the explicit point $R = (\alpha, \beta) \in E(\mathbb{F}_q)$. This symbolic addition involves denominator clearing and substitutions (3.4) for $i = \tau+1, \ldots, t$. A faster approach is to compute $R^{(2)}$ as $R' + \sum_{i=\tau+1}^{t}(-R_i) = R' + \sum_{i=\tau+1}^{t}(r_i, -y_i)$. This situation is similar to the case of Algorithm 3.3 with two modifications. For the first summand, the y-coordinate is explicitly available as an element of $\mathbb{F}_q$. Consequently, this y-coordinate is not considered in denominator-clearing procedure of Algorithm 3.2. For the other summands, the elliptic-curve point being added is $(r_i, -y_i)$ instead of (r_i, y_i). Therefore, Step 1 of Algorithm 3.3 should return $(r_j, -y_j)$.

3.4.1 Algorithm S1′

Step 7 of Algorithm S1 can be replaced by the two symbolic additions given by Eqn (3.18). In that case, we replace Step 8 by the initial generation of two equations $x(R^{(1)}) = x(R^{(2)})$ and $y(R^{(1)}) = y(R^{(2)})$. It is easy to argue that $x(R^{(1)})$ is a polynomial in $y_1, y_2, \ldots, y_\tau$ with each non-zero term having even degree, whereas $y(R^{(1)})$ is a polynomial in $y_1, y_2, \ldots, y_\tau$ with each non-zero term having odd degree. That is, the number of non-zero terms in these two expressions is $2^{\tau-1} = \frac{\sqrt{m}}{2}$. However, the presence of $R' = (\alpha, \beta)$ on the right side of the expression for $R^{(2)}$ (Eqn 3.18) lets both $x(R^{(2)})$ and $y(R^{(2)})$ contain all (square-free) monomials in $y_{\tau+1}, y_{\tau+2}, \ldots, y_t$ (both even and odd degrees). There are exactly $2^{\lfloor t/2 \rfloor} - 1 \leqslant \sqrt{m} - 1$ monomials in these two expressions. In the linearized system that we subsequently generate, we consider, as variables, only the even-degree monomials in $y_1, y_2, \ldots, y_\tau$ and all monomials in $y_{\tau+1}, y_{\tau+2}, \ldots, y_t$. To start with, we have one equation $x(R^{(1)}) = x(R^{(2)})$ in these $\Theta(\sqrt{m})$ monomials.

Subsequently, we keep on squaring the equation $x(R^{(1)}) = x(R^{(2)})$ (and substituting values of y_i^2 wherever necessary). This sequence does not increase the number of monomials in the linearized equations. More precisely, for any $j \geqslant 0$, the equation $x(R^{(1)})^{2^j} = x(R^{(2)})^{2^j}$ contains only the $\Theta(\sqrt{m})$ monomials with which we start. If we fail to obtain a linearized system of full rank, we start squaring the other initial equation $y(R^{(1)}) = y(R^{(2)})$. For any $j \geqslant 1$, the equation $y(R^{(1)})^{2^j} = y(R^{(2)})^{2^j}$ again contains only the monomials with which we start. In all the cases studied, we have been able to obtain a full-rank linearized system by squaring the two initial equations. Since the number of linearized variables in $\Theta(\sqrt{m})$, the total cost of Step 9 of Algorithm S1 now reduces to $O(m^{3/2}t)$ field operations. Finally, in Step 10, we solve a system with $\Theta(\sqrt{m})$ variables. This step calls for $\Theta(m^{3/2})$ field operations. Algorithm 3.6 summarize the Algorithm S1′.

To sum up, using the trick introduced in this section decreases the number of field operations from $\Theta(m^3t)$ to $\Theta(m^{3/2}t)$. Let us plan to call this efficient variant of S1 as S1′. Fundamentally, S1′ is not a different algorithm from S1. In particular, the security of S1′ is the same as the security of S1 (in fact, little better, because fewer linearized equations are involved). However, the reduction in the running time is very significant, both theoretically and practically. The space complexity too improves from $\Theta(m^2)$ to $\Theta(m)$.

3.4.2 Algorithm S2′

Instead of starting with $\phi = R_x - \alpha$ (Step 9 of Algorithm S2), we start with the initial expression

$$\phi = x(R^{(1)}) - x(R^{(2)}). \tag{3.19}$$

We then repeatedly eliminate $y_1, y_2, \ldots, y_t$ as in Step 10. Although the initial expression of ϕ contains much less number of monomials than in the original Algorithm S2, elimination of y_1 itself introduces many new monomials in ϕ, that is, soon ϕ becomes almost *full*. Consequently, Step 10 continues to make $\Theta(m^2t)$ field operations as before, that is, the theoretical running time of S2′ is the same as that of S2. The space complexity also remains the same as S2, namely, $\Theta(m)$ field elements. Still, the effects of our heuristic are clearly noticeable in practical implementations.

As described in Section 3.3.1, the y-coordinates $y(R^{(1)})$ and $y(R^{(2)})$ need not be

Algorithm 3.6 ECDSA Batch-Verification Algorithm S1′

INPUT: Domain Parameters, messages $M_1, M_2, \ldots, M_t$, corresponding signatures $(r_1, s_1), (r_2, s_2), \ldots, (r_t, s_t)$ and public keys $Q_1, Q_2, \ldots, Q_t$ of the signers.

OUTPUT: Accept/Reject all the signatures.

1. Compute $w_i = s_i^{-1} \pmod n$ for all $i = 1, 2, \ldots, t$.
2. Compute $u_i = H(M_i)w_i \pmod n$ for all $i = 1, 2, \ldots, t$.
3. Compute $v_i = r_i w_i \pmod n$ for all $i = 1, 2, \ldots, t$.
4. Compute $R' = (\sum_{i=1}^{t} u_i)P + \sum_{i=1}^{t} v_i Q_i \in E(\mathbb{F}_q)$.

   Club together the points Q_i from same signers during the computation of R'. For example, if all the signatures belong to the same signer, compute R' as $(\sum_{i=1}^{t} u_i)P + (\sum_{i=1}^{t} v_i)Q$.
5. If R' is the point at infinity or if $y(R') = 0$, resort to individual verification, else proceed as follows.
6. Let $R_i = (r_i, y_i)$ with variables y_i for all $i = 1, 2, \ldots, t$.
7. let $\tau = \lceil t/2 \rceil$. Compute two sums: $R^{(1)} = \sum_{i=1}^{\tau} R_i$ and $R^{(2)} = R' - \sum_{i=\tau+1}^{t} R_i$, where $R' = (\alpha, \beta)$, symbolically using the substitutions $y_i^2 = r_i^3 + ar_i + b$ whenever necessary. The polynomials $R_x^{(1)}$ and $R_y^{(1)}$, $R_x^{(2)}$ and $R_y^{(2)}$ are linear with respect to each y_i.
8. Generate equations $R_x^{(1)} = R_x^{(2)}$ and $R_y^{(1)} = R_y^{(2)}$.
9. Create the requisite number of other equations by the repeated squaring of $R_x^{(1)} = R_x^{(2)}$. Make the substitutions $y_i^2 = r_i^3 + ar_i + b$ whenever needed.
10. Solve the linearized system to find out the values of all even-degree square-free monomials in $y_1, y_2, \ldots, y_\tau$ and of all square-free monomials in $y_{\tau+1}, y_{\tau+2}, \ldots, y_t$. If the system is not uniquely solvable, resort to individual verification.
11. Multiply both sides of $R_y^{(1)} = R_y^{(2)}$ by y_1 and solve for y_1.
12. Compute $y_i = y_1 y_i / y_1$ from the monomials $y_1 y_i$ for all $i = 2, 3, \ldots, \tau$. The other y-coordinates $y_{\tau+1}, y_{\tau+2}, \ldots, y_t$ are available from Step 10.

13. Accept all the signatures if and only if $y_i^2 = r_i^3 + ar_i + b \pmod{q}$ for all $i = 1, 2, \ldots, t$.

computed. It is, however, necessary to symbolically compute the y-coordinates of all intermediate sums.

This variant of Algorithm S2, in which Step 8 is replaced by the computations given by Eqn (3.18), and Step 9 is replaced by the initialization given by Eqn (3.19), is denoted as S2′. Notice that S2′ is not fundamentally different from S2. For example, the security of S2′ is identical to that of S2. Algorithm 3.7 summarize all the steps.

3.4.3 Cases of Failure for Algorithms S1′ and S2′

From the viewpoint of failure analysis, the faster variants S1′ and S2′ are identical to Algorithms S1 and S2, respectively. The coefficients in the monomials can be treated as polynomials in $r_1, r_2, \ldots, r_t$. For S1′ and S2′, these polynomials have smaller total degrees in r_j's than S1 and S2. Consequently, the failure probabilities arising out of $\det M$ being zero (for S1 and S1′) or of ϕ becoming identically zero before all elimination rounds are smaller for S1′ and S2′ than for S1 and S2. A more precise upper bound on these failure probabilities can be computed for S1′ and S2′. However, since S1 and S2 already enjoy negligibly small failure probabilities, an exact determination of these probabilities for S1′ and S2′ would be of theoretical interests only, and is omitted here.

3.5 Experimental Results

All experiments are carried out in a 2.33 GHz Xeon server running Ubuntu Linux Server Version 2012 LTS. The algorithms are implemented using the GP/PARI calculator [15] (version 2.5.0 compiled by the GNU C compiler 4.6.2). We have used the symbolic-computation facilities of the calculator in our programs. All other functions (like scalar multiplication and square-root computation) are written as subroutines in which function-call overheads are minimized as much as possible.

Algorithm 3.7 ECDSA Batch-verification Algorithm S2′

INPUT: Domain Parameters, messages $M_1, M_2, \ldots, M_t$, corresponding signatures $(r_1, s_1), (r_2, s_2), \ldots, (r_t, s_t)$ and public keys $Q_1, Q_2, \ldots, Q_t$ of the signers.

OUTPUT: Accept/Reject all the signatures.

1. Compute $w_i = s_i^{-1} \pmod n$ for all $i = 1, 2, \ldots, t$.
2. Compute $u_i = H(M_i)w_i \pmod n$ for all $i = 1, 2, \ldots, t$.
3. Compute $v_i = r_i w_i \pmod n$ for all $i = 1, 2, \ldots, t$.
4. Compute $R' = (\sum_{i=1}^{t} u_i)P + \sum_{i=1}^{t} v_i Q_i \in E(\mathbb{F}_q)$.

   Club together the points Q_i from same signers during the computation of R'. For example, if all the signatures belong to the same signer, compute R' as $(\sum_{i=1}^{t} u_i)P + (\sum_{i=1}^{t} v_i)Q$.
5. If R' is the point at infinity or if $y(R') = 0$, resort to individual verification, else proceed as follows.
6. For each $i = 1, 2, \ldots, t$, if $r_i^3 + ar_i + b$ is neither zero nor a quadratic residue modulo q, reject the i-th signature, and remove it from the batch.
7. Let $R_i = (r_i, y_i)$ with variables y_i for all $i = 1, 2, \ldots, t$.
8. let $\tau = \lceil t/2 \rceil$. Compute two sums: $R^{(1)} = \sum_{i=1}^{\tau} R_i$ and $R^{(2)} = R' - \sum_{i=\tau+1}^{t} R_i$, where $R' = (\alpha, \beta)$, symbolically using the substitutions $y_i^2 = r_i^3 + ar_i + b$ whenever necessary. The polynomials $R_x^{(1)}$ and $R_y^{(1)}$, $R_x^{(2)}$ and $R_y^{(2)}$ are linear with respect to each y_i.
9. Let $\phi = R_x^{(1)} - R_x^{(2)}$.
10. For $i = 1, 2, 3, \ldots, t-1$, eliminate y_i from ϕ as in the following steps:

    If ϕ is identically zero, resort to individual verification.
    Otherwise, write $\phi = uy_i + v$ where u, v are polynomials in $y_{i+1}, \ldots, y_t$.
    If u is the zero polynomial, resort to individual verification.
    Set $\phi = u^2(r_i^3 + ar_i + b) - v^2$.
    Substitute y_j^2 in ϕ by $r_j^3 + ar_j + b$ for $j = i+1, \ldots, t$.
11. Accept all the signatures if and only if $\phi = 0$.

We have used the best formulas supplied in [9, 16]. We only used the built-in field arithmetic provided by the calculator. Since all algorithms are evaluated in terms of number of field operations, this gives a fare comparison of experimental data with the theoretical estimates.

In Table 3.3, we have listed the average times for carrying out single elliptic-curve scalar multiplications in the NIST elliptic curves over prime fields. When a batch of t signatures is of concern, individual verification calls for t double scalar multiplications, whereas ECDSA batch verification algorithms involve between 2 and $t+1$ (both inclusive) scalar multiplications, irrespective of which batch-verification algorithm we use. Table 3.3 also lists the times for single square-root calculations in the underlying fields.

Individual verification can be speeded up using fixed-base scalar multiplication at the cost of some precomputation time and the requirement of substantial storage. If this storage can be afforded, then the fastest individual verification can be obtained using fixed-based double scalar multiplication. The corresponding timing figures are shown in Table 3.4.

Table 3.5 lists the worst-case overheads associated with the three algorithms N, S1 and S2. These overhead figures do not include the scalar-multiplication times. Indeed, Steps 1–4 are common to all the three batch-verification algorithms. Two variants of the naive algorithm N are experimented with. Algorithm 3.1 is specified as N in the table. If we remove the checking of $m = 2^t$ conditions (applicable to ECDSA$^{\#}$ which appends one bit to each ECDSA signature), we arrive at the algorithm indicated as N$'$. The faster variants S1$'$ and S2$'$ of the symbolic-computation algorithms are also implemented. The algorithms S1, S1$'$ and S2 become impractical for batch sizes $t > 6$, so these algorithms are not implemented for $t = 7$ and $t = 8$. All these running times pertain to the case of *no randomizers*. The effects of randomizers are illustrated in Chapter 5.

Table 3.5 indicates that Algorithm S1 is not very practical, since the overhead increases rapidly with the batch size. Only for $t \leqslant 4$ and for large fields, Algorithm S1 is more efficient than Algorithm N. However, its improved variant S1$'$ significantly outperforms S1 in all cases of primes and batch sizes. Indeed, S1$'$ outperforms even N in most of these studied cases.

Algorithm S2′ is always faster than Algorithm S2, and is more efficient than Algorithm N in all reported cases, whereas in all cases for $t \leqslant 4$ Algorithm S2 is more efficient than Algorithm N and in some cases (large fields) for $t = 5$ and $t = 6$ too. The superiority between Algorithms N and S2 is determined by the relative cost of square-root computations in $\mathbb{F}_q$ and symbolic manipulations in $\mathbb{F}_q[y_1, y_2, \ldots, y_t]$.

All of the batch-verification algorithms N, S1, S1′, S2, S2′ become impractical beyond some small values of t (since their running times are exponential in t). So, we have restricted our experiments only to the values $2 \leqslant t \leqslant 8$. Algorithm N′, on the other hand, does not carry out any effort exponential in the batch size t. Consequently, Algorithm N′ eventually outperforms all the other five algorithms for large values of t. Our experiments reveal that for small batch sizes, symbolic computation provides faster alternatives. It is also worthwhile to note here that Algorithm N′ makes use of *extra* information. The other five algorithms, on the other hand, are fully compliant with standardized ECDSA signatures.

Table 3.6 records the speedup values achieved by the six algorithms N, N′, S1, S1′, S2 and S2′. Here, the speedup is computed with respect to individual verification, and incorporates both scalar-multiplication times and batch-verification overheads. Individual verification does not use fixed-base scalar multiplication in this table. In this case, the maximum achievable speedup values (t in the case of same signer, and $2t/(t+1)$ in the case of different signers) are also listed in Table 3.6, in order to indicate how our batch-verification algorithms compare with the ideal cases. The maximum speedup achieved by our fully ECDSA-compliant algorithms is 6.16 in the case of same signer, and 1.54 in the case of different signers. These records are achieved by Algorithm S2′ for the curve P-521 and P-384, respectively, and for the batch size $t = 7$.

From Table 3.6, it is evident that one should use Algorithm S2′ if extra information (a bit identifying the correct square root of each $r_i^3 + ar_i + b$) is not available. In this case, the optimal batch size is $t = 7$ (or $t = 6$ if the underlying field is small). If, on the other hand, disambiguating extra bits are appended to ECDSA signatures, one should use S2′ for $t \leqslant 4$ for (curves over) small fields and for $t \leqslant 6$ (or $t \leqslant 7$) for large fields. If the batch size increases beyond these bounds, it is preferable to use Algorithm N′.

Now, suppose that one can afford the huge storage overhead associated with the

Table 3.3: Timings (ms) for NIST prime curves

	P-192	P-224	P-256	P-384	P-521
Scalar Multiplication time	2.20	2.77	3.08	5.33	9.09
Double Scalar Multiplication time	3.39	4.22	5.27	9.78	16.16
Square-root time (in $\mathbb{F}_q$)	0.15	5.43	0.16	0.32	0.52

Table 3.4: Total individual verification times (ms) for t signatures using fixed-base double scalar multiplication
(Overhead associated with preparing the tables of 2^iQ and $2^i(P+Q)$ are included)

Coordinate	Number of Signatures (t)	P-192	P-224	P-256	P-384	P-521
Affine	2	7.43	9.19	11.13	20.94	35.05
	3	9.26	11.36	13.82	25.84	43.03
	4	11.07	13.55	16.51	30.74	51.00
	5	12.91	15.75	19.20	35.65	58.98
	6	14.73	17.93	21.87	40.55	66.95
	7	16.57	20.12	24.58	45.47	74.94
	8	18.39	22.30	27.25	50.36	82.92
	9	20.21	24.50	29.94	55.25	90.91
	10	22.03	26.69	32.63	60.14	98.88
Jacobian	2	7.70	10.04	10.84	19.12	32.62
	3	9.87	12.79	13.88	24.39	41.66
	4	12.03	15.55	16.93	29.68	50.67
	5	14.21	18.32	19.98	34.96	59.71
	6	16.39	21.05	23.02	40.22	68.72
	7	18.55	23.81	26.06	45.52	77.72
	8	20.71	26.57	29.10	50.77	86.75
	9	22.89	29.33	32.15	56.03	95.78
	10	25.05	32.07	35.18	61.34	104.80

Table 3.5: Overheads (ms) for different batch-verification algorithms

	Naive (N)							Naive (N′)						
	t							*t*						
Curve	2	3	4	5	6	7	8	2	3	4	5	6	7	8
P-192	0.08	0.27	0.74	1.66	4.01	9.51	21.56	0.02	0.03	0.05	0.06	0.07	0.07	0.08
P-224	0.09	0.29	0.79	1.82	4.36	10.30	23.31	0.02	0.04	0.05	0.06	0.08	0.08	0.09
P-256	0.09	0.29	0.78	1.80	4.31	10.13	22.99	0.02	0.04	0.05	0.06	0.08	0.08	0.09
P-384	0.11	0.33	0.91	2.08	5.26	11.96	26.95	0.03	0.04	0.06	0.07	0.09	0.10	0.11
P-521	0.13	0.40	1.10	2.60	6.21	14.68	33.35	0.03	0.05	0.07	0.09	0.10	0.12	0.13

	Symbolic (S1)					Symbolic (S1′)				
	t					*t*				
Curve	2	3	4	5	6	2	3	4	5	6
P-192	0.10	0.30	1.01	4.34	18.17	0.06	0.14	0.37	0.86	2.22
P-224	0.10	0.32	1.06	4.59	19.49	0.06	0.15	0.40	0.92	2.36
P-256	0.11	0.33	1.08	4.59	19.51	0.07	0.16	0.42	0.93	2.38
P-384	0.15	0.41	1.27	5.34	23.61	0.10	0.22	0.51	1.11	2.77
P-521	0.19	0.50	1.54	6.63	30.54	0.13	0.28	0.64	1.36	3.41

	Symbolic (S2)					Symbolic (S2′)					
	t					*t*					
Curve	2	3	4	5	6	3	4	5	6	7	8
P-192	0.07	0.16	0.45	1.91	4.77	0.06	0.16	0.35	0.64	1.40	3.34
P-224	0.08	0.17	0.48	2.00	5.02	0.07	0.17	0.38	0.69	1.51	3.62
P-256	0.08	0.18	0.48	2.00	4.97	0.08	0.18	0.39	0.70	1.50	3.58
P-384	0.11	0.22	0.58	2.22	5.61	0.11	0.24	0.48	0.85	1.78	4.26
P-521	0.14	0.28	0.70	2.65	6.70	0.17	0.32	0.60	1.06	2.21	5.41

Table 3.6: Speedup obtained by different batch-verification algorithms
(Fixed-base scalar multiplication *not* used during individual verification)
(Experiment not carried out for cells marked as –)

Curve	t	Same signer						Different signers					
		N	N′	S1	S1′	S2	S2′	N	N′	S1	S1′	S2	S2′
P-192	2	1.80	1.83	1.94	1.97	1.96	–	0.97	0.98	1.01	1.02	1.01	–
	3	2.47	2.63	2.76	2.88	2.86	2.95	1.07	1.10	1.12	1.14	1.12	1.15
	4	2.87	3.36	3.08	3.61	3.53	3.82	1.10	1.16	1.13	1.19	1.13	1.22
	5	2.92	4.04	2.19	3.99	3.20	4.53	1.09	1.21	0.97	1.21	0.97	1.25
	6	2.45	4.67	0.94	3.63	2.49	5.05	1.00	1.24	0.61	1.15	0.61	1.27
	7	1.70	5.26	–	–	–	4.95	0.84	1.27	–	–	–	1.25
	8	1.04	5.81	–	–	–	4.03	0.64	1.29	–	–	–	1.17
P-224	2	0.56	0.56	1.95	1.97	1.96	–	0.56	0.56	2.02	2.04	2.02	–
	3	0.61	0.62	2.79	2.90	2.88	2.95	0.57	0.58	2.20	2.26	2.20	2.30
	4	0.63	0.65	3.20	3.65	3.59	3.85	0.58	0.59	2.15	2.34	2.15	2.42
	5	0.64	0.67	2.40	4.11	3.39	4.59	0.57	0.60	1.65	2.32	1.65	2.47
	6	0.62	0.69	1.07	3.85	2.74	5.16	0.54	0.60	0.87	2.13	0.87	2.48
	7	0.56	0.70	–	–	–	5.16	0.50	0.60	–	–	–	2.38
	8	0.48	0.71	–	–	–	4.31	0.43	0.61	–	–	–	2.13
P-256	2	1.86	1.88	1.96	1.97	1.97	–	1.09	1.10	1.13	1.13	1.13	–
	3	2.62	2.73	2.82	2.91	2.90	2.96	1.21	1.23	1.25	1.27	1.25	1.28
	4	3.15	3.54	3.32	3.70	3.67	3.87	1.25	1.31	1.28	1.33	1.28	1.35
	5	3.35	4.30	2.67	4.25	3.62	4.66	1.25	1.36	1.14	1.36	1.14	1.40
	6	3.00	5.01	1.28	4.13	3.09	5.30	1.18	1.40	0.77	1.32	0.77	1.42
	7	2.23	5.70	–	–	–	5.45	1.03	1.43	–	–	–	1.41
	8	1.43	6.35	–	–	–	4.76	0.81	1.45	–	–	–	1.35
P-384	2	1.86	1.87	1.97	1.98	1.98	–	1.17	1.17	1.21	1.22	1.21	–
	3	2.65	2.72	2.88	2.93	2.93	2.97	1.30	1.31	1.35	1.36	1.36	1.37
	4	3.27	3.52	3.54	3.80	3.78	3.90	1.36	1.40	1.40	1.44	1.44	1.45
	5	3.63	4.27	3.23	4.49	4.07	4.77	1.37	1.45	1.31	1.48	1.43	1.51
	6	3.46	4.98	1.76	4.68	3.81	5.52	1.32	1.49	0.96	1.46	1.37	1.54
	7	2.85	5.65	–	–	–	5.92	1.20	1.52	–	–	–	1.54
	8	1.99	6.28	–	–	–	5.57	1.01	1.55	–	–	–	1.50
P-521	2	1.86	1.88	1.98	1.98	1.98	–	1.14	1.14	1.18	1.18	1.18	–
	3	2.68	2.73	2.91	2.95	2.95	2.97	1.27	1.28	1.32	1.32	1.32	1.33
	4	3.34	3.53	3.65	3.85	3.83	3.92	1.33	1.36	1.38	1.40	1.40	1.41
	5	3.78	4.29	3.55	4.61	4.30	4.82	1.35	1.41	1.32	1.45	1.41	1.47
	6	3.80	5.00	2.08	4.95	4.24	5.63	1.33	1.45	1.03	1.45	1.38	1.50
	7	3.28	5.68	–	–	–	6.16	1.24	1.48	–	–	–	1.51
	8	2.41	6.32	–	–	–	5.99	1.08	1.50	–	–	–	1.48

Table 3.7: Speedup obtained by different batch-verification algorithms
(Fixed-base scalar multiplication *used* during individual verification)
(Experiment not carried out for cells marked as –)

Curve	t	Same signer					
		N	N′	S1	S1′	S2	S2′
P-192	2	1.97	2.00	2.13	2.15	2.15	–
	3	2.25	2.39	2.51	2.62	2.61	2.68
	4	2.34	2.74	2.52	2.94	2.88	3.12
	5	2.23	3.07	1.67	3.04	2.44	3.45
	6	1.77	3.38	0.68	2.63	1.81	3.66
	7	1.19	3.67	–	–	–	3.46
	8	0.70	3.94	–	–	–	2.73
P-224	2	0.61	0.61	2.13	2.15	2.14	–
	3	0.55	0.55	2.50	2.60	2.59	2.65
	4	0.51	0.52	2.57	2.93	2.88	3.09
	5	0.47	0.50	1.79	3.06	2.53	3.42
	6	0.44	0.49	0.76	2.72	1.94	3.65
	7	0.38	0.48	–	–	–	3.51
	8	0.31	0.47	–	–	–	2.84
P-256	2	1.96	1.98	2.07	2.08	2.08	–
	3	2.29	2.39	2.47	2.55	2.54	2.58
	4	2.47	2.77	2.60	2.90	2.87	3.03
	5	2.44	3.13	1.95	3.10	2.64	3.39
	6	2.07	3.47	0.88	2.86	2.14	3.66
	7	1.49	3.80	–	–	–	3.63
	8	0.92	4.10	–	–	–	3.08
P-384	2	1.99	2.00	2.11	2.12	2.12	–
	3	2.33	2.40	2.54	2.58	2.58	2.61
	4	2.57	2.76	2.78	2.99	2.97	3.07
	5	2.65	3.11	2.36	3.27	2.97	3.47
	6	2.39	3.44	1.21	3.23	2.63	3.81
	7	1.90	3.75	–	–	–	3.93
	8	1.28	4.04	–	–	–	3.59
P-521	2	1.88	1.89	2.00	2.00	2.00	–
	3	2.30	2.34	2.50	2.53	2.53	2.55
	4	2.62	2.77	2.86	3.02	3.01	3.07
	5	2.76	3.13	2.59	3.37	3.14	3.52
	6	2.63	3.45	1.43	3.42	2.93	3.89
	7	2.17	3.76	–	–	–	4.08
	8	1.54	4.05	–	–	–	3.84

precomputed tables in fixed-base double scalar multiplication. Table 3.4 shows the overhead associated with t individual verifications using this strategy. The Jacobian coordinate system is found to be faster than the affine coordinate system for NIST prime curves. Table 3.7 lists the speedup figures over this improved inidividual verification. Only the cases of the same signer are shown. The case of different signers is not effective for fixed-base double scalar multiplication. Although individual verification receives a significant boost from the precomputed tables, particularly for large batch sizes, the symbolic-computation algorithms are still capable of producing speedup figures above four.

3.6 Koblitz Curves

A Koblitz curve is defined over a binary field $\mathbb{F}_{2^d}$ by the equation

$$y^2 + xy = x^3 + ax^2 + 1. \tag{3.20}$$

Here, a is either 0 or 1. If $a = 0$, the cofactor is $h = 4$, whereas $h = 2$ if $a = 1$. The NIST standard lists five such curves K-163, K-233, K-283, K-409 and K-571, where the number after K- indicates the size of the field elements (that is, the extension degree d). The standard also specifies how the arithmetic of each such field $\mathbb{F}_{2^d}$ is to be implemented.

The Koblitz curves, being defined over fields of characteristic two, are subtly different from the prime curves. We can no longer use the simplified Weierstrass equation used for curves defined over large prime fields. Indeed, the two y-coordinates of points with a given x-coordinate r now satisfy the equation

$$y^2 + ry + (r^3 + ar^2 + 1) = 0. \tag{3.21}$$

The sum of the two roots is r, and their product is $r^3 + ar^2 + 1$. This calls for suitable changes in the batch-verification algorithms described so far, since they are based upon the simplified Weierstrass equation.

The NIST standard also specifies a set of random elliptic curves defined over the above five binary fields. They all correspond to the cofactor value $h = 2$. Adaptations of our batch-verification algorithms to these curves are straightforward, given those

for Koblitz curves. Consequently, we do not discuss this random family in the rest of this chapter. Indeed, following the adaptations to Koblitz curves, one can easily modify our algorithms to work for any non-supersingular elliptic curve over $\mathbb{F}_{2^d}$.

3.7 Adaptation of the Naive Algorithms

The basic computational task involved in the algorithms N and N′ was the computation of square roots in the underlying field. In the case of binary fields, we need to solve the quadratic equation (3.21) both the roots of which lie in the field. We can use a root-finding or polynomial-factoring algorithm for computing these two roots, such as Berlekamp's trace algorithm or its randomized variant as described in Menezes et al. [38]. For quadratic polynomials over $\mathbb{F}_{2^d}$, these algorithms take $O(d^3)$ expected running time—theoretically the same as the Tonelli-Shanks algorithm. In practice, these algorithms are significantly slower than modular square-root computations in prime fields of comparable sizes. The implication of this is that the naive algorithms are rather inefficient for fields of characteristic two, and consequently, our symbolic-computation algorithms exhibit noticeably better performances than naive algorithms.

3.7.1 Adaptation of the Symbolic-computation Algorithms

The symbolic-computation algorithms S1, S1′, S2 and S2′ require modifications in view of the changed equation for y in terms of r. In all these algorithms, we can keep each y_i-degree below 2 by making repeated substitutions $y_i^2 = r_i y_i + (r_i^3 + a r_i^2 + 1)$. The introduction of the term $r_i y_i$ creates some trouble with the parity of the degrees of the terms. The sequence of linearized equations generated in Steps 7–9 of Algorithm S1 now involves also the odd-degree monomials in $y_1, y_2, \ldots, y_t$, that is, the count of linearized variables increases from $2^{t-1} - 1$ to $2^t - 1$. This results in a degradation of the performance of S1.

In both Algorithms S1 (Step 7) and S2 (Steps 8 and 10), we need to eliminate y_i from a multivariate polynomial linear with respect to y_i. Let us write such a polynomial as

$$\phi = u y_i + v,$$

where u and v are multivariate polynomials not containing y_i. Multiplying ϕ by $u(y_i + r_i) + v$ gives

$$(u(y_i + r_i) + v)\phi = (u(y_i + r_i) + v)(uy_i + v) = u^2(y_i^2 + r_i y_i) + uvr_i + v^2$$
$$= u^2(r_i^3 + ar_i^2 + 1) + uvr_i + v^2.$$

The last expression is free from y_i. The polynomials u^2, uv, v^2 may contain y_j^2 for $j \neq i$. Each such occurrence of y_j^2 is to be substituted by $r_j y_j + (r_j^3 + ar_j^2 + 1)$ in order to reduce the y_j-degree of the expression to below 2. Algorithms S2 and S2′ do not deal with linearized systems, but still experience an increase in the count of non-zero terms from a maximum of 2^{t-1} to a maximum of 2^t. This reduces the performance benefits of S2 and S2′, but this degradation is much more graceful than for S1. The variant S1′ is also less affected than S1.

3.7.2 Experimental Results

We continued our experiments with GP/PARI 2.5.0. It seems that the GP/PARI routines for field arithmetic over binary fields $\mathbb{F}_{2^d}$ are not very optimized. Table 3.8 indicates that elliptic-curve scalar-multiplication times for Koblitz curves are over three orders of magnitude slower than those for prime curves (Table 3.3) for fields of comparable sizes. A more critical difference is in the times for solving quadratic equations over the underlying fields. For prime curves, the modular square-root algorithm is over 20 times faster than individual scalar multiplication. For Koblitz curves, we use a root-finding algorithm based on the half-trace method. The apparent inefficiency of the library for binary fields, however, does not invalidate our experimental conclusions, since our speedup figures are ratios.

Table 3.10 lists the worst-case overheads (excluding the computation of the point $R = (\alpha, \beta)$) associated with the five batch-verification algorithms studied. Both the variants of the naive algorithm are crippled by huge running times taken by root-finding algorithms. The symbolic-computation algorithms do not involve this built-in function and perform significantly better than the naive algorithms. Since Algorithm S1 involves generating and solving large linear systems, we have not implemented it. Its efficient variant S1′ is only implemented. Once again, randomizers are not considered in these running times. We deal with randomization techniques in detail in Chapter 5.

Table 3.8: Timings (ms) for NIST Koblitz curves

	K-163	K-233	K-283	K-409	K-571
Scalar multiplication time	68.84	139.41	216.00	506.56	1072.00
Double scalar multiplication time	248.00	579.39	908.62	2266.63	4968.33
Root computation time (Half-Trace method)	3.32	5.44	8.31	15.12	30.83

Table 3.9: Total individual verification times (ms) for t signatures using fixed-base double scalar multiplication

(Overhead associated with preparing the tables of 2^iQ and $2^i(P+Q)$ are included)

Coordinate	Number of Signatures	K-163	K-233	K-283	K-409	K-571
Affine	2	597.73	1343.18	2085.30	5011.02	10959.24
	3	734.20	1644.68	2549.67	6114.47	13349.05
	4	870.48	1946.21	3014.10	7215.79	15735.61
	5	1006.73	2247.62	3478.33	8318.79	18125.51
	6	1143.28	2549.18	3942.87	9421.87	20516.72
	7	1279.42	2850.62	4407.22	10524.41	22902.89
	8	1415.93	3151.96	4871.86	11626.49	25290.65
	9	1552.10	3453.80	5335.79	12729.09	27684.71
	10	1688.73	3754.84	5800.57	13833.68	30070.93
López-Dahab	2	1080.09	2327.74	3610.66	8714.05	19617.63
	3	1408.92	3046.51	4725.96	11440.08	25796.96
	4	1738.05	3764.85	5841.84	14162.52	31977.52
	5	2066.82	4483.77	6955.23	16884.55	38167.97
	6	2395.84	5201.61	8071.48	19606.45	44343.75
	7	2724.61	5920.04	9185.28	22329.80	50523.72
	8	3053.74	6638.19	10301.16	25050.64	56711.33
	9	3382.71	7356.67	11416.44	27776.58	62888.67
	10	3711.50	8075.64	12532.36	30499.59	69068.49

Table 3.10: Overheads (ms) for different batch-verification algorithms

	Naive (N)					
	t					
Curve	2	3	4	5	6	7
K-163	9.72	37.23	107.73	280.36	685.05	1666.47
K-233	15.34	57.74	164.76	432.01	1071.94	2562.47
K-283	19.67	73.92	212.28	560.11	1378.18	3335.52
K-409	32.94	123.85	354.92	934.88	2319.59	5615.76
K-571	53.23	200.76	581.65	1527.10	3785.93	9118.68

	Naive (N′)					
	t					
Curve	2	3	4	5	6	7
K-163	2.48	4.93	6.88	9.04	11.20	13.41
K-233	4.04	7.49	10.76	14.01	17.43	20.90
K-283	5.04	9.57	13.68	18.33	22.18	26.90
K-409	8.56	16.21	23.08	30.30	37.28	45.39
K-571	13.86	26.03	37.57	49.14	60.49	73.41

	Symbolic (S1′)				
	t				
Curve	2	3	4	5	6
K-163	9.96	26.18	78.25	429.00	848.39
K-233	17.10	40.52	123.64	733.51	1380.45
K-283	24.07	52.77	160.52	1002.30	1857.51
K-409	44.26	88.45	271.30	1803.57	3318.89
K-571	79.85	142.24	439.67	3145.40	5355.98

	Symbolic (S2)					Symbolic (S2′)					
	t					t					
Curve	2	3	4	5	6	2	3	4	5	6	7
K-163	5.08	34.28	170.62	677.68	2154.77	3.84	8.77	20.62	70.44	156.27	503.74
K-233	7.46	52.92	293.90	1164.91	3607.88	5.64	13.10	31.45	107.36	240.64	779.22
K-283	9.65	68.22	396.69	1583.60	4888.38	7.24	16.87	40.31	139.09	313.12	1013.23
K-409	16.38	113.08	289.97	2918.75	8742.71	12.41	29.03	68.02	231.84	521.76	1702.94
K-571	26.56	183.93	1260.72	5173.57	15222.20	20.14	46.91	110.02	376.45	861.00	2773.46

Table 3.11: Speedup obtained by different batch-verification algorithms (Fixed-base scalar multiplication *not* used for individual verification)

Curve	t	Same signer					Different signers				
		N	N′	S1′	S2	S2′	N	N′	S1′	S2	S2′
K-163	2	1.79	1.88	1.87	1.93	1.95	1.24	1.28	1.27	1.30	1.31
	3	2.23	2.71	2.52	2.40	2.82	1.28	1.42	1.37	1.33	1.45
	4	2.13	3.49	2.55	1.79	3.48	1.18	1.51	1.30	1.07	1.51
	5	1.58	4.22	1.21	0.84	3.31	0.97	1.57	0.82	0.63	1.42
	6	0.98	4.89	0.84	0.36	2.81	0.70	1.61	0.62	0.31	1.29
	7	0.53	5.53	–	–	1.50	0.43	1.64	–	–	0.91
K-233	2	1.83	1.90	1.88	1.95	1.96	1.25	1.29	1.28	1.31	1.32
	3	2.37	2.76	2.62	2.52	2.87	1.32	1.44	1.40	1.37	1.47
	4	2.40	3.58	2.77	1.95	3.59	1.26	1.53	1.36	1.13	1.53
	5	1.89	4.36	1.38	0.97	3.61	1.08	1.59	0.89	0.70	1.48
	6	1.21	5.09	1.01	0.43	3.22	0.80	1.63	0.71	0.36	1.38
	7	0.68	5.78	–	–	1.84	0.53	1.66	–	–	1.03
K-283	2	1.85	1.90	1.89	1.96	1.97	1.26	1.29	1.29	1.31	1.32
	3	2.44	2.78	2.67	2.59	2.89	1.35	1.44	1.41	1.39	1.47
	4	2.55	3.61	2.92	2.09	3.66	1.30	1.53	1.39	1.17	1.54
	5	2.09	4.39	1.51	1.07	3.78	1.14	1.59	0.94	0.75	1.51
	6	1.39	5.14	1.13	0.49	3.48	0.88	1.64	0.77	0.40	1.42
	7	0.79	5.85	–	–	2.09	0.59	1.67	–	–	1.10
K-409	2	1.88	1.93	1.92	1.97	1.98	1.28	1.30	1.30	1.32	1.32
	3	2.57	2.83	2.76	2.70	2.92	1.38	1.46	1.44	1.42	1.48
	4	2.84	3.70	3.16	3.11	3.75	1.37	1.55	1.45	1.44	1.56
	5	2.50	4.53	1.80	1.29	4.07	1.25	1.61	1.05	0.85	1.55
	6	1.78	5.33	1.40	0.62	3.96	1.02	1.65	0.89	0.49	1.49
	7	1.05	6.09	–	–	2.61	0.73	1.69	–	–	1.23
K-571	2	1.90	1.93	1.93	1.98	1.98	1.29	1.30	1.30	1.32	1.33
	3	2.64	2.84	2.81	2.76	2.94	1.40	1.46	1.45	1.44	1.48
	4	3.01	3.72	3.32	2.52	3.80	1.41	1.55	1.48	1.30	1.57
	5	2.80	4.57	2.03	1.46	4.25	1.32	1.62	1.12	0.92	1.57
	6	2.10	5.38	1.72	0.74	4.28	1.12	1.66	1.00	0.57	1.54
	7	1.31	6.17	–	–	3.05	0.84	1.69	–	–	1.32

Table 3.12: Speedup obtained by different batch-verification algorithms
(Fixed-base scalar multiplication *used* during individual verification)
(Experiment not carried out for cells marked as –)

Curve	t	Same signer				
		N	N′	S1′	S2	S2′
K-163	2	3.88	4.07	4.05	4.19	4.22
	3	3.97	4.81	4.48	4.27	5.01
	4	3.36	5.51	4.03	2.82	5.50
	5	2.32	6.16	1.78	1.23	4.84
	6	1.36	6.77	1.16	0.50	3.89
	7	0.70	7.34	–	–	1.99
K-233	2	4.40	4.57	4.54	4.69	4.72
	3	4.66	5.43	5.15	4.96	5.63
	4	4.18	6.25	4.84	3.40	6.27
	5	3.05	7.02	2.22	1.56	5.82
	6	1.84	7.75	1.54	0.66	4.91
	7	0.99	8.44	–	–	2.69
K-283	2	4.45	4.60	4.57	4.72	4.75
	3	4.80	5.47	5.26	5.10	5.68
	4	4.45	6.29	5.09	3.64	6.38
	5	3.37	7.07	2.43	1.73	6.09
	6	2.12	7.82	1.72	0.74	5.29
	7	1.15	8.52	–	–	3.05
K-409	2	4.66	4.76	4.74	4.87	4.89
	3	5.17	5.69	5.55	5.43	5.87
	4	5.05	6.58	5.62	5.54	6.67
	5	4.11	7.43	2.95	2.12	6.68
	6	2.75	8.26	2.17	0.97	6.14
	7	1.56	9.04	–	–	3.87
K-571	2	4.85	4.94	4.93	5.05	5.06
	3	5.48	5.90	5.84	5.73	6.09
	4	5.52	6.83	6.09	4.62	6.98
	5	4.74	7.72	3.43	2.48	7.19
	6	3.36	8.59	2.74	1.18	6.83
	7	2.00	9.41	–	–	4.66

Speedup figures over individual verification (without precomputation and storage overheads associated with fixed-base double scalar multiplication) are listed in Table 3.11. The maximum speedup achieved by the naive algorithms is 3.01 for the case of the same signer and 1.41 for the case of different signers, indicating that these algorithms are not effective at all for Koblitz curves. The symbolic-computation algorithms, on the other hand, exhibit a similar pattern for Koblitz curves, as they have done for prime curves. The maximum recorded speedup is achieved always by S2$'$. For the curve K-571 and for $t = 6$, this is 4.28 in the case of the same signer, and 1.57 at $t = 5$ in the case of different signers. Even for small fields, S2 and S2$'$ exhibit decent performance. The performance of S1$'$ is between those of S2 and S2$'$.

Table 3.9 lists the individual verification times if we use fixed-base double scalar multiplication. Both affine and López-Dahab coordinate systems are implemented. In all our experiments, the affine coordinate system gives better performance.

Speedup figures obtained by our batch-verification algorithms over individual verification with fixed-base double scalar multiplication are presented in Table 3.12. This only corresponds to the case of the same signer. The point P is a system-wide constant, so the multiples 2^iP are precomputed and used in all individual verifications. The public key Q of the signer is assumed constant for a small number of signatures (like those in a batch). For each such set of signatures, we need to precompute two tables for storing the points 2^iQ and $2^i(P+Q)$. When this precomputation overhead is amortized over a relatively large number t of signatures, we get practical benefits. Table 3.12 indicates that t should be $\geqslant 7$ for this heuristic to be practically useful. For smaller batch sizes, it is preferable to use τ-NAF scalar multiplications during each individual verification (without precomputing or storing of 2^iP, 2^iQ and $2^i(P+Q)$).

3.8 Chapter Summary

In this chapter, we have proposed six algorithms for the batch verification of ECDSA signatures. To the best of our knowledge, these are the first batch-verification methods ever proposed for ECDSA. In particular, development of algorithms based upon the concept of symbolic manipulations appears to be a novel approach in the history of ECDSA batch-verification algorithms.

Chapter 4

ECDSA Batch Verification Using Summation Polynomials

Several batch-verification algorithms for original ECDSA signatures are proposed for the first time in Chapter 3. Two of these algorithms are based on the naive idea of taking square roots in the underlying fields, and the others perform symbolic manipulations to verify small batches of ECDSA signatures. In this chapter, we use elliptic-curve summation polynomials to design a new and more efficient ECDSA batch-verification algorithm which is theoretically and experimentally much faster than the symbolic algorithms of Chapter 3. Our experiments on NIST prime and Koblitz curves demonstrate that our proposed algorithm increases the optimal batch size from seven to nine. We finally establish a connection between our symbolic-computation and summation-polynomial algorithms.

4.1 Batch-Verification Algorithm SP for ECDSA

The new batch-verification algorithm we propose in this chapter is based on elliptic-curve summation polynomials introduced by Semaev [67] in the context of improving the known bounds of the index-calculus method for solving the elliptic-curve discrete-logarithm problem.

Let E be the elliptic curve defined over a prime field $\mathbb{F}_q$ by the equation

$$E : y^2 = x^3 + ax + b. \tag{4.1}$$

Let $x_1, x_2, \ldots, x_t$ be $t \geqslant 2$ elements of $\mathbb{F}_q$. The t-variable summation polynomial f_t is recursively defined as follows:

$$\begin{aligned} f_2(x_1, x_2) &= x_1 - x_2, & (4.2)\\ f_3(x_1, x_2, x_3) &= (x_1 - x_2)^2 {x_3}^2 - 2((x_1 + x_2)(x_1 x_2 + a) + 2b)x_3 + \\ &\qquad ((x_1 x_2 - a)^2 - 4b(x_1 + x_2)), & (4.3)\\ f_t(x_1, x_2, \ldots, x_t) &= \mathrm{Res}_X(f_{t-k}(x_1, \ldots, x_{t-k-1}, X), f_{k+2}(x_{t-k}, \ldots, x_t, X)) \\ &\qquad \text{for } t \geqslant 4 \text{ and for any } k \text{ in the range } 1 \leqslant k \leqslant t-3. & (4.4) \end{aligned}$$

Here, Res_X stands for the resultant [39] of two polynomials with respect to the variable X. Semaev proves that $f_t(x_1, x_2, \ldots, x_t) = 0$ if and only if there exist $y_1, y_2, \ldots, y_t \in \overline{\mathbb{F}}_q$ with (x_i, y_i) satisfying Eqn (4.1) for all $i = 1, 2, \ldots, t$ such that we have the following sum in the elliptic-curve group $E(\overline{\mathbb{F}}_q)$:

$$(x_1, y_1) + (x_2, y_2) + \cdots + (x_t, y_t) = \mathscr{O}, \tag{4.5}$$

where $\mathscr{O}$ is the point at infinity on E, and $\overline{\mathbb{F}}_q$ is the algebraic closure of $\mathbb{F}_q$.

For the batch verification of t ECDSA signatures (M_i, r_i, s_i), we first compute the numeric sum R on the right side of Eqn (3.2). Let $R = (\alpha, \beta)$, where $\alpha, \beta \in \mathbb{F}_q$. Eqn (3.2) or (3.3) can be rewritten as

$$(r_1, y_1) + (r_2, y_2) + \cdots + (r_t, y_t) + (\alpha, -\beta) = \mathscr{O}. \tag{4.6}$$

Eqn (4.5) implies that Eqn (4.6) is equivalent to the condition $f_{t+1}(r_1, r_2, \ldots, r_t, \alpha) = 0$. Algorithm 4.1 incorporates this idea to verify a batch of t ECDSA signatures.

4.2 Analysis of Algorithm SP

4.2.1 Properties of Summation Polynomials

For $t = 2$ or 3, we straightaway use the formulas given in Eqn (4.2) or (4.3). For $t \geqslant 4$, we make two recursive calls as given in Eqn (4.4). In order to optimize efficiency,

Algorithm 4.1 ECDSA Batch-verification Algorithm SP for NIST Prime Curves

INPUT: Domain Parameters, ECDSA signatures (M_1, r_1, s_1), (M_2, r_2, s_2), ..., (M_t, r_t, s_t) and public key Q of the signer.

OUTPUT: Accept/Reject the batch of t signatures.

1. Optional sanity check: For each $i = 1, 2, \ldots, t$, check whether $r_i^3 + ar_i + b$ is a quadratic residue modulo q. If not, reject the i-th signature and remove it from the batch. Let us assume that all the signatures in the batch pass the sanity check. (Also see Section 4.2.5.)
2. Compute $w_i = s_i^{-1} \pmod{n}$ for all $i = 1, 2, \ldots, t$.
3. Compute $u_i = H(M_i)w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
4. Compute $v_i = r_i w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
5. Compute $R = (\sum_{i=1}^{t} u_i)P + (\sum_{i=1}^{t} v_i)Q = (\alpha, \beta)$.
6. Compute the value of the summation polynomial $\phi = f_{t+1}(r_1, r_2, \ldots, r_t, \alpha)$.
7. Accept the batch of signatures if and only if $\phi = 0$.

the number of variables in each recursive call should be about $t/2$. More precisely, we always choose $k = \lceil t/2 \rceil$ (this is in the allowed range of values of k), so the first recursive call computes $f_{\lceil t/2 \rceil + 1}$ and the second recursive call computes $f_{\lfloor t/2 \rfloor + 1}$. The leaves of the recursion tree deal with the base cases of Eqns (4.2) and (4.3).

Theorem 4.2.1. *Let $t = 2^h + 2$ for some $h \geqslant 0$. Then, the recursion tree of computing the f_t is a complete binary tree of height h, and all the leaves correspond to the computation of f_3 by Eqn (4.3).*

Proof. We proceed by induction on h. For $h = 0$, we compute f_3 straightaway from Eqn (4.3) without making any recursive call, that is, the recursion tree is of height zero. For $h \geqslant 1$, suppose that the assertion holds for the computation of $f_{2^{h-1}+1}$. The computation of f_t proceeds as

$$f_t(x_1, x_2, \ldots, x_t) = \mathrm{Res}_X(f_{\frac{t}{2}+1}(x_1, \ldots, x_{\frac{t}{2}}, X), f_{\frac{t}{2}+1}(x_{\frac{t}{2}+1}, \ldots, x_t, X)).$$

Here, t is even, so $\lceil t/2 \rceil = \lfloor t/2 \rfloor = t/2$. Moreover, $t/2 = 2^{h-1} + 1$, so by the induction hypothesis, the sub-trees for the two recursive calls are complete binary trees with

each leaf computing f_3. □

In general, let h be the height of the recursion tree for the computation of f_t. By Theorem 4.2.1, we have $2^{h-1}+2 < t \leqslant 2^h+2$, that is, $\log_2(t-2) \leqslant h < 1+\log_2(t-2)$, that is, the height of the recursion tree is $\Theta(\log t)$.

Theorem 4.2.2. *Let $t = 2^h+2$ with $h \geqslant 1$. If we compute f_t recursively as*

$$f_t(x_1,x_2,\ldots,x_t) = Res_X\left(f_{\frac{t}{2}+1}\left(x_1,\ldots,x_{\frac{t}{2}},X\right), f_{\frac{t}{2}+1}\left(x_{\frac{t}{2}+1},\ldots,x_t,X\right)\right), \tag{4.7}$$

then we take the resultant of two polynomials in X of degrees equal to $2^{\left(\frac{t-2}{2}\right)}$.

Proof. We first supply a direct proof based upon induction on h. For the base case, that is, $h = 1$, we have four elements x_1,x_2,x_3,x_4. We compute

$$f_4(x_1,x_2,x_3,x_4) = \mathrm{Res}_X(f_3(x_1,x_2,X), f_3(x_3,x_4,X)).$$

By Eqn (4.3), the X-degree of each of the two arguments of Res_X is $2 = 2^{\left(\frac{4-2}{2}\right)}$.

Now, let $h \geqslant 2$ and $t' = \frac{t}{2}+1 = 2^{h-1}+2$. We have $f_{t'}(x_1,x_2,\ldots,x_{t'-1},X) = \mathrm{Res}_Y\left(f_{\frac{t'}{2}+1}\left(x_1,\ldots,x_{\frac{t'}{2}},Y\right), f_{\frac{t'}{2}+1}\left(x_{\frac{t'}{2}+1},\ldots,x_{t'-1},X,Y\right)\right)$. We inductively assume that this computation of $f_{t'}$ involves the resultant calculation of two polynomials of Y-degree $\delta = 2^{\left(\frac{t'-2}{2}\right)}$ each. But each summation polynomial is symmetric about its arguments. Therefore, the X-degree of the second argument is again $\delta = 2^{\left(\frac{t'-2}{2}\right)}$. Let us write

$$f_{\frac{t'}{2}+1}\left(x_1,\ldots,x_{\frac{t'}{2}},Y\right) = a_\delta Y^\delta + a_{\delta-1}Y^{\delta-1} + \cdots + a_0, \tag{4.8}$$

$$f_{\frac{t'}{2}+1}\left(x_{\frac{t'}{2}+1},\ldots,x_{t'-1},X,Y\right) = b_\delta Y^\delta + b_{\delta-1}Y^{\delta-1} + \cdots + b_0. \tag{4.9}$$

Here, the coefficients a_i do not involve X, whereas the coefficients b_i are polynomials in X. Since the X-degree of the second polynomial is δ, and the polynomial is symmetric about X and Y, we conclude that the X-degree of b_δ is δ. The X-degrees of the other coefficients b_i are $\leqslant \delta$.

The $(2\delta)\times(2\delta)$ Sylvester matrix [39] of the polynomials in Eqns (4.8) and (4.9)

is

$$S = \begin{pmatrix} a_\delta & a_{\delta-1} & \cdots & a_0 & 0 & \cdots & 0 \\ 0 & a_\delta & a_{\delta-1} & \cdots & a_0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & a_\delta & a_{\delta-1} & \cdots & a_0 \\ b_\delta & b_{\delta-1} & \cdots & b_0 & 0 & \cdots & 0 \\ 0 & b_\delta & b_{\delta-1} & \cdots & b_0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & b_\delta & b_{\delta-1} & \cdots & b_0 \end{pmatrix}$$

The X-degree of $\det S$ is the X-degree of b_δ^δ, that is, $\delta^2 = 2^{t'-2} = 2^{\left(\frac{t-2}{2}\right)}$. Similarly, the X-degree of $f_{\frac{t}{2}+1}\left(x_{\frac{t}{2}+1},\ldots,x_t,X\right)$ is again $2^{\left(\frac{t-2}{2}\right)}$. □

Alternative proof In [67], Semaev proves that the summation polynomial f_k, $k \geqslant 3$, is of the form

$$f_k(x_1,x_2,\ldots,x_k) = f_{k-1}^2(x_1,x_2,\ldots,x_{k-1})x_k^{2^{k-2}} + \cdots . \tag{4.10}$$

Now, we put $k = \frac{t}{2}+1$ and $x_k = X$, and rewrite Eqn (4.10) as

$$f_{\frac{t}{2}+1}\left(x_1,\ldots,x_{\frac{t}{2}},X\right) = f_{\frac{t}{2}}^2\left(x_1,x_2,\ldots,x_{\frac{t}{2}}\right)X^{2^{\left(\frac{t-2}{2}\right)}} + \cdots ,$$

that is, $f_{\frac{t}{2}+1}\left(x_1,\ldots,x_{\frac{t}{2}},X\right)$ is a polynomial of degree $2^{\left(\frac{t-2}{2}\right)}$ in X. Likewise, the polynomial $f_{\frac{t}{2}+1}\left(x_{\frac{t}{2}+1},\ldots,x_n,X\right)$ too is a polynomial of degree $2^{\left(\frac{t-2}{2}\right)}$ in X. •

Theorem 4.2.2 can be generalized to any value of t (that is, values not only of the form 2^h+2). However, the resulting formulas involve many floor and ceiling expressions. For the sake of simplicity, we restrict only to the special case which already portrays the performance of SP as a function of t.

4.2.2 A Strategy to Handle the Variables in the Recursion Tree

Let r_i denote the known x-coordinates, and X_j the variables used in the recursion of Eqn (4.4). For achieving good performance, we reduce the number of variables in each node of the recursion tree. Each child of the root has one variable. Now, let

some node compute the summation polynomial of $r_i, r_{i+1}, \ldots, r_{i+k-1}, X_j$ (the case of one variable). Its two child nodes compute the summation polynomials of $r_i, r_{i+1}, \ldots, r_{i+\lceil k/2 \rceil - 1}, X_{j'}$ and $r_{i+\lceil k/2 \rceil}, \ldots, r_{i+k-1}, X_j, X_{j'}$. On the other hand, if a node computes the summation polynomial of $r_i, r_{i+1}, \ldots, r_{i+k-1}, X_j, X_{j'}$ (the case of two variables), then its two child nodes compute the summation polynomials of $r_i, r_{i+1}, \ldots, r_{i+\lceil k/2 \rceil - 1}$, $X_j, X_{j''}$ and $r_i, r_{i+1}, \ldots, r_{i+k-1}, X_{j'}, X_{j''}$. This is allowed since summation polynomials are symmetric about the arguments. It is thus ensured that the number of variables in each node never exceeds two. At each node of the leftmost paths in the two subtrees of the root, the number of variables is exactly one. At every other node in the tree except the root node, the number of variables is exactly two. Figure 4.1 shows the recursive construction of $f_{10}(r_1, r_2, \ldots, r_{10})$. Only the nodes on the paths from the root to the leaves (r_1, r_2, X_4) and (r_6, r_7, X_6) have numbers of variables equal to one.

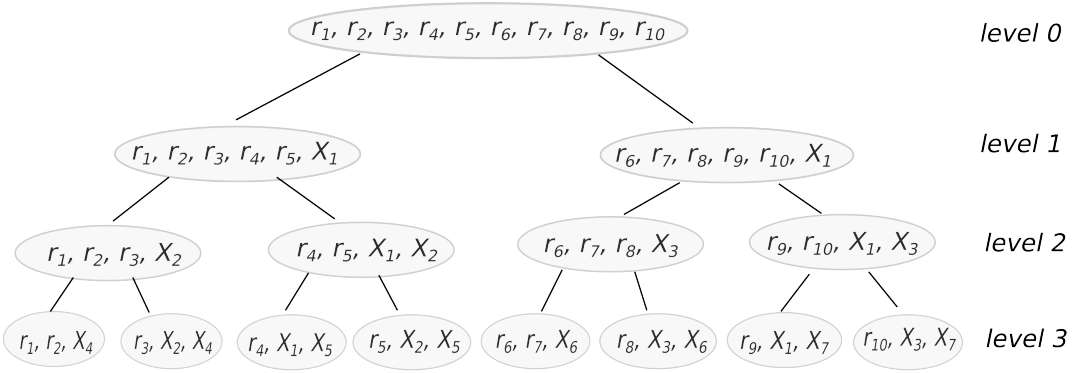

Figure 4.1: Recursion tree for computing the summation polynomial of ten variables

4.2.3 Running Time of SP

Let $C(t)$ denote the running time of Algorithm SP in the number of field operations on a batch of size t. In view of Eqn (4.6), we need to compute f_{t+1}. If $t = 2$, we use the base case which return in constant time. For $t \geqslant 3$, we use the recursive strategy of Eqn (4.7). Recursion stops in all cases at the base case of the computation of f_3. By Theorem 4.2.2, we need to take the resultant of two polynomials of degree $2^{\left(\frac{t-1}{2}\right)}$ each. The time complexity of resultant computation for two k-degree polynomials by the subresultant PRS algorithm [13, 17, 39] is $\mathrm{O}(k^2)$.

The running time of SP is dominated by the times for the computation of the resultants. The degrees of the polynomials, of which the resultant is computed, is a function of the level λ in the tree. In addition, the resultant-computation time depends

on how many variables are involved at that node, call it v. We can have $v = 0, 1, 2$ only. Let $C_v^{(\lambda)}$ denote the time for resultant computation for a given λ and v. The case of $C_0^{(\lambda)}$ occurs at level $\lambda = 0$ only. The case of $C_1^{(\lambda)}$ occurs on the leftmost paths of the two subtrees of the root. At all other nodes, the resultant-computation cost is $C_2^{(\lambda)}$.

For simplicity, we assume that the recursion tree is a complete binary tree of height h, that is, $t+1 = 2^h + 2$. We have $C_0^{(0)} = \mathrm{O}(2^{2^h})$, $C_1^{(\lambda)} = \mathrm{O}(2^{2^{h-\lambda-1}\times 3})$, and $C_2^{(\lambda)} = \mathrm{O}(2^{2^{h-\lambda+1}})$. At level zero, we have the cost C_0, whereas at any other level λ we have exactly two cases of $C_1^{(\lambda)}$ and $2^\lambda - 2$ cases of $C_2^{(\lambda)}$. Moreover, $C_1^{(\lambda)} < C_2^{(\lambda)}$ for each fixed λ. Therefore, the total cost $C(t)$ of computing f_{t+1} is of the order of

$$\begin{aligned} & C_0^{(0)} + \sum_{\lambda=1}^{h-1} \left(2C_1^{(\lambda)} + (2^\lambda - 2) C_2^{(\lambda)} \right) < C_0^{(0)} + \sum_{\lambda=1}^{h-1} 2^\lambda C_2^{(\lambda)} \\ = \; & \mathrm{O}\left(2^{2^h} + \sum_{\lambda=1}^{h-1} 2^{\lambda + 2^{h-\lambda+1}} \right). \end{aligned}$$

The inequality $\lambda + 2^{h-\lambda+1} > (\lambda+1) + 2^{h-(\lambda+1)+1}$ holds if and only if we have $2^{h-\lambda} > 1$. For all λ in the range $1 \leqslant \lambda \leqslant h-1$, we have $2^{h-\lambda} > 1$. Therefore, $C(t)$ is of the order of

$$\begin{aligned} & 2^{2^h} + \sum_{\lambda=1}^{h-1} \left[2^{\lambda + 2^{h-\lambda+1}} \right] = 2^{2^h} + \left[2^{1+2^h} + 2^{2+2^{h-1}} + \cdots + 2^{(h-1)+2^2} \right] \\ < \; & 2^{2^h} + \sum_{i=0}^{1+2^h} 2^i < 2^{2^h} + 2^{2+2^h} = 2^{2^h} + 2^{2+2^h} = 5 \times 2^{2^h}. \end{aligned}$$

Substituting 2^h by $t-1$, we see that $C(t) = \mathrm{O}(m)$, where $m = 2^t$.

The above analysis implies that the computation of the resultants at the top two levels determines the order of $C(t)$. For a general t of the form $2^{h-1} + 2 < t + 1 \leqslant 2^h + 2$, we let $\tau = \left\lceil \frac{t+1}{2} \right\rceil$, and conclude that $C(t)$ is of the order of

$$\begin{aligned} & 2^{\left\lceil \frac{t+1}{2} \right\rceil + \left\lfloor \frac{t+1}{2} \right\rfloor - 2} + 2 \times \left(2^{\left(\left\lceil \frac{\tau+1}{2} \right\rceil + \left\lfloor \frac{\tau+1}{2} \right\rfloor - 2 \right)} \right)^2 \\ = \; & 2^{t-1} + 2^{2(\tau+1)-3} \leqslant 2^{t-1} + 2^t = \frac{3}{2} \times 2^t = \mathrm{O}(m). \end{aligned}$$

4.2.4 Security of SP

In this section, we prove the equivalence between the security of Algorithm SP and the security of the standard ECDSA* batch-verification algorithm. Suppose that

the x-coordinates $r_1, r_2, \ldots, r_t$ in ECDSA signatures are available to an adversary and that the batch is accepted by Algorithm SP. By Eqn (4.6), there exist exactly two solutions $(y_1, y_2, \ldots, y_t)$ and $(-y_1, -y_2, \ldots, -y_t)$ for the y-coordinates satisfying $y_i^2 = r_i^3 + ar_i + b$ for $i = 1, 2, \ldots, t$ such that $(r_1, y_1) + (r_2, y_2) + \cdots + (r_t, y_t) = (\alpha, \beta)$, and $(r_1, -y_1) + (r_2, -y_2) + \cdots + (r_t, -y_t) = (\alpha, -\beta)$. These are the only cases in which $f_{t+1}(r_1, r_2, \ldots, r_t, \alpha) = 0$. Both these solutions are consistent with $\phi = 0$ (Step 7 of Algorithm SP). One of these solutions corresponds to the ECDSA* signatures based upon the disclosed values $r_1, r_2, \ldots, r_t$. For ECDSA*, the y-coordinates are known, and we have only one possibility $(r_1, y_1) + (r_2, y_2) + \cdots + (r_t, y_t) = (\alpha, \beta)$. Given r_i alone, the adversary can obtain the y-coordinates y_i up to sign by making t square-root computations which demand only moderate computing resources. The sign ambiguity can be removed by trying all of the 2^t sign combinations (as in Algorithm N). For small values of t (as we deal with), this too is a tolerable overhead to the adversary. To sum up, if the adversary can forge ECDSA signatures that pass Algorithm SP, (s)he can produce in feasible time ECDSA* signatures too that pass the standard ECDSA* batch-verification algorithm. The converse is obvious.

4.2.5 Necessity of the Sanity Check

The security proof in the last section assumes that all y_i reside in $\mathbb{F}_q$ itself, that is, the points (r_i, y_i) lie on the curve E defined over $\mathbb{F}_q$. The sanity check made in Step 1 of Algorithm 4.1 ensures this.

The sanity check may be unnecessary in many situations. For example, suppose that an adversary chooses an r_i for which $r_i^3 + ar_i + b$ is a quadratic non-residue modulo q. The square roots of all quadratic non-residues in $\mathbb{F}_q$ lie in $\mathbb{F}_{q^2}$, that is, we now get two y-coordinates in $\mathbb{F}_{q^2}$ (but outside $\mathbb{F}_q$). The corresponding points $(r_i, \pm y_i)$ lie in $E(\mathbb{F}_{q^2})$. The right sides of Eqns (3.2) and (3.3) always lie in the group $E(\mathbb{F}_q)$ generated by the base point P. The batch-verification condition demands the sum of $R_1, R_2, \ldots, R_t$ to lie in $E(\mathbb{F}_q)$ in order to pass the test $f_{t+1}(r_1, r_2, \ldots, r_t, \alpha) = 0$ (see Eqn (4.5)). If one or more of the points R_i are defined over $\mathbb{F}_{q^2}$ (but not over $\mathbb{F}_q$), then what is the probability of $\sum_{i=1}^{t} R_i \in E(\mathbb{F}_q)$?

A satisfactory answer to this question can be given if the group structure of $E(\mathbb{F}_{q^2})$ is known to us. $E(\mathbb{F}_q)$ is already a cyclic subgroup of $E(\mathbb{F}_{q^2})$ of large prime order

n. If $E(\mathbb{F}_{q^2})$ is cyclic too, randomly chosen points $R_i \in E(\mathbb{F}_{q^2})$ have a probability of about $1/q$ to have their sum in $E(\mathbb{F}_q)$. Even when $E(\mathbb{F}_{q^2})$ is of rank two with no small-order subgroups (like $\mathbb{Z}_n \oplus \mathbb{Z}_n$), there may be little problem. The use of randomizers makes the probability of $\sum_{i=1}^{t} R_i \in E(\mathbb{F}_q)$ negligible even when the x-coordinates r_i are carefully crafted by the adversary. Only when $E(\mathbb{F}_{q^2})$ contains subgroups of small orders, the adversary may win with non-negligible probability. Randomizers do not seem to help much in this case. Section 4.8 deals with the cases of some of the NIST curves.

In Algorithm 4.1, the sanity check involves the computation of t Legendre symbols $\left(\frac{r_i^3+ar_i+b}{q}\right)$. This is anyway not a huge overhead compared to the computation of f_{t+1} (unless t is very small). Consequently, there is little harm in conducting the sanity check even when the probability of $\sum_{i=1}^{t} R_i \in E(\mathbb{F}_q)$ for points R_i defined over $\mathbb{F}_{q^2}$ is overwhelmingly small.

A sanity check like this is needed for our symbolic-computation Algorithms S2 and S2$'$ too. This issue is discussed in detail in the current chapter.

4.2.6 Cases of Failure of SP

The symbolic-manipulation algorithms of Chapter 3 have a few cases of failure. But the Algorithm SP is robust against most of these failures.

1. Computations which treat y_i as symbols cannot distinguish between the cases of point addition and point doubling. On the contrary, summation polynomials work equally well for both of these operations.

2. Algorithms S1, S2, S1$'$ and S2$'$ fail when the point $R = (\alpha, \beta)$ computed from the right side of Eqn (3.2) or (3.3) is the point $\mathscr{O}$ at infinity. Algorithm SP continues to work. Eqn (4.6) is now rewritten as $(r_1, y_1) + (r_2, y_2) + \cdots + (r_t, y_t) = \mathscr{O}$. That is, instead of computing $f_{t+1}(r_1, r_2, \ldots, r_t, \alpha)$, we now compute $f_t(r_1, r_2, \ldots, r_t)$.

4.3 Algorithm S3

In this section, we propose an alternative method of computing the elliptic-curve summation polynomial using symbolic manipulation. This method is derived from the symbolic addition of elliptic-curve points discussed in Chapter 3 in connection with Algorithm S2′. Let $(x_1,y_1),(x_2,y_2),\ldots,(x_t,y_t)$ be elliptic-curve points corresponding to a batch of t ECDSA signatures. All these points satisfy the elliptic-curve Eqn (4.1). We consider all the y-coordinates as symbols. Now, we compute the symbolic sum $R=\sum_{i=1}^{t}(x_i,y_i)$ using Algorithm 3.3 discussed in Chapter 3. Denote the sum by (X,Y). Then, we can write

$$x(R)=X,\ \text{where } x(R)\in\mathbb{F}_q[y_1,y_2,\ldots,y_t]. \tag{4.11}$$

Eqn (4.11) can now be rewritten as

$$\phi=X-x(R). \tag{4.12}$$

Using Algorithm 4.2, we eliminate all the symbols y_i from the Eqn (4.12) and arrive at a polynomial in X. Let us denote this polynomial by $F_t(X)$. We shortly establish that $F_t(X)$ is essentially the same as the summation polynomial $f_{t+1}(x_1,x_2,\ldots,x_t,X)$.

Algorithm 4.2 Alternative Construction of Summation Polynomial Using Symbolic Manipulation

INPUT: $\phi=X-x(R)$.

OUTPUT: A polynomial $F_t(X)\in\mathbb{F}_q[X]$ with all y_i eliminated from ϕ.

1. For $i=1,2,3,\ldots,t-1$, eliminate y_i from ϕ as in the following steps:
   (a) Write $\phi=v+uy_i$ where u,v are polynomials in $y_{i+1},\ldots,y_t,X$.
   (b) Set $\phi=v^2-u^2(x_i^3+ax_i+b)$.
   (c) Substitute y_j^2 in ϕ by $x_j^3+ax_j+b$ for $j=i+1,\ldots,t$.
2. Set $F_t=\phi$.

Let the right-hand side of Eqn (3.2) or (3.3) be (α,β). In Algorithm S3, we compute the two symbolic sums $R^{(1)}=\sum_{i=1}^{\lceil\frac{t}{2}\rceil}(x_i,y_i)$ and $R^{(2)}=(\alpha,\beta)-$

$\sum_{i=\lceil \frac{t}{2} \rceil+1}^{t}(x_i, y_i)$ using the symbolic elliptic-curve addition formula. The batch-verification condition is $R^{(1)} = R^{(2)}$. We equate the x-coordinates of these two expressions of the same point as

$$x(R^{(1)}) = X, \text{ and } x(R^{(2)}) = X.$$

Eliminating the symbols y_i from the above two equations by Algorithm 4.2, we construct two polynomials $F_{\lceil \frac{t}{2} \rceil}(X)$ and $F_{\lceil \frac{t+1}{2} \rceil}(X)$ in the variable X. The batch-verification condition demands these two polynomials to have a common root (the one corresponding to the correct choices of the square-roots of $r_i^3 + ar_i + b$). Therefore, we compute the resultant of $F_{\lceil \frac{t}{2} \rceil}(X)$ and $F_{\lceil \frac{t+1}{2} \rceil}(X)$ with respect to X. The batch is accepted if and only if the resultant is zero. The steps of the Algorithm S3 are summarized in Algorithm 4.3.

4.3.1 Relation between f_t and F_t

Theorem 4.3.1. *Let $(x_1, y_1), (x_2, y_2), \ldots, (x_t, y_t)$ be t elliptic-curve points. Then, the monic polynomials corresponding to the summation polynomial $f_{t+1}(x_1, x_2, \ldots, x_t, X)$ and the polynomial F_t in X obtained by the elimination method of Algorithm 4.2 are equal. Here, we assume that during the symbolic summation, only the case of point addition occurs (that is, there is no case of point doubling).*

Proof. Semaev in [67] proves that the summation polynomial f_k, $k \geqslant 3$, is of the form

$$f_k(x_1, x_2, \ldots, x_k) = f_{k-1}^2(x_1, x_2, \ldots, x_{k-1}) x_k^{2^{k-2}} + \cdots . \tag{4.13}$$

By Eqn (4.13), the summation polynomial $f_{t+1}(x_1, x_2, \ldots, x_t, X)$ is a polynomial in X of degree 2^{t+1-2}, that is, 2^{t-1}. On the other hand, the computation of $F_t(X)$ by Algorithm 4.2 goes through $(t-1)$ squaring stages. Because ϕ contains only even-degree monomials of $y_1, y_2, \ldots, y_t$ by Theorem 3.2.1, the last two symbols y_{t-1}, y_t are eliminated together. Therefore, F_t also has X-degree 2^{t-1}.

There are 2^t possible combinations of $\sum_{i=1}^{t}(x_i, \pm y_i)$. But the combinations $\sum_{i=1}^{t}(x_i, y_i)$ and $\sum_{i=1}^{t}(x_i, -y_i)$ have the same x-coordinate. This implies that the sum $\sum_{i=1}^{t}(x_i, \pm y_i)$ can have only 2^{t-1} different x-coordinates. These x-coordinates are among the roots of both the polynomials f_{t+1} and F_t. Moreover, since their degrees are exactly equal to 2^{t-1}, these are precisely all the roots of both the polynomials.

Algorithm 4.3 ECDSA Batch-verification Algorithm S3 for NIST Prime Curves

INPUT: Domain Parameters, ECDSA signatures (M_1, r_1, s_1), (M_2, r_2, s_2), ..., (M_t, r_t, s_t) and the public key Q of the signers.

OUTPUT: Accept/Reject the batch of t signatures.

1. Optional sanity check: For each $i = 1, 2, \ldots, t$, check whether $r_i^3 + ar_i + b$ is a quadratic residue modulo q. If not, reject the i-th signature and remove it from the batch. Let us assume that all the signatures in the batch pass the sanity check. (Also see Section 4.2.5.)
2. Compute $w_i = s_i^{-1} \pmod{n}$ for all $i = 1, 2, \ldots, t$.
3. Compute $u_i = H(M_i)w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
4. Compute $v_i = r_i w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
5. Compute $R = (\sum_{i=1}^{t} u_i)P + (\sum_{i=1}^{t} v_i)Q = (\alpha, \beta)$.
6. Compute the symbolic sums $R^{(1)} = \sum_{i=1}^{\lceil \frac{t}{2} \rceil}(x_i, y_i)$ and $R^{(2)} = (\alpha, \beta) - \sum_{i=\lceil \frac{t}{2} \rceil + 1}^{t}(x_i, y_i)$ by Algorithm 3.3.
7. Let $\phi = x(R^{(1)}) - X$. Use Algorithm 4.2 on ϕ to compute $F_{\lceil \frac{t}{2} \rceil}(X) \in \mathbb{F}_q[X]$.
8. Let $\phi = x(R^{(2)}) - X$. Use Algorithm 4.2 on ϕ to compute $F_{\lceil \frac{t+1}{2} \rceil}(X) \in \mathbb{F}_q[X]$.
9. Compute $F_t = \mathrm{Res}_X\left(F_{\lceil \frac{t}{2} \rceil}(X), F_{\lceil \frac{t+1}{2} \rceil}(X)\right)$.
10. Accept the batch of signatures if and only if $F_t = 0$.

Therefore, if we make f_{t+1} and F_t monic, they must be equal because they have the same degree and their sets of roots are equal. □

4.3.2 Analysis of Algorithm S3

Running time complexity

The computation of the symbolic sum of $\frac{t}{2}$ elliptic-curve points requires $\Theta(\sqrt{m}t^2)$ field operations (as derived during the analysis of Algorithms S1 and S2 in Chapter 3).

During the elimination of y_i in Algorithm 4.2, we compute the polynomial

$$v(y_{i+1},\ldots,y_{t/2},X)^2 - u(y_{i+1},\ldots,y_{t/2},X)^2(r_i^3 + ar_i + b)$$

followed by $t/2-i$ substitutions of y_j^2. The coefficients of all the monomials in $y_{i+1},\ldots,y_{t/2}$ in u and v are polynomials in X with degrees $\leqslant 2^{i-1}$. Moreover, there are exactly $2^{t/2-i-1} = \frac{\sqrt{m}}{2^{i+1}}$ monomials in each of u and v. Using fast polynomial multiplication lets us compute u^2 and v^2 using $\frac{\sqrt{m}}{2^{i+1}} \ln \frac{\sqrt{m}}{2^{i+1}} \ln\ln \frac{\sqrt{m}}{2^{i+1}} \times 2^{i-1}\ln 2^{i-1}\ln\ln 2^{i-1} = \mathrm{O}(\sqrt{m}t^2)$ field operations. This square contains $\binom{\frac{\sqrt{m}}{2^{i+1}}}{2}$ monomials containing y_j^2, $j = i+1,\ldots,t/2$. Therefore, $t/2-i$ substitutions of y_j^2s by $(r_i^3 + ar_i + b)$ after the squaring requires $(t/2-i)\binom{\frac{\sqrt{m}}{2^{i+1}}}{2}2^{i-1} = \mathrm{O}(mt/2^i)$ field operations. Then, the total cost of elimination stage is of the order of

$$\sum_{i=1}^{t/2}\left(\sqrt{m}t^2 + \frac{mt}{2^i}\right) \leqslant \sqrt{m}\,t^3 + mt.$$

Since $m = 2^t$, this means $\mathrm{O}(mt)$ field operations. Thus, the computation of the summation polynomials $F_{\lceil \frac{t}{2} \rceil}$ and $F_{\lceil \frac{t+1}{2} \rceil}$ by symbolic manipulation in Algorithm S3 requires $\mathrm{O}(mt)$ field operations. The resultant computation of $F_{\lceil \frac{t}{2} \rceil}$ and $F_{\lceil \frac{t+1}{2} \rceil}$ takes $\mathrm{O}(m)$ field operations. Therefore, the time complexity of Algorithm S3 is dominated by the symbolic computaion of $F_{\lceil \frac{t}{2} \rceil}$ and $F_{\lceil \frac{t+1}{2} \rceil}$, and is $\mathrm{O}(mt)$. It follows that Algorithm SP is a better way to compute summation polynomials than Algorithm S3.

Cases of Failure of S3

1. The derivation of $F_{\lceil \frac{t}{2} \rceil}$ and $F_{\lceil \frac{t+1}{2} \rceil}$ in Algorithms S3 is based upon the point-addition formula on the curve E. If point doubling is ever encountered (which we cannot detect from symbolic sums), then the above polynomials are not correctly computed.

2. Point additions of the form $U + (-U)$ cannot also be handled by the addition formula. So long as we work symbolically using the unknown quantities $y_1, y_2, \ldots, y_t$, it is impossible to predict when the two points being added turn out to be equal or opposite.

4.4 Faster Variants of Algorithms SP and S3

In Algorithm SP, we compute the summation polynomial f_{t+1} for a batch of t ECDSA signatures. If $f_{t+1}(x_1, x_2, \ldots, x_t, \alpha) = 0$, then we accept the batch. We compute f_{t+1} by Eqn (4.4) as

$$f_{t+1}(x_1, x_2, \ldots, x_t, \alpha) = \mathrm{Res}_X(f_{t-k+1}(x_1, \ldots, x_{t-k}, X), f_{k+2}(x_{t-k+1}, \ldots, x_t, \alpha, X))$$

for $t \geqslant 3$ and for any k in the range $1 \leqslant k \leqslant t-2$. In order to compute f_{t+1}, we have computed the resultant of the two summation polynomials $f_{t-k+1}(x_1, \ldots, x_{t-k}, X)$ and $f_{k+2}(x_{t-k+1}, \ldots, x_t, \alpha, X)$. The resultant of these two summation polynomials is zero if and only if they have a common root in the algebraic closure of $\mathbb{F}_q$. Therefore, instead of computing the final resultant, we can compute the gcd of the polynomials $f_{t-k+1}(x_1, \ldots, x_{t-k}, X)$ and $f_{k+2}(x_{t-k+1}, \ldots, x_t, \alpha, X)$, and accept the batch if and only if this gcd is a non-constant polynomial, that is, a polynomial of degree at least one.

Theorem 4.4.1. *Let $f_1 = a_m X^m + a_{m-1} X^{m-1} + \cdots + a_0$ and $f_2 = b_n X^n + b_{n-1} X^{n-1} + \cdots + b_0$ be two polynomials with $a_m, \ldots, a_0, b_n, \ldots, b_0 \in \mathbb{F}_q$ and $a_m \neq 0$, $b_n \neq 0$. Then, the resultant $Res_X(f_1, f_2)$ is zero if and only if f_1 and f_2 have a common root in $\overline{\mathbb{F}}_q$, or equivalently,* $\gcd(f_1, f_2)$ *is a non-constant polynomial.*

Proof. Suppose that the roots of f_1 and f_2 are $\nu_1, \nu_2, \ldots, \nu_m$ and $\eta_1, \eta_2, \ldots, \eta_n$. The resultant of f_1 and f_2 is defined as (see [39]):

$$\mathrm{Res}_X(f_1, f_2) = a_m^n b_n^m \prod_{i=1}^{m} \prod_{j=1}^{n} (\nu_i - \eta_j).$$

It follows that $\mathrm{Res}_X(f_1, f_2) = 0$ if and only if $v_i = \eta_j$ for some i, j, $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$.

Moreover, if $v \in \overline{\mathbb{F}_q}$ is a common root of f_1, f_2, then the minimal polynomial of v over $\mathbb{F}_q$ divides $\gcd(f_1, f_2)$. □

Algorithm $\mathrm{SP}_{\mathrm{gcd}}$ is based upon the computation of $\gcd(f_{k+2}, f_{t-k+1})$. As in Algorithm SP, we first compute the summation polynomials $f_{\lceil \frac{t}{2} \rceil + 1}(x_1, \ldots, x_{\lceil \frac{t}{2} \rceil}, X)$ and $f_{\lceil \frac{t+1}{2} \rceil + 1}(x_{\lceil \frac{t}{2} \rceil + 1}, \ldots, x_t, \alpha, X)$. Algorithm $\mathrm{SP}_{\mathrm{gcd}}$ then computes the gcd of these two polynomials and accepts the batch if and only if this gcd is non-constant.

As we proved earlier, $f_{\lceil \frac{t}{2} \rceil + 1}$ and $f_{\lceil \frac{t+1}{2} \rceil + 1}$ are equal to $F_{\lceil \frac{t}{2} \rceil}$ and $F_{\lceil \frac{t+1}{2} \rceil}$ of Algorithm S3. Thus, Theorem 4.4.1 is equally applicable to Algorithm S3. We can replace the resultant computation at the last stage by a gcd computation.

In Algorithms SP and S3, the last resultant computation is the most time-consuming operation. Although resultants are typically computed by a gcd-like algorithm, the computation involves some additional operations. As a result, we achieve some practical benefits from replacing this resultant computation by a gcd computation. We call these practically faster variants Algorithms $\mathrm{SP}_{\mathrm{gcd}}$ and $\mathrm{S3}_{\mathrm{gcd}}$.

4.4.1 Algorithm $\mathrm{SP}_{\mathrm{gcd}}$

The steps of Algorithm $\mathrm{SP}_{\mathrm{gcd}}$ are summarized in Algorithm 4.4. In Step 7, we replace the last resultant computation by practically faster gcd computation. However, resultant and gcd computations have the same theoretical running times, so the time complexity of Algorithm $\mathrm{SP}_{\mathrm{gcd}}$ is theoretically the same as that of Algorithm SP, that is, $\mathrm{O}(m)$ $\mathbb{F}_q$-operations, where $m = 2^t$. As discussed earlier, this gcd variant only provides some practical benefits.

4.4.2 Algorithm $\mathrm{S3}_{\mathrm{gcd}}$

Algorithm $\mathrm{S3}_{\mathrm{gcd}}$ combines the concept of summation-polynomial computation using symbolic manipulations and a gcd computation to check the existence of common

Algorithm 4.4 ECDSA Batch-verification Algorithm $\mathrm{SP}_{\mathrm{gcd}}$ for NIST Prime Curves

INPUT: Domain Parameters, ECDSA signatures (M_1, r_1, s_1), (M_2, r_2, s_2), $\ldots$, (M_t, r_t, s_t) and public key Q of the signer.

OUTPUT: Accept/Reject the batch of t signatures.

1. Optional sanity check: For each $i = 1, 2, \ldots, t$, check whether $r_i^3 + ar_i + b$ is a quadratic residue modulo q. If not, reject the i-th signature and remove it from the batch. Let us assume that all the signatures in the batch pass the sanity check. (Also see Section 4.2.5.)
2. Compute $w_i = s_i^{-1} \pmod{n}$ for all $i = 1, 2, \ldots, t$.
3. Compute $u_i = H(M_i)w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
4. Compute $v_i = r_i w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
5. Compute $R = (\sum_{i=1}^{t} u_i)P + (\sum_{i=1}^{t} v_i)Q = (\alpha, \beta)$.
6. Compute the summation polynomials $f_{\lceil \frac{t}{2} \rceil + 1}(r_1, r_2, \ldots, r_{\lceil \frac{t}{2} \rceil}, X)$ and $f_{\lceil \frac{t+1}{2} \rceil + 1}(r_{\lceil \frac{t}{2} \rceil + 1}, \ldots, r_t, \alpha, X)$ in $\mathbb{F}_q[X]$.
7. Compute $d(X) = \gcd\left(f_{\lceil \frac{t}{2} \rceil + 1}, f_{\lceil \frac{t+1}{2} \rceil + 1}\right) \in \mathbb{F}_q[X]$.
8. Accept the batch of signatures if and only if $\deg d(X) \geqslant 1$.

Algorithm 4.5 ECDSA Batch-verification Algorithm $S3_{\text{gcd}}$ for NIST Prime Curves

INPUT: Domain Parameters, ECDSA signatures (M_1, r_1, s_1), (M_2, r_2, s_2), $\ldots$, (M_t, r_t, s_t) and public key Q of the signer.

OUTPUT: Accept/Reject the batch of t signatures.

1. Optional sanity check: For each $i = 1, 2, \ldots, t$, check whether $r_i^3 + ar_i + b$ is a quadratic residue modulo q. If not, reject the i-th signature and remove it from the batch. Let us assume that all the signatures in the batch pass the sanity check. (Also see Section 4.2.5.)
2. Compute $w_i = s_i^{-1} \pmod{n}$ for all $i = 1, 2, \ldots, t$.
3. Compute $u_i = H(M_i)w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
4. Compute $v_i = r_i w_i \pmod{n}$ for all $i = 1, 2, \ldots, t$.
5. Compute $R = (\sum_{i=1}^{t} u_i)P + (\sum_{i=1}^{t} v_i)Q = (\alpha, \beta)$.
6. Compute the symbolic sums $R^{(1)} = \sum_{i=1}^{\lfloor \frac{t}{2} \rfloor} (x_i, y_i)$ and $R^{(2)} = (\alpha, \beta) - \sum_{i=\lfloor \frac{t}{2} \rfloor + 1}^{t} (x_i, y_i)$ by Algorithm 3.3.
7. Let $\phi = x(R^{(1)}) - X$. Use Algorithm 4.2 on ϕ to compute $F_{\lceil \frac{t}{2} \rceil}(X) \in \mathbb{F}_q[X]$.
8. Let $\phi = x(R^{(2)}) - X$. Use Algorithm 4.2 on ϕ to compute $F_{\lceil \frac{t+1}{2} \rceil}(X) \in \mathbb{F}_q[X]$.
9. Compute $d(X) = \gcd\left(F_{\lceil \frac{t}{2} \rceil}(X), F_{\lceil \frac{t+1}{2} \rceil}(X)\right) \in \mathbb{F}_q[X]$.
10. Accept the batch of signatures if and only if $\deg d(X) \geqslant 1$.

roots. We give the detailed description of this method as Algorithm 4.5. Since the resultant computation and the gcd computation have the same theoretical running times, the running time of Algorithm $S3_{gcd}$ is the same as the running time of Algorithm S3, that is, $O(mt)$ $\mathbb{F}_q$-operations, where $m = 2^t$.

4.5 Experimental Results

All experiments are carried out in the same environment as described in Section 3.5 of Chapter 3. We have used the symbolic-computation facilities of the GP/PARI calculator in our programs. All other functions (like scalar multiplication and square-root computation) are written as subroutines in which function-call overheads are minimized as much as possible. We only used the built-in field arithmetic provided by the calculator. Since all algorithms are evaluated in terms of number of field operations, this gives a fare comparison of experimental data with the theoretical estimates.

Table 4.1 lists the overheads associated with the ECDSA batch-verification algorithms introduced in this chapter. The speedup figures (over individual verification) are listed in Tables 4.2–4.5. For ease of comparison, these tables include the speedup factors obtained by the batch-verification algorithms introduced in Chapter 3. Tables 4.2 and 4.3 deal with the case where all the signatures belong to the same signer. Tables 4.4 and 4.5 include the speedup figures obtained when all the signatures are from different signers. In all these tables, we do not prepare or use the precomputation tables for $2^iP, 2^iQ, 2^i(P+Q)$ during individual verification.

The experimental results clearly indicate that SP_{gcd} is the most efficient batch-verification algorithm for standard ECDSA signatures. Even for ECDSA$^{\#}$ signatures, Algorithm SP_{gcd} often outperforms the naive method N'. The optimal batch size for Algorithm $S2'$ is $t = 7$ for prime curves. With Algorithm SP_{gcd}, the optimal batch size is $t = 9$ for prime curves. The maximum speedup is noticeably higher in Algorithm SP_{gcd} than what is achieved by Algorithm $S2'$.

Now, suppose that we can afford the huge storage overhead associated with the precomputed tables in fixed-base double scalar multiplication. Tables 4.6 and 4.7

Table 4.1: Overheads (ms) for different batch verification algorithms

	Algorithm S3						Algorithm $S3_{gcd}$					
	t						*t*					
Curve	3	4	5	6	7	8	3	4	5	6	7	8
P-192	0.08	0.20	0.41	0.67	1.30	2.16	0.07	0.18	0.34	0.54	0.93	1.62
P-224	0.08	0.21	0.44	0.72	1.39	2.31	0.08	0.19	0.37	0.59	1.01	1.75
P-256	0.09	0.23	0.47	0.74	1.43	2.34	0.09	0.20	0.38	0.60	1.02	1.76
P-384	0.13	0.30	0.59	0.92	1.77	2.83	0.13	0.27	0.47	0.74	1.23	2.11
P-521	0.17	0.38	0.73	1.14	2.20	3.52	0.17	0.35	0.60	0.93	1.52	2.65

	Algorithm SP								
	t								
Curve	2	3	4	5	6	7	8	9	10
P-192	0.03	0.06	0.12	0.21	0.43	0.79	1.50	2.76	5.54
P-224	0.04	0.07	0.14	0.24	0.50	0.91	1.71	3.19	6.51
P-256	0.04	0.12	0.15	0.25	0.51	0.94	1.76	3.30	6.49
P-384	0.06	0.11	0.20	0.34	0.70	1.32	2.43	4.62	8.91
P-521	0.08	0.15	0.27	0.46	0.95	1.78	3.38	6.56	12.69

	Algorithm SP_{gcd}								
	t								
Curve	2	3	4	5	6	7	8	9	10
P-192	0.03	0.06	0.11	0.19	0.41	0.65	1.36	2.11	4.90
P-224	0.04	0.07	0.14	0.21	0.46	0.74	1.60	2.50	5.76
P-256	0.04	0.12	0.14	0.22	0.48	0.77	1.58	2.47	5.63
P-384	0.06	0.11	0.20	0.30	0.65	1.05	2.13	3.33	7.61
P-521	0.08	0.16	0.27	0.41	0.89	1.43	3.01	4.73	10.82

Table 4.2: Speedup obtained by different batch-verification algorithms where all the signatures are from the same signer

(Fixed-base scalar multiplication *not* used during individual verification)

(Experiment not carried out for cells marked as –)

Curve	t	Same signer						
		N	N′	S2′	S3	$S3_{gcd}$	SP	SP_{gcd}
P-192	2	1.80	1.83	–	–	–	1.98	1.97
	3	2.47	2.63	2.95	2.93	2.94	2.95	3.00
	4	2.87	3.36	3.82	3.78	3.80	3.86	3.87
	5	2.92	4.04	4.53	4.46	4.54	4.71	4.73
	6	2.45	4.67	5.05	5.01	5.18	5.32	5.35
	7	1.70	5.26	4.95	5.06	5.49	5.68	5.87
	8	1.04	5.81	4.03	4.89	5.41	5.55	5.71
	9	–	–	–	–	–	4.96	5.55
	10	–	–	–	–	–	3.80	4.09
P-224	2	0.56	0.56	–	–	–	1.98	1.98
	3	0.61	0.62	2.95	2.94	2.94	2.95	2.95
	4	0.63	0.65	3.85	3.81	3.83	3.87	3.87
	5	0.64	0.67	4.59	4.53	4.60	4.73	4.76
	6	0.62	0.69	5.16	5.13	5.26	5.36	5.41
	7	0.56	0.70	5.16	5.27	5.65	5.76	5.96
	8	0.48	0.71	4.31	5.17	5.65	5.69	5.80
	9	–	–	–	–	–	5.13	5.65
	10	–	–	–	–	–	3.93	4.23
P-256	2	1.86	1.88	–	–	–	1.98	1.98
	3	2.62	2.73	2.96	2.95	2.95	2.93	2.93
	4	3.15	3.54	3.87	3.83	3.85	3.89	3.90
	5	3.35	4.30	4.66	4.59	4.66	4.77	4.80
	6	3.00	5.01	5.30	5.26	5.39	5.47	5.50
	7	2.23	5.70	5.45	5.51	5.86	5.94	6.11
	8	1.43	6.35	4.76	5.54	6.00	6.00	6.15
	9	–	–	–	–	–	5.53	6.13
	10	–	–	–	–	–	4.48	4.83

Table 4.3: Speedup obtained by different batch-verification algorithms where all the signatures are from the same signer

(Fixed-base scalar multiplication *not* used during individual verification)

(Experiment not carried out for cells marked as –)

		Same signer						
Curve	t	N	N′	S2′	S3	$S3_{gcd}$	SP	SP_{gcd}
P-384	2	1.86	1.87	–	–	–	1.99	1.99
	3	2.65	2.72	2.97	2.96	2.96	2.97	2.97
	4	3.27	3.52	3.90	3.88	3.89	3.92	3.92
	5	3.63	4.27	4.77	4.72	4.77	4.83	4.85
	6	3.46	4.98	5.52	5.48	5.58	5.60	5.63
	7	2.85	5.65	5.92	5.93	6.22	6.17	6.32
	8	1.99	6.28	5.57	6.20	6.58	6.41	6.57
	9	–	–	–	–	–	6.11	6.71
	10	–	–	–	–	–	5.23	5.62
P-521	2	1.86	1.88	–	–	–	1.99	1.99
	3	2.68	2.73	2.97	2.97	2.97	2.97	2.97
	4	3.34	3.53	3.92	3.91	3.92	3.93	3.93
	5	3.78	4.29	4.82	4.78	4.82	4.86	4.88
	6	3.80	5.00	5.63	5.60	5.67	5.67	5.69
	7	3.28	5.68	6.16	6.16	6.40	6.31	6.43
	8	2.41	6.32	5.99	6.57	6.87	6.62	6.74
	9	–	–	–	–	–	6.40	6.96
	10	–	–	–	–	–	5.60	5.99

Table 4.4: Speedup obtained by different batch-verification algorithms where all the signatures are from different signers

(Fixed-base scalar multiplication *not* used during individual verification)

(Experiment not carried out for cells marked as –)

Curve	t	Different signer						
		N	N′	S2′	S3	$S3_{gcd}$	SP	SP_{gcd}
P-192	2	0.97	0.98	–	–	–	1.02	1.02
	3	1.07	1.10	1.15	1.15	1.15	1.15	1.16
	4	1.10	1.16	1.22	1.21	1.21	1.22	1.22
	5	1.09	1.21	1.25	1.25	1.25	1.26	1.27
	6	1.00	1.24	1.27	1.27	1.28	1.28	1.29
	7	0.84	1.27	1.25	1.26	1.28	1.29	1.30
	8	0.64	1.29	1.17	1.23	1.27	1.27	1.28
	9	–	–	–	–	–	1.23	1.27
	10	–	–	–	–	–	1.14	1.16
P-224	2	0.56	0.56	–	–	–	2.05	2.05
	3	0.57	0.58	2.30	2.29	2.29	2.30	2.30
	4	0.58	0.59	2.42	2.41	2.41	2.43	2.43
	5	0.57	0.60	2.47	2.45	2.47	2.51	2.52
	6	0.54	0.60	2.48	2.47	2.50	2.53	2.54
	7	0.50	0.60	2.38	2.41	2.48	2.51	2.54
	8	0.43	0.61	2.13	2.32	2.41	2.42	2.44
	9	–	–	–	–	–	2.26	2.36
	10	–	–	–	–	–	1.97	2.04
P-256	2	1.09	1.10	–	–	–	1.14	1.14
	3	1.21	1.23	1.28	1.27	1.27	1.27	1.27
	4	1.25	1.31	1.35	1.35	1.35	1.36	1.36
	5	1.25	1.36	1.40	1.39	1.40	1.41	1.41
	6	1.18	1.40	1.42	1.42	1.43	1.43	1.43
	7	1.03	1.43	1.41	1.42	1.44	1.44	1.45
	8	0.81	1.45	1.35	1.40	1.43	1.43	1.44
	9	–	–	–	–	–	1.39	1.43
	10	–	–	–	–	–	1.31	1.33

Table 4.5: Speedup obtained by different batch-verification algorithms where all the signatures are from different signers

(Fixed-base scalar multiplication *not* used during individual verification)

(Experiment not carried out for cells marked as –)

		Different signer						
Curve	t	N	N′	S2′	S3	$S3_{gcd}$	SP	SP_{gcd}
P-384	2	1.17	1.17	–	–	–	1.22	1.22
	3	1.30	1.31	1.37	1.37	1.37	1.37	1.37
	4	1.36	1.40	1.45	1.45	1.45	1.46	1.46
	5	1.37	1.45	1.51	1.50	1.51	1.51	1.51
	6	1.32	1.49	1.54	1.53	1.54	1.54	1.55
	7	1.20	1.52	1.54	1.54	1.56	1.56	1.57
	8	1.01	1.55	1.50	1.54	1.56	1.55	1.56
	9	–	–	–	–	–	1.52	1.55
	10	–	–	–	–	–	1.45	1.48
P-521	2	1.14	1.14	–	–	–	1.18	1.18
	3	1.27	1.28	1.33	1.33	1.33	1.33	1.33
	4	1.33	1.36	1.41	1.41	1.41	1.41	1.41
	5	1.35	1.41	1.47	1.46	1.47	1.47	1.47
	6	1.33	1.45	1.50	1.50	1.50	1.50	1.50
	7	1.24	1.48	1.51	1.51	1.52	1.52	1.53
	8	1.08	1.50	1.48	1.52	1.53	1.52	1.52
	9	–	–	–	–	–	1.49	1.52
	10	–	–	–	–	–	1.43	1.46

Table 4.6: Speedup obtained by different batch-verification algorithms
(Fixed-base scalar multiplication *used* during individual verification)
(Experiment not carried out for cells marked as –)

Curve	t	Same signer						
		N	N′	S2′	S3	$S3_{gcd}$	SP	SP_{gcd}
P-192	2	1.97	2.00	–	–	–	2.17	2.15
	3	2.25	2.39	2.68	2.67	2.68	2.68	2.73
	4	2.34	2.74	3.12	3.08	3.10	3.15	3.16
	5	2.23	3.07	3.45	3.40	3.46	3.59	3.61
	6	1.77	3.38	3.66	3.63	3.75	3.86	3.88
	7	1.19	3.67	3.46	3.53	3.84	3.96	4.10
	8	0.70	3.94	2.73	3.31	3.67	3.76	3.87
	9	–	–	–	–	–	3.29	3.67
	10	–	–	–	–	–	2.47	2.66
P-224	2	0.61	0.61	–	–	–	2.16	2.16
	3	0.55	0.55	2.65	2.64	2.64	2.65	2.65
	4	0.51	0.52	3.09	3.06	3.07	3.11	3.11
	5	0.47	0.50	3.42	3.38	3.43	3.53	3.56
	6	0.44	0.49	3.65	3.63	3.73	3.80	3.83
	7	0.38	0.48	3.51	3.59	3.85	3.92	4.06
	8	0.31	0.47	2.84	3.42	3.74	3.76	3.83
	9	–	–	–	–	–	3.31	3.65
	10	–	–	–	–	–	2.49	2.67
P-256	2	1.96	1.98	–	–	–	2.10	2.10
	3	2.29	2.39	2.58	2.58	2.58	2.56	2.56
	4	2.47	2.77	3.03	3.00	3.02	3.05	3.05
	5	2.44	3.13	3.39	3.34	3.40	3.48	3.50
	6	2.07	3.47	3.66	3.64	3.73	3.78	3.80
	7	1.49	3.80	3.63	3.67	3.91	3.96	4.07
	8	0.92	4.10	3.08	3.58	3.88	3.88	3.98
	9	–	–	–	–	–	3.49	3.87
	10	–	–	–	–	–	2.77	2.99

Table 4.7: Speedup obtained by different batch-verification algorithms
(Fixed-base scalar multiplication *used* during individual verification)
(Experiment not carried out for cells marked as –)

Curve	t	Same signer						
		N	N′	S2′	S3	$S3_{gcd}$	SP	SP_{gcd}
P-384	2	1.99	2.00	–	–	–	2.13	2.13
	3	2.33	2.40	2.61	2.61	2.61	2.61	2.61
	4	2.57	2.76	3.07	3.05	3.06	3.08	3.08
	5	2.65	3.11	3.47	3.44	3.48	3.52	3.54
	6	2.39	3.44	3.81	3.79	3.85	3.87	3.89
	7	1.90	3.75	3.93	3.94	4.13	4.10	4.20
	8	1.28	4.04	3.59	3.99	4.24	4.12	4.23
	9	–	–	–	–	–	3.84	4.21
	10	–	–	–	–	–	3.22	3.46
P-521	2	1.88	1.89	–	–	–	2.01	2.01
	3	2.30	2.34	2.55	2.55	2.55	2.55	2.55
	4	2.62	2.77	3.07	3.06	3.07	3.08	3.08
	5	2.76	3.13	3.52	3.49	3.52	3.55	3.56
	6	2.63	3.45	3.89	3.87	3.92	3.91	3.93
	7	2.17	3.76	4.08	4.08	4.24	4.18	4.26
	8	1.54	4.05	3.84	4.21	4.41	4.24	4.33
	9	–	–	–	–	–	4.00	4.35
	10	–	–	–	–	–	3.43	3.66

include the speedup figures over this improved individual verification. As discussed in Chapter 3, the case of different signers is not effective for fixed-base double scalar multiplication. Although individual verification receives a significant boost from the precomputed tables, particularly for large batch sizes, batch verification based on elliptic-curve summation polynomials is still capable of producing speedup figures above four.

4.6 Adaptation of Algorithms SP and S3 to Koblitz Curves

Let E be a Koblitz curve defined over a binary field $\mathbb{F}_{2^d}$ by the equation

$$E : y^2 + xy = x^3 + ax^2 + 1, \text{ where } a \in \{0,1\}. \tag{4.14}$$

Let n be the order of the group we work in, and $\hat{h} = |E(\mathbb{F}_{2^d})|/n$ the cofactor. For Koblitz curves, $\hat{h} = 2$ or 4. Because $\hat{h}$ is small, appending a few extra bit(s) to ECDSA signatures, we can uniquely retrieve the x-coordinates from the published value of r in a signature. We therefore assume that the x-coordinates are known to us. We denote these x-coordinates by r_i itself. We can apply our batch-verification Algorithms SP and S3 *mutatis mutandis* to Koblitz curves.

4.6.1 Summation Polynomials for Koblitz Curves

Here, we only supply the first three base cases of summation polynomials f_2, f_3, f_4. The recurrence relation for Koblitz-curve summation polynomials f_t with $t \geqslant 5$ is identical to the case of prime curves (see Eqn (4.4)).

$$\begin{aligned}
f_2(x_1,x_2) &= x_1 + x_2, \\
f_3(x_1,x_2,x_3) &= (x_1x_2 + x_1x_3 + x_2x_3)^2 + x_1x_2x_3 + 1, \\
f_4(x_1,x_2,x_3,x_4) &= (x_1+x_2+x_3+x_4)^4 + (x_1x_2x_3+x_1x_2x_4+x_1x_3x_4+x_2x_3x_4)^4 + \\
&\quad x_1x_2x_3x_4(x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3x_4 + x_1 + x_2 + x_3 + x_4)^2 + \\
&\quad (x_1x_2x_3x_4)^2(x_1 + x_2 + x_3 + x_4)^2 + (x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3x_4)^2.
\end{aligned}$$

Eqn (4.5) holds for Koblitz curve too, that is, there exist $y_1, y_2, \ldots, y_t \in \mathbb{F}_{2^d}$ with each (x_i, y_i) satisfying Eqn (4.14) if and only if $f_t(x_1, x_2, \ldots, x_t) = 0$.

For prime curves, we always reduce the recursion to the computation of f_3, since the explicit formula for f_4 is rather clumsy. For Koblitz curves, we use both the cases f_3 and f_4 as those that terminate recursion. This helps us to reduce the height of the recursion tree for most of the batch sizes.

Notice that all batch-verification algorithms for Koblitz curves can be readily adapted to other ordinary (non-supersingular) curves over binary fields. We deal with the NIST family of Koblitz curves as an illustrative sample.

4.6.2 Computation of Summation Polynomials using Symbolic Manipulation

The method is similar to the case of prime curves. Only the elimination procedure has to be modified for Koblitz curves. Let $(x_1,y_1),(x_2,y_2),\ldots,(x_t,y_t)$ be t points on the Koblitz curve. First, we compute the symbolic sum $R=\sum_{i=1}^{t}(x_i,y_i)$. Let the sum be (X,Y). We then write

$$x(R)=X,\ \text{where } x(R)\in\mathbb{F}_{2^d}[y_1,y_2,\ldots,y_t]. \tag{4.15}$$

Eqn (4.15) can be written as

$$\phi=X-x(R). \tag{4.16}$$

Using Algorithm 4.6, we eliminate all the symbols y_i from Eqn (4.16) and arrive at the summation polynomial $F_t(X)=f_{t+1}(x_1,x_2,\ldots,x_t,X)$ corresponding to the elliptic-curve points $(x_1,y_1),(x_2,y_2),\ldots,(x_t,y_t)$.

4.6.3 Adaptation of the Sanity Check

The sanity check (the equivalent of Step 1 in Algorithm 4.1) is quite easy in the context of NIST Koblitz curves. In order that the point $R_i=(r_i,y_i)$ is defined over $\mathbb{F}_{2^d}$, we now need the equation $y_i^2+r_iy_i+(r_i^3+ar_i^2+1)=0$ to be solvable (for y_i) in $\mathbb{F}_{2^d}$. This is equivalent to the condition that the trace of $\frac{r_i^3+ar_i^2+1}{r_i^2}$ over $\mathbb{F}_{2^d}$ is zero. Let the field $\mathbb{F}_{2^d}=\mathbb{F}_2[X]/\langle F(X)\rangle$ be defined by the irreducible polynomial $F(X)=X^d+a_{d-1}X^{d-1}+\cdots+a_1X+a_0$, where $a_i\in\mathbb{F}_2$. Let $\theta\in\mathbb{F}_{2^d}$ be a root of $F(X)$. Any element $c\in\mathbb{F}_{2^d}$ can be represented as $c=\sum_{k=0}^{d-1}c_k\theta^k$. We can compute

Algorithm 4.6 Alternative Construction of Summation Polynomials using Symbolic Manipulation

INPUT: $\phi = X - x(R)$.

OUTPUT: Summation polynomial $F_t(X) = f_{t+1}(x_1, x_2, \ldots, x_t, X)$ corresponding to the points $(x_1, y_1), (x_2, y_2), \ldots, (x_t, y_t)$.

1. For $i = 1, 2, 3, \ldots, t-1$, eliminate y_i from ϕ as in the following steps:

   (a) Write $\phi = v + uy_i$ where u, v are polynomials in $y_{i+1}, \ldots, y_t$.

   (b) Set $\phi = (v + uy_i)(v + u(y_i + x_i)) = v^2 + uvx_i + u^2(x_i^3 + ax_i^2 + b)$.

   (c) Substitute y_j^2 in ϕ by $x_j^3 + ax_j^2 + b$ for $j = i+1, \ldots, t$.

2. Set $F_t = \phi$.

the trace of c as $\mathrm{Tr}(c) = c_0 + \sum_{k=1}^{d-1} kc_k a_{d-k}$ (see [16] for a discussion). For NIST Koblitz curves, the defining polynomial $F(X)$ has very few non-zero coefficients, so the computation of $\mathrm{Tr}(c)$ is essentially a constant-time effort given any $c \in \mathbb{F}_{2^d}$.

Even when the solutions for y_i lie in $\mathbb{F}_{2^d}$, there is no guarantee that the point $R_i = (r_i, y_i)$ belongs to the subgroup of $E(\mathbb{F}_{2^d})$ generated by the base point P, since Koblitz curves have cofactors $\hat{h} > 1$. At present, we do not know any efficient solution of this problem. If $E(\mathbb{F}_{2^d})$ is cyclic, then R_i is in the subgroup generated by P if and only if $nR_i = \mathcal{O}$. However, computing the scalar multiplication nR_i for each i lets us forfeit the speedup obtained by batch verification.

4.7 Experimental Results

We continue our experiments in the same environment as that for prime curves. The GP/PARI calculator is much slower for binary fields than for prime fields. However, this speed difference is not very significant for the experimental speedup figures which are ratios.

Table 4.8 lists the overheads associated with different ECDSA batch-verification algorithms introduced in this chapter. The speedup figures (over individual

Table 4.8: Overheads (ms) for different batch verification algorithms

	Algorithm S3					
	t					
Curve	2	3	4	5	6	7
K-163	6.36	15.08	32.91	80.56	148.37	317.00
K-233	10.65	26.18	57.02	137.82	248.40	525.40
K-283	12.00	35.67	77.16	185.06	325.26	692.94
K-409	28.00	66.88	143.56	338.52	621.18	1273.32
K-571	47.59	120.46	255.86	598.58	1072.00	2223.01

	Algorithm $S3_{gcd}$					
	t					
Curve	2	3	4	5	6	7
K-163	6.16	23.02	35.48	83.59	149.49	317.00
K-233	10.42	42.12	61.80	143.08	265.82	535.88
K-283	16.00	60.39	84.21	194.17	333.99	709.36
K-409	26.17	116.94	157.64	358.70	628.41	1310.68
K-571	46.68	217.92	284.80	642.77	1108.00	2310.75

	Algorithm SP								
	t								
Curve	2	3	4	5	6	7	8	9	10
K-163	2.44	4.49	17.31	29.07	61.64	268.63	333.49	630.57	2034.46
K-233	3.82	7.10	28.39	51.49	103.15	445.44	557.20	1062.03	2470.94
K-283	4.93	9.29	37.94	70.55	139.29	596.38	750.44	1425.83	3287.13
K-409	8.55	16.20	67.74	134.67	250.45	1052.69	1334.87	2548.02	5692.80
K-571	14.04	26.95	116.46	244.11	437.98	1817.06	2318.83	4365.29	9653.16

	Algorithm SP_{gcd}								
	t								
Curve	2	3	4	5	6	7	8	9	10
K-163	2.45	4.50	8.99	13.55	41.41	233.62	272.32	478.50	1789.05
K-233	3.81	7.10	14.12	21.57	65.80	383.35	449.84	780.91	2461.89
K-283	4.94	9.36	18.24	27.12	86.14	510.90	596.30	1041.63	3255.48
K-409	8.61	16.18	31.43	48.24	146.58	898.51	1052.50	1832.59	5679.54
K-571	14.02	27.06	51.96	78.71	242.11	1551.94	1798.60	3136.67	9665.94

Table 4.9: Speedup obtained by different batch-verification algorithms where all the signatures are from the same signer

(Fixed-base scalar multiplication *not* used during individual verification)

(Experiment not carried out for cells marked as –)

		Same signer						
Curve	t	N	N′	S2′	S3	$S3_{gcd}$	SP	SP_{gcd}
K-163	2	1.79	1.88	1.95	1.91	1.91	1.97	1.97
	3	2.23	2.71	2.82	2.70	2.57	2.91	2.91
	4	2.13	3.49	3.48	3.23	3.18	3.55	3.75
	5	1.58	4.22	3.31	3.15	3.11	4.13	4.55
	6	0.98	4.89	2.81	2.89	2.88	4.14	4.61
	7	0.53	5.53	1.50	2.12	2.12	2.37	2.60
	8	–	–	–	–	–	2.34	2.69
	9	–	–	–	–	–	1.61	2.01
	10	–	–	–	–	–	0.63	0.71
K-233	2	1.83	1.90	1.96	1.93	1.93	1.97	1.97
	3	2.37	2.76	2.87	2.74	2.61	2.93	2.93
	4	2.40	3.58	3.59	3.32	3.27	3.63	3.81
	5	1.89	4.36	3.61	3.35	3.30	4.22	4.64
	6	1.21	5.09	3.22	3.17	3.07	4.38	4.85
	7	0.68	5.78	1.84	2.43	2.40	2.69	2.95
	8	–	–	–	–	–	2.67	3.06
	9	–	–	–	–	–	1.87	2.37
	10	–	–	–	–	–	1.01	1.02
K-283	2	1.85	1.90	1.97	1.95	1.93	1.98	1.98
	3	2.44	2.78	2.89	2.77	2.63	2.94	2.94
	4	2.55	3.61	3.66	3.39	3.35	3.68	3.84
	5	2.09	4.39	3.78	3.50	3.45	4.30	4.70
	6	1.39	5.14	3.48	3.42	3.38	4.54	5.00
	7	0.79	5.85	2.09	2.69	2.65	2.94	3.21
	8	–	–	–	–	–	2.92	3.36
	9	–	–	–	–	–	2.09	2.64
	10	–	–	–	–	–	1.16	1.17

Table 4.10: Speedup obtained by different batch-verification algorithms where all the signatures are from the same signer

(Fixed-base scalar multiplication *not* used during individual verification)

(Experiment not carried out for cells marked as –)

Curve	t	Same signer						
		N	N′	S2′	S3	$S3_{gcd}$	SP	SP_{gcd}
K-409	2	1.88	1.93	1.98	1.95	1.95	1.98	1.98
	3	2.57	2.83	2.92	2.81	2.69	2.95	2.95
	4	2.84	3.70	3.75	3.50	3.46	3.75	3.88
	5	2.50	4.53	4.07	3.75	3.69	4.41	4.77
	6	1.78	5.33	3.96	3.72	3.70	4.81	5.24
	7	1.05	6.09	2.61	3.10	3.05	3.43	3.71
	8	–	–	–	–	–	3.45	3.92
	9	–	–	–	–	–	2.56	3.20
	10	–	–	–	–	–	1.51	1.51
K-571	2	1.90	1.93	1.98	1.96	1.96	1.99	1.99
	3	2.64	2.84	2.94	2.84	2.72	2.96	2.96
	4	3.01	3.72	3.80	3.57	3.53	3.79	3.91
	5	2.80	4.57	4.25	3.91	3.85	4.49	4.82
	6	2.10	5.38	4.28	4.00	3.96	4.98	5.39
	7	1.31	6.17	3.05	3.44	3.37	3.79	4.06
	8	–	–	–	–	–	3.84	4.35
	9	–	–	–	–	–	2.96	3.65
	10	–	–	–	–	–	1.82	1.82

Table 4.11: Speedup obtained by different batch-verification algorithms where all the signatures are from different signers

(Fixed-base scalar multiplication *not* used during individual verification)

(Experiment not carried out for cells marked as –)

Curve	t	Different signer						
		N	N′	S2′	S3	$S3_{gcd}$	SP	SP_{gcd}
K-163	2	1.24	1.28	1.31	1.29	1.29	1.32	1.32
	3	1.28	1.42	1.45	1.42	1.38	1.48	1.48
	4	1.18	1.51	1.51	1.46	1.45	1.52	1.56
	5	0.97	1.57	1.42	1.39	1.39	1.56	1.61
	6	0.70	1.61	1.29	1.31	1.31	1.52	1.58
	7	0.43	1.64	0.91	1.11	1.11	1.18	1.23
	8	–	–	–	–	–	1.16	1.23
	9	–	–	–	–	–	0.94	1.06
	10	–	–	–	–	–	0.49	0.54
K-233	2	1.25	1.29	1.32	1.30	1.30	1.32	1.32
	3	1.32	1.44	1.47	1.43	1.39	1.48	1.48
	4	1.26	1.53	1.53	1.48	1.47	1.54	1.57
	5	1.08	1.59	1.48	1.43	1.42	1.57	1.62
	6	0.80	1.63	1.38	1.37	1.35	1.55	1.61
	7	0.53	1.66	1.03	1.19	1.18	1.25	1.30
	8	–	–	–	–	–	1.23	1.31
	9	–	–	–	–	–	1.02	1.15
	10	–	–	–	–	–	0.70	0.70
K-283	2	1.26	1.29	1.32	1.31	1.30	1.32	1.32
	3	1.35	1.44	1.47	1.44	1.40	1.48	1.48
	4	1.30	1.53	1.54	1.49	1.48	1.55	1.57
	5	1.14	1.59	1.51	1.46	1.45	1.58	1.63
	6	0.88	1.64	1.42	1.41	1.40	1.57	1.62
	7	0.59	1.67	1.10	1.25	1.24	1.30	1.35
	8	–	–	–	–	–	1.28	1.36
	9	–	–	–	–	–	1.08	1.21
	10	–	–	–	–	–	0.76	0.77

Table 4.12: Speedup obtained by different batch-verification algorithms where all the signatures are from different signers

(Fixed-base scalar multiplication *not* used during individual verification)

(Experiment not carried out for cells marked as –)

		Different signer						
Curve	t	N	N′	S2′	S3	$S3_{gcd}$	SP	SP_{gcd}
K-409	2	1.28	1.30	1.32	1.31	1.31	1.33	1.33
	3	1.38	1.46	1.48	1.45	1.42	1.49	1.49
	4	1.37	1.55	1.56	1.51	1.51	1.56	1.58
	5	1.25	1.61	1.55	1.50	1.49	1.60	1.64
	6	1.02	1.65	1.49	1.46	1.46	1.60	1.65
	7	0.73	1.69	1.23	1.33	1.32	1.39	1.43
	8	–	–	–	–	–	1.38	1.44
	9	–	–	–	–	–	1.20	1.32
	10	–	–	–	–	–	0.90	0.90
K-571	2	1.29	1.30	1.33	1.31	1.31	1.33	1.33
	3	1.40	1.46	1.48	1.46	1.43	1.49	1.49
	4	1.41	1.55	1.57	1.53	1.52	1.57	1.58
	5	1.32	1.62	1.57	1.52	1.52	1.61	1.65
	6	1.12	1.66	1.54	1.50	1.49	1.62	1.66
	7	0.84	1.69	1.32	1.39	1.38	1.44	1.48
	8	–	–	–	–	–	1.43	1.50
	9	–	–	–	–	–	1.28	1.39
	10	–	–	–	–	–	1.00	1.00

Table 4.13: Speedup obtained by different batch-verification algorithms
(Fixed-base scalar multiplication *used* during individual verification)
(Experiment not carried out for cells marked as –)

		Same signer						
Curve	t	N	N′	S2′	S3	$S3_{gcd}$	SP	SP_{gcd}
K-163	2	3.88	4.07	4.22	4.15	4.16	4.27	4.27
	3	3.97	4.81	5.01	4.81	4.57	5.16	5.16
	4	3.36	5.51	5.50	5.10	5.03	5.62	5.93
	5	2.32	6.16	4.84	4.61	4.55	6.04	6.66
	6	1.36	6.77	3.89	4.00	3.98	5.74	6.38
	7	0.70	7.34	1.99	2.81	2.81	3.15	3.45
	8	–	–	–	–	–	3.01	3.45
	9	–	–	–	–	–	2.02	2.52
	10	–	–	–	–	–	0.78	0.88
K-233	2	4.40	4.57	4.72	4.64	4.64	4.75	4.75
	3	4.66	5.43	5.63	5.39	5.12	5.75	5.75
	4	4.18	6.25	6.27	5.80	5.71	6.34	6.64
	5	3.05	7.02	5.82	5.39	5.33	6.80	7.48
	6	1.84	7.75	4.91	4.84	4.68	6.67	7.40
	7	0.99	8.44	2.69	3.54	3.50	3.94	4.30
	8	–	–	–	–	–	3.77	4.33
	9	–	–	–	–	–	2.58	3.26
	10	–	–	–	–	–	1.37	1.37
K-283	2	4.45	4.60	4.75	4.70	4.65	4.77	4.77
	3	4.80	5.47	5.68	5.45	5.18	5.78	5.78
	4	4.45	6.29	6.38	5.92	5.84	6.41	6.69
	5	3.37	7.07	6.09	5.64	5.55	6.92	7.58
	6	2.12	7.82	5.29	5.21	5.15	6.90	7.61
	7	1.15	8.52	3.05	3.92	3.86	4.29	4.67
	8	–	–	–	–	–	4.12	4.74
	9	–	–	–	–	–	2.87	3.62
	10	–	–	–	–	–	1.56	1.57

Table 4.14: Speedup obtained by different batch-verification algorithms
(Fixed-base scalar multiplication *used* during individual verification)
(Experiment not carried out for cells marked as –)

Curve	t	Same signer						
		N	N′	S2′	S3	$S3_{gcd}$	SP	SP_{gcd}
K-409	2	4.66	4.76	4.89	4.81	4.82	4.90	4.90
	3	5.17	5.69	5.87	5.66	5.41	5.94	5.94
	4	5.05	6.58	6.67	6.24	6.16	6.68	6.91
	5	4.11	7.43	6.68	6.15	6.06	7.25	7.84
	6	2.75	8.26	6.14	5.77	5.74	7.46	8.12
	7	1.56	9.04	3.87	4.60	4.53	5.09	5.51
	8	–	–	–	–	–	4.95	5.63
	9	–	–	–	–	–	3.57	4.47
	10	–	–	–	–	–	2.06	2.07
K-571	2	4.85	4.94	5.06	5.00	5.00	5.08	5.08
	3	5.48	5.90	6.09	5.90	5.65	6.15	6.15
	4	5.52	6.83	6.98	6.56	6.48	6.96	7.17
	5	4.74	7.72	7.19	6.61	6.50	7.59	8.15
	6	3.36	8.59	6.83	6.38	6.31	7.95	8.60
	7	2.00	9.41	4.66	5.24	5.14	5.78	6.20
	8	–	–	–	–	–	5.67	6.41
	9	–	–	–	–	–	4.25	5.24
	10	–	–	–	–	–	2.55	2.55

verification) are listed in Tables 4.9–4.12. Tables 4.9 and 4.10 correspond to the case of the same signer, whereas Tables 4.11 and 4.12 to the case of different signers. In these tables, individual verification does not use fixed-base double scalar multiplication.

The experimental results clearly indicate that $\mathrm{SP}_{\mathrm{gcd}}$ is the most efficient batch-verification algorithm for standard ECDSA signatures. Even for ECDSA$^{\#}$ signatures, Algorithm $\mathrm{SP}_{\mathrm{gcd}}$ often outperforms the naive method N$'$. The optimal batch size for Algorithm S2$'$ is $t = 5$ or 6 for Koblitz curves. With Algorithm $\mathrm{SP}_{\mathrm{gcd}}$, the optimal batch sizes are $t = 8$ for Koblitz curves. The maximum speedup is noticeably higher in Algorithm $\mathrm{SP}_{\mathrm{gcd}}$ than what is achieved by Algorithm S2$'$. For the sake of ease of comparison, the tables also list the speedup figures achieved by the naive and symbolic-computation algorithms introduced in Chapter 3.

Tables 4.13 and 4.14 assume that individual verification can afford the storage associated with fixed-base double scalar multiplication. The tables demonstrate that when the number t of signatures is $\geqslant 7$, this way of performing individual verification leads to practical benefits. For smaller t, τ-NAF scalar multiplication with affine coordinates is recommended for each individual verification.

4.8 The Group Structures in Quadratic Extensions

Here, we investigate the groups $E(\mathbb{F}_{q^2})$ and $E(\mathbb{F}_{2^{2l}})$ for the NIST prime and Koblitz curves [48] for which we have reported our experimental results. Since the sizes of the groups over the base fields are known, it is easy to compute the orders of the groups over quadratic extensions using a well-known result by Weil [16]. These sizes give an initial (sometimes complete) understanding of the structures of the groups over the extension fields.

The curve P-256 is defined over $\mathbb{F}_q$ for a 256-bit prime q. The order of $E(\mathbb{F}_q)$ is a prime n, so $E(\mathbb{F}_q)$ is a cyclic group. The size of the group $E(\mathbb{F}_{q^2})$ is

$$|E(\mathbb{F}_{q^2})| = 3 \times 5 \times 13 \times 179 \times n \times n',$$

where n' is a 241-bit prime (different from n). Since $|E(\mathbb{F}_{q^2})|$ is square-free, the group $E(\mathbb{F}_{q^2})$ is cyclic. However, it contains subgroups of small orders.

The curve P-521 is defined over $\mathbb{F}_q$ for a 521-bit prime q. The order of $E(\mathbb{F}_q)$ is a prime n, so $E(\mathbb{F}_q)$ is a cyclic group. The size of the group $E(\mathbb{F}_{q^2})$ is

$$|E(\mathbb{F}_{q^2})| = 5 \times 7 \times 69697531 \times 635884237 \times n \times n',$$

where n' is a 461-bit prime (different from n). Again $E(\mathbb{F}_{q^2})$ is cyclic, since its order is square-free. This group too has subgroups of small orders.

The Koblitz curve K-283 is defined over $\mathbb{F}_{2^d}$, $d = 283$, and has an order $4n$ for a prime n. If the group $E(\mathbb{F}_{2^d})$ is not cyclic, we must have $E(\mathbb{F}_{2^d}) \cong \mathbb{Z}_{2n} \oplus \mathbb{Z}_2$. But then, by the structure theorem of elliptic-curve groups of rank two, we have $2|(2^d - 1)$, which is impossible. So $E(\mathbb{F}_{2^{283}})$ is cyclic. In the quadratic extension $\mathbb{F}_{2^{2d}}$, the group has order

$$|E(\mathbb{F}_{2^{2d}})| = 2^3 \times n \times n'$$

for a 283-bit prime n' (different from n). As argued above, $E(\mathbb{F}_{2^{2d}})$ is easily seen to be cyclic. However, it contains subgroups of small orders.

The Koblitz curve K-571 is defined over $\mathbb{F}_{2^d}$, $d = 571$, and has order $4n$ for a prime n. We have the order

$$\begin{aligned} |E(\mathbb{F}_{2^{2d}})| &= 2^3 \times 835205577201087993065806 99 \times \\ &\quad 596201686362718542354710701 \times n \times n' \end{aligned}$$

for a 395-bit prime $n' \neq n$. Both $E(\mathbb{F}_{2^d})$ and $E(\mathbb{F}_{2^{2d}})$ are cyclic. Again, $E(\mathbb{F}_{2^{2d}})$ contains subgroups of small orders.

Since each of these groups in the quadratic extension has small-order subgroups, the sanity check is apparently preferred for all these curves. However, if the points of small orders on a curve over the quadratic extension do not have x-coordinates in the base field, then we can eliminate the sanity check.

The factorization of all NIST prime and Koblitz curves in respective quadratic extensions are provided in Table 4.15. In the table, p_i stands for a specific i-bit prime, and n stands for the order of the base point in the original curve. All these curves turn out to be cyclic in the quadratic extensions.

Table 4.15: Factorization of elliptic-curve groups in quadratic extensions

Curve	(Group order in quadratic extension)$/n$
P-192	$23 \times p_{94} \times p_{95}$
P-224	$3^2 \times 11 \times 47 \times 3015283 \times 40375823 \times p_{48} \times p_{118}$
P-256	$3 \times 5 \times 13 \times 179 \times p_{241}$
P-384	p_{385}
P-521	$5 \times 7 \times 69697531 \times 635884237 \times p_{461}$
K-163	$2^3 \times 653 \times 6521 \times p_{55} \times p_{85}$
K-233	$2^3 \times 92269 \times 114861079 \times 130034039 \times p_{63} \times p_{100}$
K-283	$2^3 \times p_{283}$
K-409	$2^3 \times 5616389 \times 90250595219 \times p_{116} \times p_{234}$
K-571	$2^3 \times p_{87} \times p_{89} \times p_{395}$

4.9 Chapter Summary

In this chapter, we propose a new and efficient batch-verification algorithm for original ECDSA signatures. This algorithm is based on Semaev's summation polynomials. We establish a connection of our summation-polynomial algorithm of this chapter with a symbolic-computation algorithm introduced in Chapter 3. We theoretically and experimentally establish the superiority of our batch-verification algorithm based on summation polynomials, both for the NIST prime and the NIST Koblitz families of elliptic curves. The elliptic-curve group structures over quadratic extensions of the base fields are determined for all NIST prime and Koblitz curves in order to gauge the necessity of running a sanity check associated with many of our batch-verification algorithms.

Chapter 5

Randomized Batch Verification of Standard ECDSA Signatures

In Chapters 3 and 4, several algorithms are proposed for the batch verification of ECDSA signatures. In this chapter, we propose three randomization methods for these batch-verification algorithms. Our first proposal is based on Montgomery ladders, and the second on computing square-roots in the underlying field. Both these techniques use numeric arithmetic only. Our third proposal exploits symbolic computations leading to a seminumeric algorithm. We theoretically and experimentally establish that for standard ECDSA signatures, our seminumeric randomization algorithm in tandem with the batch-verification algorithm $\mathrm{SP}_{\mathrm{gcd}}$ gives the best speedup over individual verification. In ECDSA$^{\#}$, each signature contains an extra bit to identify the correct y-coordinate of the elliptic-curve point appearing in the signature. In this case, the second numeric randomization algorithm followed by the naive batch-verification algorithm N' yields the best performance gains. We detail our study for NIST prime and Koblitz curves.

5.1 Introduction

We recall that an ECDSA signature [33] on a message M is a triple (M, r, s), where r is the x-coordinate of an elliptic-curve point R, and s is an integer that absorbs the hash of M. Both r and s are reduced modulo the size n of the elliptic-curve group. During

verification, two scalars u, v are computed using modulo n arithmetic, and the point R is reconstructed as $R = uP + vQ$, where P is the base point in the elliptic-curve group, and Q is the signer's public key. Verification succeeds if and only if $x(R) = r$.

Suppose that we want to verify a batch of t ECDSA signatures (M_i, r_i, s_i). For the i-th signature, the verification equation is $R_i = u_i P + v_i Q_i$. The t signatures can be combined as

$$\sum_{i=1}^{t} R_i = \left(\sum_{i=1}^{t} u_i\right) P + \left(\sum_{i=1}^{t} v_i Q_i\right). \tag{5.1}$$

Since the y-coordinates of R_i are not available in the signatures, we cannot straightaway compute the sum on the left side. In Chapters 3 and 4, several batch-verification algorithms are proposed to solve this problem. The naive algorithms are based upon the determination of the missing y-coordinate of each R_i using a square-root computation (we have $y_i^2 = r_i^3 + ar_i + b$). The symbolic-manipulation algorithms treat the unknown y-coordinates as symbols. Batch verification involves the eventual elimination of all these y-coordinates from Eqn (5.1) using the elliptic-curve equation. The symbolic algorithm $\mathrm{SP}_{\mathrm{gcd}}$ turns out to be the fastest of the batch-verification algorithms proposed in Chapter 4.

Bernstein at al. propose two attacks on these batch-verification algorithms [7]. They also suggest that these attacks can be largely eliminated by randomizing the batch-verification process (a concept introduced by Naccache et al. [45]). For randomly chosen non-zero multipliers $\xi_1, \xi_2, \ldots, \xi_t$, the individual verification equations are now combined as

$$\sum_{i=1}^{t} \xi_i R_i = \left(\sum_{i=1}^{t} \xi_i u_i\right) P + \left(\sum_{i=1}^{t} \xi_i v_i Q_i\right). \tag{5.2}$$

Since the y-coordinates of R_i are not available in the ECDSA signatures, Eqn (5.2) is not directly applicable. In this chapter, we propose three efficient ways of randomizing the batch-verification algorithms. We mostly concentrate on standard ECDSA signatures (M, r, s) on M. If the ECDSA signature contains an extra bit to identify the correct square-root y of $r^3 + ar + b$ [2], we call it an ECDSA$^{\#}$ signature. In another variant known as ECDSA* [2, 14], the entire point R replaces r in the signature. Neither ECDSA$^{\#}$ nor ECDSA* is accepted as a standard. Since ECDSA* results in an unreasonable expansion in the signature size without any increase in the security, we do not consider this variant in this chapter. ECDSA$^{\#}$, however, adds only

one extra bit to a signature, and so we study the implications of having this extra bit. Our three randomization techniques are based on the following ideas.

- *Montgomery ladders*: Given only the x-coordinate of an elliptic-curve point R, one can uniquely obtain the x-coordinate of any non-zero multiple ξR [42]. We first compute $x(\xi_i R_i)$ for all signatures in the batch. Then we feed these x-coordinates to the batch-verification algorithms.

- *Numeric computation*: We explicitly compute y_i from each r_i by taking a square-root of $r_i^3 + ar_i + b$. In ECDSA$^\#$, we uniquely obtain R_i from the extra bit. In ECDSA, we have two possibilities $\pm R$. We start with any possibility and numerically compute ξR or $-\xi R$ using standard elliptic-curve doubling and addition formulas.[1]

- *Seminumeric computation*: We treat each y_i as a symbol, and compute $\xi_i R_i$ as a point in the form $(h_i, k_i y_i)$, where the field elements h_i and k_i are computed from the knowledge of $r_i = x(R_i)$ alone. We precompute the quantity $r_i^3 + ar_i + b$ and follow a slightly modified version of the standard elliptic-curve scalar-multiplication algorithm. Joye in [34] proves that in prime fields the y-coordinate of $\xi_i R_i$ is of the form $(h_i(r_i), k_i(r_i) y_i)$ for functions h_i, k_i of r_i alone. In this chapter, we complement that study by providing explicit computational determination of h_i, k_i, and exploit this procedure to obtain a randomization algorithm that performs better than the above two methods for standard ECDSA signatures. Moreover, we derive such explicit formulas for Koblitz curves (in this chapter) and Edwards curves (in Chapter 6) too.

Since the only batch-verification algorithms that deal with standard ECDSA signatures are introduced in Chapters 3 and 4, randomizing these algorithms is of practical importance in real-time cryptographic applications—particularly those which run on resource-constrained platforms.

We experiment with the NIST prime family of elliptic curves [48]. Montgomery ladders face a few problems. Each iteration in the scalar-multiplication loop involves one addition and one doubling. More importantly, it is not known how

[1] A study of this method is inspired by a comment from an anonymous referee of a paper related to this chapter.

to adapt Montgomery ladders to windowed scalar multiplication (see Section 5.3). Montgomery's paper [43] proposes some ways of generating short Montgomery chains. As pointed out in [71, 72], the creations of the addition chains in these improved variants are rather costly. The practical method of [43] is effective only when the scalar multiplier remains constant, so the addition chain can be precomputed. Since this is not the case with randomizers, we have implemented only the binary ladder. The numeric and the seminumeric randomization algorithms can be adapted to any windowed variant. We theoretically and experimentally establish that the binary Montgomery ladder is slower than the best known windowed variants of the numeric and the seminumeric algorithms. Montgomery arithmetic is efficient for prime curves of the particular form $By^2 = x^3 + Ax^2 + x$. However, the NIST prime curves have large prime orders and cannot be converted to a curve in the Montgomery form which contains the point $(0,0)$ of order two. Point multiplication using Montgomery ladders is more resistant to simple *side-channel attacks* (SCA) than the numeric and seminumeric algorithms. In this chapter, SCA resistance is not of concern, since verification of digital signatures uses no private keys.

5.1.1 Attacks on ECDSA Batch Verification

Attack 1 In the first attack of Bernstein et al. [7], the batch verifier handles $t-2$ genuine signatures along with two forged signatures (r,s) and $(r,-s)$ on the same message M. Since the sum of the elliptic-curve points (r,s) and $(r,-s)$ is $\mathcal{O}$, the entire batch of t signatures is verified as genuine.

Attack 2 In the second attack, the forger knows a valid key pair (d_1,Q_1), and can fool the verifier by a forged signature for any message M_2 under any valid public key Q_2 along with a message M_1 under the public key Q_1. The forger selects a random k_2, computes $R_2 = k_2P$ and $r_2 = x(R_2)$. For another random s_2, the signature on M_2 under Q_2 is presented as (r_2,s_2). For the message M_1, the signature (r_1,s_1) is computed as $R_1 = r_2s_2^{-1}Q_2$, $r_1 = x(R_1)$, and $s_1 = (e_1 + r_1d_1)(k_2 - e_2s_2^{-1})^{-1}$, where $e_1 = H(m_1)$, $e_2 = H(m_2)$, and H is a secure hash function. Now, $R_1 + R_2$ and $(e_1s_1^{-1} + e_2s_2^{-1})P + r_1s_1^{-1}Q_1 + r_2s_2^{-1}Q_2$ have the same value as $(k_2P + r_2s_2^{-1}Q_2)$. These forged signatures are verified if they are in the same batch.

Both these attacks become infeasible by the use of randomizers. If the verifier chooses l-bit randomizers, the security of the batch-verification procedure increases by 2^l. The randomizers need not be of full lengths (of lengths close to that of the prime order q of the relevant elliptic-curve group). As discussed in [4], much smaller randomizers typically suffice to make most attacks on batch-verification schemes infeasible. If the underlying field is of size d bits, then the best known algorithms (the square-root methods [55, 56, 68]) to solve the ECDLP take $\tilde{O}(2^{d/2})$ times. As a result, $d/2$-bit randomizers do not degrade the security of the ECDSA scheme. Another possibility is to take $l = 128$ to get 128-bit security independent of the security guarantees of ECDSA.

5.2 Randomization of ECDSA Batch Verification

For the randomization of ECDSA batch verification as given in Eqn (5.2), the basic problem is to compute the x-coordinate $x(\xi R)$ from $r = x(R)$ and an l-bit scalar $\xi = 1\xi_{l-2}\xi_{l-3}\ldots\xi_1\xi_0$.

5.2.1 Montgomery Ladders

Montgomery ladders are discussed in [12, 36, 42]. For the sake of completeness, we present the relevant formulas for point addition and doubling. Suppose that $x(P_1) = h_1$, $x(P_2) = h_2$ and $x(P_1 - P_2) = h_4$ are known to us. We can compute $h_3 = x(P_1 + P_2)$ and $h_5 = x(2P_1)$ by Eqns (5.3) and (5.4), respectively.

$$h_3h_4(h_1 - h_2)^2 = (h_1h_2 - a)^2 - 4b(h_1 + h_2). \tag{5.3}$$

$$4h_5(h_1^3 + ah_1 + b) = (h_1^2 - a)^2 - 8bh_1. \tag{5.4}$$

The above formulas [12] are adapted from Montgomery's original derivation [42]. Fischer et al. [22] propose a slightly improved addition formula given by

$$(h_4 + h_3)(h_1 - h_2)^2 = 2(h_1 + h_2)(h_1h_2 + a) + 4b.$$

The Montgomery ladder described in Algorithm 5.1 never uses nor computes the y-coordinate of any point in its repeated double-and-add point-multiplication loop.

The loop maintains the invariance $T - S = R$. Since $x(T)$, $x(S)$ and $x(T-S) = x(R)$ are known, we can compute $x(T+S)$, $x(2S)$ and $x(2T)$.

Algorithm 5.1 Montgomery Ladder for Computing $x(\xi R)$ from ξ and $x(R)$

Initialize $x(S) := x(R)$ and $x(T) := x(2R)$.
For $(i = l-2, l-3, \ldots, 1, 0)$ {
  If $(\xi_i = 0)$, assign $x(T) := x(T+S)$ and $x(S) := x(2S)$;
  else assign $x(S) := x(T+S)$ and $x(T) := x(2T)$;
}
Return $x(S)$.

In many cases, using projective coordinates can speed up the Montgomery-ladder loop. Both the x- and the z-coordinates can be computed from the knowledge of $x(R)$ alone (we assume $z(R) = 1$). Some explicit formulas can be found at [9, 16]. Fischer et al. [22] propose an optimization of the Montgomery loop. Irrespective of the bit value ξ_i, the loop computes the x- and z-coordinates of two points $P+Q$ and $2P$, where P is one of the points S, T, and Q is the other point. These operations can be combined together yielding a reduced count of field operations. The problem with Algorithm 5.1 is that no effective windowed adaptation of it is known (see [71, 72] and Section 5.3).

5.2.2 Numeric Computation

We first compute a square-root y of $r^3 + ar + b$. The point R is either (r, y) or $(r, -y)$. An ECDSA$^{\#}$ signature has enough information to identify which of these two points is the correct R. An ECDSA signature cannot resolve this ambiguity. However, that is not a serious problem, since both ξR and $-\xi R$ have the same x-coordinate. Therefore, we start with any of the two points $\pm R$, and compute its ξ-th multiple using any standard elliptic-curve scalar-multiplication algorithm. The y-coordinate of this multiple is also computed as a byproduct.

A square root of $r^3 + ar + b$ modulo the prime q can be computed by well-known algorithms (like Tonelli-Shanks algorithm [68]). If $q \equiv 3 \pmod 4$, then $(r^3 + ar + b)^{(q+1)/4} \pmod q$ is such a square root. Each square-root finding algorithm essentially requires the cost of an exponentiation in the field $\mathbb{F}_q$.

The numeric method has an important bearing on the naive batch-verification

methods N and N′ of Chapter 3. These two algorithms start by computing the square roots of $r_i^3 + ar_i + b$. If these algorithms are randomized by the numeric method, the y-coordinates of $\xi_i R_i$ are already available (up to sign in ECDSA, and uniquely in ECDSA$^{\#}$), and need not be computed again from the x-coordinates of $\xi_i R_i$. This lets the naive algorithms save significant time. The symbolic batch-verification methods (like S2′) do not use and therefore do not benefit from an explicit knowledge of the y-coordinates.

The numeric method in the context of ECDSA$^{\#}$ has another advantage. Since the points R_i are now known uniquely, we can use multiple scalar multiplication. For example, computing $\xi_1 R_1 + \xi_2 R_2$ in a single double-and-add loop needs only l doubling and at most l addition operations, where l is the length of the randomizers. On the contrary, computing $\xi_1 R_1$ and $\xi_2 R_2$ separately by even the best windowed method requires $2l$ doubling operations and some more additions. Thus, the naive batch-verification algorithm N′ derives an additional boost in its performance from multiple scalar multiplication.

5.2.3 Seminumeric Computation

We treat the y-coordinate of $R = (r, y)$ as a symbol satisfying $y^2 = r^3 + ar + b$.

Theorem 5.2.1. *Any non-zero multiple uR of R can be expressed as (h, ky), where h and k are field elements fully determined by (u and) the x-coordinate r of R.*

Proof. R itself can be so expressed with $h = r$ and $k = 1$. Next, suppose that $P_1 = (h_1, k_1 y)$ and $P_2 = (h_2, k_2 y)$ are two distinct non-zero multiples of R with $P_3 = P_1 + P_2 \neq \mathcal{O}$. The addition formula gives $P_3 = (h_3, k_3 y)$, where

$$\begin{aligned} h_3 &= \left(\frac{k_1 - k_2}{h_1 - h_2}\right)^2 (r^3 + ar + b) - h_1 - h_2, \text{ and} \\ k_3 &= \left(\frac{k_1 - k_2}{h_1 - h_2}\right)(h_1 - h_3) - k_1. \end{aligned}$$

Let $P_4 = 2P_1$. We have $P_4 = (h_4, k_4 y)$, where

$$\begin{aligned} h_4 &= \left(\frac{3h_1^2 + a}{2k_1}\right)^2 \left(\frac{1}{r^3 + ar + b}\right) - 2h_1, \text{ and} \\ k_4 &= \left(\frac{3h_1^2 + a}{2k_1}\right)\left(\frac{h_1 - h_4}{r^3 + ar + b}\right) - k_1. \end{aligned}$$

Finally, the opposite of (h,ky) is $(h,(-k)y)$. This completes the inductive proof. □

We represent the multiple (h,ky) of R by the pair (h,k) of field elements. The *symbol* y need not be explicitly maintained. R itself is represented by the pair $(r,1)$. Upon $(r,1)$ as input, we precompute the quantity r^3+ar+b and its inverse, and run the standard repeated double-and-add loop of Algorithm 5.2 with these revised addition and doubling formulas. At the end of the loop, the two computed field elements h,k yield the desired multiple $\xi R=(h,ky)$. In short, we do not need to carry out any symbolic computation at all for obtaining ξR.

Algorithm 5.2 Seminumeric Computation of $\xi R=(h,ky)$ from ξ and $R=(r,y)$

Precompute the field elements r^3+ar+b and $(r^3+ar+b)^{-1}$.
Initialize $S:=(r,1)$.
For $(i=l-2,l-3,\ldots,1,0)$ {
  Assign $S:=2S$ (use seminumeric doubling formula).
  If $(\xi_i=1)$, assign $S:=S+R$ (use seminumeric addition formula).
}
Return S.

The modified addition formula involves only one extra field multiplication (by the precomputed quantity r^3+ar+b) compared to the standard elliptic-curve addition formula. Point doubling requires two extra field multiplications (each by the precomputed inverse $(r^3+ar+b)^{-1}$). If we use Jacobian projective coordinates, we can rearrange the formula of point doubling to absorb those two extra field multiplications (see Section 5.2.4). This standard double-and-add algorithm can be adapted to any windowed variant. Some variants require precomputing multiples uR of R for some small values of u. All these multiples are precomputed and stored as pairs of field elements.

The knowledge of the entire points R_1,R_2 allows us to compute $\xi_1R_1+\xi_2R_2$ using a single double-and-add loop, yielding noticeable speedup over two point multiplications. If the y-coordinates of R_1 and R_2 are treated as symbols y_1,y_2, then too $\xi_1R_1+\xi_2R_2$ can be computed seminumerically. Any non-zero point of the form uR_1+vR_2 can be expressed as $(h+jy_1y_2,ky_1+ly_2)$ for field elements h,j,k,l uniquely determined by the x-coordinates r_1,r_2 (and u,v) alone. Addition and doubling of such points can be rephrased numerically in terms of these field elements. For example,

let $P_1 = (h_1 + j_1 y_1 y_2, k_1 y_1 + l_1 y_2)$ and $P_2 = (h_2 + j_2 y_1 y_2, k_2 y_1 + l_2 y_2)$ be two (distinct) points of the form $uR_1 + vR_2$. In order to compute their sum, we first compute the slope $\lambda = \frac{(k_1-k_2)y_1+(l_1-l_2)y_2}{(h_1-h_2)+(j_1-j_2)y_1y_2}$. Using the symbolic-manipulation techniques of Chapters 3 and 4, we free the denominator of y_1, y_2 (multiply by $(h_1 - h_2) - (j_1 - j_2)y_1y_2$ and substitute $y_i^2 = r_i^3 + ar_i + b$ for $i = 1,2$). We simplify the numerator too to express λ as $\alpha y_1 + \beta y_2$. Therefore, λ^2 and $x(P_1 + P_2) = \lambda^2 - x(P_1) - x(P_2)$ are of the form $\gamma + \delta y_1 y_2$. This process of symbolic computation of $x(P_1 + P_2)$ can be replaced by explicit numeric formulas in $h_1, k_1, j_1, l_1, h_2, k_2, j_2, l_2$. The y-coordinate of $P_1 + P_2$ and point doubling can be analogously handled. The resulting numeric formulas turn out to be clumsy, and are not expected to benefit the computation of $\xi_1 R_1 + \xi_2 R_2$ in a single double-and-add loop. For the weighted sum of three or more points, this idea of seminumeric computation can be extended at least in theory, but chances of getting practical benefits are rather slim.

5.2.4 Explicit Seminumeric Formulas for Prime Curves

Let $P_1 = (h_1, k_1 y, l_1)$ and $P_2 = (h_2, k_2 y, l_2)$ be two non-zero multiplies of $P = (r, y, 1) \in E(\mathbb{F}_q)$ with $P_1 \neq \pm P_2$, and let $P_3 = P_1 + P_2 = (h_3, k_3 y, l_3)$ and $P_4 = 2P_1 = (h_4, k_4 y, l_4)$. Before the double-and-add loop, the quantities $f_y = (r^3 + ar + b)$ and $f'_y = (r^3 + ar + b)^{-1}$ are precomputed. Jacobian coordinates are used.

Point Addition

$$L_1 = l_1^2,\ L_2 = l_2^2,\ U_1 = h_1 \cdot L_2,\ U_2 = h_2 \cdot L_1,\ S_1 = k_1 \cdot L_2 \cdot l_2,\ S_2 = k_2 \cdot L_1 \cdot l_1,$$
$$H = U_2 - U_1,\ H_2 = H^2,\ H_3 = H_2 \cdot H,\ R = S_2 - S_1,\ R_2 = R^2 \cdot f_y,\ U_3 = U_1 \cdot H_2,$$
$$h_3 = R_2 - H_3 - 2 \cdot U_3,\ k_3 = R \cdot (U_3 - h_3) - S_1 \cdot H_3,\ l_3 = H \cdot l_1 \cdot l_2.$$

Point Doubling

$$K = k_1^2,\ L = l_1^2,\ T_1 = h_1 - l_2,\ T_2 = h_1 + l_2,\ T = T_1 \cdot T_2,\ H = 3 \cdot T,\ H_1 = H \cdot f'_y,$$
$$H_2 = H \cdot H_1,\ R_1 = 4 \cdot K,\ R = R_1 \cdot h_1,\ h_4 = (H_2 - 2 \cdot R),\ k_4 = H_1 \cdot (R - h_4) - R_1^2/2,$$
$$l_4 = 2 \cdot l_1 \cdot k_1.$$

5.3 Non-Adaptability of Montgomery Ladders to Windowed Variants

The seminumeric formulas readily adapt to windowed or addition-chain-based variants, and is only slightly slower than the standard variant. On the contrary, we now heuristically argue that Montgomery ladders do not benefit from these windowed variants. We assume that a Montgomery ladder is based upon the availability of only the following two primitives. To the best of our knowledge, no other known primitive operation can be exploited by Montgomery ladders.

1. *Montgomery addition:* Given $x(U), x(V)$ and $x(U-V)$, it is possible to compute $x(U+V)$.
2. *Montgomery doubling:* Given $x(U)$, it is possible to compute $x(2U)$.

Each iteration of the Montgomery ladder performs one addition and one doubling. Let us think about an adaptation of this to a standard w-bit windowed variant. An iteration in this variant converts $x(S), x(T)$ to $x(S') = x(2^w S + aR)$ and $x(T') = x(2^w S + (a+1)R)$ for some $a \in \{0, 1, 2, \ldots, 2^w - 1\}$. If we make w doubling operations of S, we obtain $2^w S$. But then, computing the x-coordinate of any point of the form $2^w S + \alpha P$ with $\alpha \neq 0$ requires the knowledge of the x-coordinate of $2^w S - \alpha P$. Conversely, computing $x(2^w S - \alpha P)$ requires the knowledge of $x(2^w S + \alpha P)$. Therefore, the Montgomery loop in this way cannot convert $x(S), x(T)$ to $x(S'), x(T')$ using only w doubling and one or two addition primitives, even if we precompute the x-coordinates of small multiples of R.

A way in which the above problem can be solved is the computation of $x(2^w S + \alpha R)$ using the doubling primitive on $x(2^{w-1} S + \beta R)$, or the addition primitive on $x(u_1 S + \beta_1 R)$ and $x(u_2 S + \beta_2 R)$ with $u_1 + u_2 = 2^w$. If $u_1 \neq u_2$, then this addition also requires the availability of $x((u_1 - u_2)S + (\beta_1 - \beta_2)R)$. Therefore, the minimal requirement in this case is having $u_1 = u_2 = 2^{w-1}$.

The windowed Montgomery loop needs to compute the x-coordinates of two points $2^w S + \alpha_1 R$ and $2^w S + \alpha_2 R$ with one of α_1, α_2 odd and the other even. Indeed, $\alpha_1 = a$ and $\alpha_2 = a + 1$, but the following argument does not require α_1, α_2 to have

this special property. The point corresponding to the the odd value of α cannot be obtained by doubling. The minimal requirement for computing this point is the availability of the x-coordinates of two points of the form $2^{w-1}S + \beta_1 R$ and $2^{w-1}S + \beta_2 R$. Again, one of β_1, β_2 must be odd, and the other even. The best way to obtain the x-coordinate of $2^w S + \alpha_i R$ with α_i even is using the doubling primitive on one of the two points $2^{w-1}S + \beta_1 R$ and $2^{w-1}S + \beta_2 R$. In short, the computation of $x(2^w S + \alpha_1 R)$ and $x(2^w S + \alpha_2 R)$ boils down to the computation of at least $x(2^{w-1}S + \beta_1 R)$ and $x(2^{w-1}S + \beta_2 R)$ at the expense of one addition and one doubling primitive. Recursively, we conclude that the windowed iteration requires at least w addition and w doubling primitives. Not only this is the same count of operations as in the non-windowed version, but also the windowed procedure is essentially the same as w iterations of the non-windowed loop.

The argument presented above is not a rigorous derivation of the shortest length of a Montgomery ladder. For a rigorous analysis, we refer the reader to Montgomery's manuscript [43]. Our algorithm resembles the binary method discussed in this article, for which the bound is close to twice the bit length of the randomizer. There are theoretically faster alternatives proposed in this article. However, the practical implementation behavior of these alternatives seems to be unavailable in the literature. The reported software and hardware implementations deal with only the binary ladder. For example, see [24, 58]. In view of this, we have implemented only the non-windowed version of Montgomery ladders. We are, on the other hand, free to choose any windowed variant for the seminumeric algorithm.

5.4 Comparison Among the Randomization Algorithms

In this section, we first count the field operations in the randomization algorithms. For each of these algorithms, we take the best variant (windowed, if applicable, and with a suitable choice of the coordinate system) known to us. We then experimentally validate our theoretical observations.

5.4.1 Comparison of Montgomery Ladders and Seminumeric Method

For the purpose of theoretical comparison, we use standard projective coordinates in the Montgomery-ladder method, and Jacobian projective coordinates in the NAF variant [44, 59] of the seminumeric method. The Montgomery-ladder method in standard projective coordinates produced the best results almost always, whereas the NAF variant of the seminumeric method in Jacobian projective coordinates gave us the best results for curves over large fields. Comparisons among other variants can be analogously carried out.

Let us analyze the Montgomery-ladder implementation first. Let $P_1 = (h_1, k_1, l_1)$, $P_2 = (h_2, k_2, l_2)$, and $P_1 - P_2 = (r, -y, 1) \in E(\mathbb{F}_p)$ be given in projective coordinates. We do not use the y-coordinates k_1, k_2, y. We only compute the x- and z-coordinates of $P_1 + P_2$ and $2P_1$ using the Montgomery-ladder adapted to projective coordinates [22]:

$$\begin{aligned} x(P_1+P_2) &= 2(h_1l_2+h_2l_1)(h_1h_2+al_1l_2)+4bl_1^2l_2^2-r(h_1l_2-h_2l_1)^2, \\ z(P_1+P_2) &= (h_1l_2-h_2l_1)^2, \\ x(2P_1) &= (h_1^2-al_1^2)^2-8bh_1l_1^3, \text{ and} \\ z(2P_1) &= 4h_1l_1(h_1^2+al_1^2)+4bl_1^4. \end{aligned}$$

If we precompute the field element $-4b$, point addition and point doubling require $M_{Mont} = 14M + 5S + 9A + 5(2*)$ field operations (see Table 5.1 for the notations, and [22] for the derivation of this count). For an l-bit randomizer, the Montgomery-ladder scalar-multiplication does lM_{Mont} operations.

Next, we analyze the seminumeric method. Any non-zero multiple of $(r, y, 1) \in E(\mathbb{F}_q)$ is of the form $(\beta_x, \beta_y y, \beta_z)$ with $\beta_x, \beta_y, \beta_z \in \mathbb{F}_p$. Let $P_1 = (h_1, k_1 y, l_1)$ and $P_2 = (h_2, k_2 y, l_2)$ be two such multiples, where $P_1 \neq \pm P_2$, and y satisfies the equation $y^2 = r^3 + ar + b$ with r known. We treat y as a symbol for the rest of this section. We modify the point-addition and doubling formulas of Section 5.2.3 as given in [24]. All these formulas are derived assuming that $a = -3$. In particular, the x-, y- and z-coordinates of $P_3 = P_1 + P_2 = (h_3, k_3 y, l_3)$ and $P_4 = 2P_2 = (h_4, k_4 y, l_4)$ are computed as:

$$\begin{aligned} &H = h_2l_1^2 - h_1l_2^2,\ R = k_2l_1^3 - k_1l_2^3,\ R' = R^2y^2,\ h_3 = R' - H^3 - 2h_1l_2^2H^2, \\ &k_3 = R(k_1l_2^2 - h_3) - k_1l_2^3, \text{ and } l_3 = Hl_1l_2. \end{aligned}$$

Table 5.1: Descriptions of the Symbols

Symbol	Description
M	Finite field Multiplication
S	Finite field Square
I	Finite field Inverse
A	Finite field Addition or subtraction
$(u*)$	Finite field multiplication by the constant element u
M_{Mont}	Montgomery-ladder merged addition-doubling in projective coordinates
A_{Semi}	Seminumeric point addition in the mentioned coordinates
D_{Semi}	Seminumeric point doubling in the mentioned coordinates

$$H_1 = 3(h_1 - l_1^2)(h_1 + l_1^2),\ H_2 = H_1^2/y^2,\ R'' = 4h_1k_1^2,\ h_4 = H_2 - 2R'',$$
$$k_4 = H_1(R'' - h_4)/y^2 - 8k_1^4, \text{ and } l_4 = 2k_1l_1.$$

We need to perform $A_{Semi} = 13M + 4S + 6A + 1(2*)$ and $D_{Semi} = 6M + 3S + 5A + 1(3*) + 4(2*) + 1(\frac{1}{2}*)$ field operations for point addition and doubling (see Section 5.2.4), respectively (with $\frac{1}{2}$, y^2 and y^{-2} precomputed). Each point addition requires only one extra field multiplication than the best implementations mentioned in [9]. Point doubling has the same multiplication count as these best implementations. If we use the w-NAF [25] representation of l-bit randomizers, then there are on an average $\frac{l}{w+1}$ non-zero digits [11, 53]. For each of these non-zero digits, A_{Semi} operations are required. Point doubling (D_{Semi}) is done for each of the l bits. Furthermore, for precomputing 2^{w-2} multiples of $(r, y, 1)$, we need $2^{w-2} - 1$ point additions and one point doubling. Opposites of these multiples take almost zero computation cost.

The seminumeric algorithm is faster than Montgomery-ladder algorithm if:

$$\left(2^{w-2} - 1 + \frac{l}{w+1}\right) A_{Semi} + (l+1)D_{Semi} \leqslant lM_{Mont} \tag{5.5}$$

Following the convention of [16], we ignore the times required to multiply a field element by a constant (such as 2, 3 or $1/2$) and to add two field elements, since these operations take negligible times compared to field multiplication and squaring. Moreover, as suggested in [9], we take the squaring and multiplication times the same (that is, $1S = 1M$). With these simplifications, Eqn (5.5) can be rewritten as

$$17\left(2^{w-2} - 1 + \frac{l}{w+1}\right) + 9(l+1) \leqslant 19l.$$

Rearrangement of this equation gives

$$\left(10 - \frac{17}{w+1}\right) l \geqslant 9 + 17(2^{w-2} - 1).$$

Putting $w = 4$ in the equation, we get $l \geqslant 9.09$. This theoretically establishes that for $l \geqslant 10$, the seminumeric algorithm is faster than the Montgomery ladder.

It is important to highlight that the worst-case overhead $(A_{Semi} + D_{Semi})$ of an iteration of the seminumeric loop is more than the overhead M_{Mont} of each iteration of the Montgomery-ladder loop. However, the windowed variants of the seminumeric iteration are much more efficient than this worst case, on an average. Montgomery ladders, on the other hand, are unable to take this advantage.

5.4.2 Comparison of Numeric and Seminumeric Methods

The numeric and seminumeric methods use essentially the same formulas of scalar multiplication. Each seminumeric point addition uses one extra field multiplication by the precomputed quantity $r^3 + ar + b$. Seminumeric point doubling requires exactly the same number of field multiplications as needed by numeric point doubling. The numeric algorithm, on the other hand, has the extra overhead of a square-root computation. As mentioned in Section 5.2.2, this overhead is essentially that of an exponentiation in $\mathbb{F}_q$. We use an efficient windowed modular exponentiation algorithm. The effect of this overhead on the computation of ξR depends on the bit length of ξ. If ξ is a full-length scalar (that is, of bit length near that of q), then the extra overhead is slightly less than that associated with the extra multiplication in the seminumeric loop. In practice, the cryptographically most meaningful length of ξ is about half of that of q. In this case, the square-root computation overhead per bit of the randomizer ξ is doubled, and we expect the seminumeric method to be faster than the numeric method.

More precisely, let d be the bit length of q. Each square-root computation by the w-NAF method needs

$$(1S + (2^{w-2} - 1)M) + (dS + \frac{d}{w+1}M)$$

field operations. If we put $1S = 1M$, this is the same as

$$2^{w-2} + d\left(1 + \frac{1}{w+1}\right)$$

multiplications. The w-NAF scalar-multiplication time with an explicitly known y-coordinate and an l-bit scalar is about the same as that of

$$16\left(2^{w-2}-1+\frac{l}{w+1}\right)+9(l+1)$$

field multiplications [9]. Thus, the total overhead of the numeric method is that of

$$2^{w-2}+d\left(1+\frac{1}{w+1}\right)+16\left(2^{w-2}-1+\frac{l}{w+1}\right)+9(l+1)$$

multiplications for each point. On the other hand, the seminumeric method with an l-bit scalar needs an equivalent of

$$17\left(2^{w-2}-1+\frac{l}{w+1}\right)+9(l+1)$$

field multiplications for each point, yielding a saving of

$$\left(d+\frac{d-l}{w+1}+1\right)$$

field multiplications. For $l=\frac{d}{2}$ and $w=4$, $\left(\frac{11}{10}\right)d+1$ multiplications are saved.

5.4.3 Effects of Randomization on Batch-Verification Algorithms

Tables 5.11–5.18 illustrate the performance degradation caused by randomization. The speedup figures are computed over individual verification and pertain to the situation where all the signatures come from the same signer. In the table, t is the batch size and l is the bit length of the randomizer. We have taken two cryptographically meaningful values of l (half-length and 128). For original ECDSA signatures, the seminumeric randomization method gives the best performance. For ECDSA$^{\#}$, the extra square-root identifying bits give the points R_i uniquely, so the numeric randomization method is the preferred choice. In each case, the best possible windowed variant is used to compute the speedup. Whenever possible, the best windowed variants are replaced by the faster multiple scalar multiplication method. The tables also list the speedup figures without randomization. Although the increased security provided by randomization incurs reasonable overhead, we still have sizable speedup over individual verification.

5.4.4 Experimental Comparison

We continue our experiments in the same environment as that of the experiments of Chapters 3 and 4. Here, we have also used the symbolic-computation facilities of the GP/PARI calculator in our programs. All other functions (like scalar multiplication and square-root computation) are written as subroutines with minimal function-call overheads. Since the algorithms are evaluated in terms of number of field operations, this gives a fare comparison of experimental data with the theoretical estimates. We have implemented windowed, w-NAF and frac-w-NAF methods [40, 41]. We have used affine and Jacobian projective coordinates.

Tables 5.2 and 5.3 list the average times required for numeric single and double scalar multiplications. The average times of randomization achieved by the seminumeric and the Montgomery-ladder algorithms are listed in Tables 5.6 and 5.7 for the NIST prime curves. In Tables 5.2, 5.3, 5.6 and 5.7, all the scalars are randomly chosen and so we include the computation time required to compute the addition chain. In Tables 5.4 and 5.5, we include the average times to compute single and double scalar multiplications where we choose the addition chain of the scalars randomly.[2] Similarly, the average times of randomization achieved the seminumeric and the Montgomery-ladder algorithms, where we choose the addition chains of the scalars randomly, are listed in Tables 5.8 and 5.9.

Here, w is the window size, and l is the bit length of the scalar multiplier (randomizer in the batch-verification application). As mentioned in Section 5.1.1, we have chosen l to be 128, $d/2$ and d (where $d = |q|$). The seminumeric algorithm is found to be faster than the Montgomery-ladder algorithm, particularly for large randomizers. For NIST prime curves, the experimental speedup is by about a factor of two. This is consistent with the theoretical estimates.

Table 5.10 lists the overheads associated with the square-root computations. In order to compare the performances of the numeric method and the seminumeric method, we add the best possible numeric scalar multiplication time to the best possible square-root computation time. For example, for full-length randomizers with the scalars randomly chosen, the best total overheads of the numeric algorithm are

[2] An anonymous referee of one of our papers suggested using randomly chosen addition chains instead of randomly chosen scalars for which (short) addition chains need to be computed.

Table 5.2: Times (in ms) of the numeric method (square-root computation times are not inlcuded) with scalars randomly chosen

↓ Algorithm / Curve →			P-192		P-224		P-256	
			$l = 96$	$l = 192$	$l = 112$	$l = 224$	$l = 128$	$l = 256$
	$w = 3$		1.20	2.28	1.40	2.92	1.88	3.64
w-numeric	$w = 4$		1.20	2.25	1.45	2.80	1.76	3.52
(affine)	$w = 5$		1.32	2.36	1.56	2.84	1.93	3.56
	$w = 3$		1.24	2.40	1.56	3.05	1.88	3.69
w-NAF-numeric	$w = 4$		1.16	2.36	1.44	2.92	1.80	3.69
(affine)	$w = 5$		1.20	2.32	1.52	2.96	1.84	3.52
	$w = 3$	$m = 1$	1.36	2.64	1.68	3.32	2.04	4.05
		$m = 1$	1.36	2.65	1.64	3.20	2.00	4.01
	$w = 4$	$m = 3$	1.36	2.60	1.60	3.20	1.96	4.00
		$m = 5$	1.36	2.61	1.64	3.20	2.00	3.96
		$m = 1$	1.36	2.60	1.68	3.20	2.04	3.92
Frac-w-NAF-numeric		$m = 3$	1.36	2.65	1.68	3.24	2.04	3.96
(affine)		$m = 5$	1.36	2.64	1.64	3.25	2.04	3.93
	$w = 5$	$m = 7$	1.36	2.65	1.64	3.20	2.00	3.93
		$m = 9$	1.40	2.60	1.72	3.21	2.08	3.96
		$m = 11$	1.41	2.65	1.72	3.28	2.08	3.96
		$m = 13$	1.44	2.64	1.72	3.20	2.09	3.97
	$w = 3$		1.16	2.25	**1.36**	2.88	1.68	3.24
w-numeric	$w = 4$		1.17	**2.20**	1.44	**2.77**	1.60	**3.08**
(Jacobian projective)	$w = 5$		1.36	2.33	1.64	2.81	1.80	3.17
	$w = 3$		1.16	2.24	1.48	2.92	1.64	3.20
w-NAF-numeric	$w = 4$		**1.12**	**2.20**	1.40	2.84	**1.56**	3.24
(Jacobian projective)	$w = 5$		1.16	**2.20**	1.49	2.80	1.64	**3.08**
	$w = 3$	$m = 1$	1.32	2.48	1.64	3.21	1.81	3.60
		$m = 1$	1.28	2.52	1.60	3.13	1.77	3.52
	$w = 4$	$m = 3$	1.32	2.52	1.61	3.12	1.76	3.45
		$m = 5$	1.29	2.52	1.56	3.13	1.76	3.53
		$m = 1$	1.32	2.52	1.64	3.17	1.85	3.45
Frac-w-NAF-numeric		$m = 3$	1.33	2.52	1.68	3.20	1.84	3.44
(Jacobian projective)		$m = 5$	1.36	2.52	1.61	3.20	1.80	3.48
	$w = 5$	$m = 7$	1.36	2.52	1.64	3.16	1.77	3.44
		$m = 9$	1.36	2.52	1.68	3.16	1.84	3.45
		$m = 11$	1.36	2.52	1.73	3.17	1.88	3.48
		$m = 13$	1.36	2.56	1.76	3.16	1.88	3.52
Double scalar multiplcation (affine)			**1.75**	**3.39**	**2.14**	**4.22**	**2.63**	**5.27**
Double scalar multiplcation (Jacobian projective)			2.03	3.88	2.51	4.97	2.83	5.60

Table 5.3: Times (in ms) of the numeric method (square-root computation times are not included) with scalars randomly chosen

↓ Algorithm / Curve →			P-384			P-521		
			$l = 128$	$l = 192$	$l = 384$	$l = 128$	$l = 256$	$l = 521$
	$w = 3$		2.44	3.48	6.97	3.04	5.81	11.81
w-numeric	$w = 4$		2.28	3.36	6.69	2.88	5.73	11.32
(affine)	$w = 5$		2.48	3.53	6.68	3.16	5.84	11.32
	$w = 3$		2.36	3.53	7.21	2.96	5.84	12.08
w-NAF-numeric	$w = 4$		2.24	3.41	6.89	2.73	5.64	11.44
(affine)	$w = 5$		2.28	3.44	6.77	2.92	5.69	11.28
	$w = 3$	$m = 1$	2.57	3.81	7.61	3.08	6.13	12.68
		$m = 1$	2.44	3.65	7.36	2.96	6.01	12.12
	$w = 4$	$m = 3$	2.48	3.68	7.37	2.96	6.05	12.01
		$m = 5$	2.48	3.56	7.37	2.97	6.01	12.04
		$m = 1$	2.52	3.72	7.41	3.08	6.08	12.00
Frac-w-NAF-numeric		$m = 3$	2.53	3.69	7.41	3.08	6.08	12.00
(affine)		$m = 5$	2.56	3.68	7.40	3.00	6.12	12.00
	$w = 5$	$m = 7$	2.57	3.76	7.45	3.00	6.01	12.05
		$m = 9$	2.56	3.73	7.37	3.13	6.05	12.08
		$m = 11$	2.60	3.76	7.40	3.16	6.05	12.08
		$m = 13$	2.60	3.76	7.41	3.17	6.01	12.04
	$w = 3$		2.00	2.84	5.68	2.52	4.77	9.77
w-numeric	$w = 4$		1.84	2.72	5.36	2.40	4.69	9.29
(Jacobian projective)	$w = 5$		2.09	2.92	**5.33**	2.69	4.80	**9.09**
	$w = 3$		1.88	2.80	5.77	2.41	4.72	9.81
w-NAF-numeric	$w = 4$		**1.81**	2.72	5.49	**2.24**	**4.60**	9.25
(Jacobian projective)	$w = 5$		1.88	**2.69**	5.36	2.40	4.65	9.17
	$w = 3$	$m = 1$	2.08	3.08	6.20	2.60	4.96	10.36
		$m = 1$	2.00	2.96	5.97	2.49	4.92	10.00
	$w = 4$	$m = 3$	1.97	2.93	5.88	2.48	4.96	9.85
		$m = 5$	2.00	2.92	5.93	2.44	4.97	9.89
		$m = 1$	2.12	3.00	5.92	2.65	5.01	9.93
Frac-w-NAF-numeric		$m = 3$	2.12	3.04	5.97	2.64	5.05	9.89
(Jacobian projective)		$m = 5$	2.12	3.05	5.93	2.57	5.05	9.93
	$w = 5$	$m = 7$	2.12	3.04	5.92	2.56	4.96	9.97
		$m = 9$	2.12	3.08	5.93	2.60	5.04	9.97
		$m = 11$	2.13	3.09	5.97	2.64	5.04	10.00
		$m = 13$	2.16	3.08	5.96	2.68	5.00	9.93
Double scalar multiplcation (affine)			3.29	4.99	9.96	4.17	8.22	16.84
Double scalar multiplcation (Jacobian projective)			**3.23**	**4.90**	**9.78**	**4.04**	**7.86**	**16.16**

Table 5.4: Times (in ms) of the numeric method (square-root computation times are not inlcuded) with addition chains of the scalars chosen randomly

↓ Algorithm / Curve →			P-192		P-224		P-256	
			$l = 96$	$l = 192$	$l = 112$	$l = 224$	$l = 128$	$l = 256$
	$w = 3$		1.12	2.16	1.36	2.89	1.80	3.56
w-numeric	$w = 4$		1.12	2.16	1.40	2.80	1.73	3.40
(affine)	$w = 5$		1.28	2.28	1.56	2.77	1.88	3.40
	$w = 3$		1.12	2.21	1.36	2.76	1.72	3.41
w-NAF-numeric	$w = 4$		1.08	2.12	1.33	2.64	1.64	3.37
(affine)	$w = 5$		1.08	2.12	1.36	2.64	1.68	3.29
	$w = 3$	$m = 1$	1.12	2.12	1.36	2.73	1.72	3.33
		$m = 1$	1.12	2.17	1.36	2.64	1.68	3.32
	$w = 4$	$m = 3$	1.08	2.08	1.32	2.65	1.65	3.36
		$m = 5$	1.08	2.08	1.41	2.68	1.64	3.29
		$m = 1$	1.08	2.13	1.40	2.68	1.72	3.25
Frac-w-NAF-numeric		$m = 3$	1.08	2.12	1.40	2.68	1.68	3.28
(affine)		$m = 5$	1.12	2.17	1.36	2.64	1.69	3.29
	$w = 5$	$m = 7$	1.12	2.12	1.36	2.64	1.64	3.28
		$m = 9$	1.12	2.12	1.44	2.64	1.76	3.32
		$m = 11$	1.16	2.16	1.44	2.64	1.72	3.29
		$m = 13$	1.16	2.16	1.44	2.64	1.72	3.32
	$w = 3$		1.12	2.08	1.36	2.76	1.60	3.08
w-numeric	$w = 4$		1.04	2.08	1.40	2.69	1.56	2.96
(Jacobian projective)	$w = 5$		1.29	2.24	1.64	2.80	1.73	3.00
	$w = 3$		1.04	2.12	1.32	2.64	1.48	2.92
w-NAF-numeric	$w = 4$		1.05	2.00	**1.28**	2.56	**1.40**	2.84
(Jacobian projective)	$w = 5$		1.08	2.00	1.36	**2.53**	1.48	**2.76**
	$w = 3$	$m = 1$	1.04	1.97	1.32	2.68	1.44	2.88
		$m = 1$	1.04	2.00	1.32	2.56	**1.40**	2.85
	$w = 4$	$m = 3$	1.08	**1.96**	**1.28**	2.56	1.44	2.80
		$m = 5$	**1.00**	2.01	**1.28**	2.56	1.44	2.80
		$m = 1$	1.08	2.00	1.37	2.60	1.44	2.80
Frac-w-NAF-numeric		$m = 3$	1.04	2.04	1.36	2.65	1.52	2.80
(Jacobian projective)		$m = 5$	1.08	2.00	1.32	2.60	1.44	2.80
	$w = 5$	$m = 7$	1.13	2.04	1.36	2.61	1.48	2.81
		$m = 9$	1.12	2.01	1.41	2.56	1.52	2.80
		$m = 11$	1.12	2.04	1.44	2.65	1.57	2.80
		$m = 13$	1.12	2.04	1.48	2.60	1.56	2.85

Table 5.5: Times (in ms) of the numeric method (square-root computation times are not included) with addition chains of the scalars chosen randomly

↓ Algorithm / Curve →			P-384			P-521		
			$l = 128$	$l = 192$	$l = 384$	$l = 128$	$l = 256$	$l = 521$
	$w = 3$		2.36	3.41	6.77	2.96	5.73	11.68
w-numeric	$w = 4$		2.20	3.28	6.56	2.84	5.61	11.12
(affine)	$w = 5$		2.40	3.48	6.56	3.13	5.77	11.04
	$w = 3$		2.24	3.32	6.73	2.84	5.60	11.52
w-NAF-numeric	$w = 4$		2.08	3.17	6.49	2.61	5.36	10.96
(affine)	$w = 5$		2.20	3.25	6.33	2.76	5.40	10.72
	$w = 3$	$m = 1$	2.21	3.32	6.60	2.76	5.45	11.32
		$m = 1$	2.08	3.17	6.29	2.64	5.37	10.84
	$w = 4$	$m = 3$	2.12	3.16	6.33	2.64	5.41	10.72
		$m = 5$	2.12	3.13	6.21	2.64	5.37	10.72
		$m = 1$	2.17	3.24	6.28	2.80	5.45	10.72
Frac-w-NAF-numeric		$m = 3$	2.24	3.20	6.33	2.80	5.49	10.76
(affine)		$m = 5$	2.24	3.25	6.37	2.72	5.49	10.76
	$w = 5$	$m = 7$	2.24	3.24	6.41	2.68	5.41	10.72
		$m = 9$	2.24	3.25	6.33	2.76	5.44	10.77
		$m = 11$	2.25	3.24	6.33	2.80	5.44	10.72
		$m = 13$	2.24	3.28	6.32	2.84	5.40	10.72
	$w = 3$		1.88	2.76	5.44	2.48	4.65	9.49
w-numeric	$w = 4$		1.80	2.68	5.21	2.36	4.65	9.01
(Jacobian projective)	$w = 5$		2.00	2.84	5.16	2.64	4.73	8.93
	$w = 3$		1.76	2.61	5.33	2.29	4.44	9.17
w-NAF-numeric	$w = 4$		**1.65**	2.48	5.08	**2.08**	4.32	8.65
(Jacobian projective)	$w = 5$		1.76	2.52	4.97	2.25	4.32	8.57
	$w = 3$	$m = 1$	1.68	2.52	5.13	2.20	4.28	8.93
		$m = 1$	1.68	2.48	4.96	2.13	4.28	8.53
	$w = 4$	$m = 3$	1.68	2.48	4.92	2.16	4.28	**8.37**
		$m = 5$	1.69	**2.40**	4.92	2.12	**4.24**	8.45
		$m = 1$	1.76	2.53	4.93	2.24	4.32	**8.37**
Frac-w-NAF-numeric		$m = 3$	1.76	2.48	**4.88**	2.29	4.32	8.45
(Jacobian projective)		$m = 5$	1.77	2.56	4.92	2.20	4.32	8.41
	$w = 5$	$m = 7$	1.80	2.56	5.00	2.21	4.28	8.44
		$m = 9$	1.80	2.60	4.96	2.24	4.33	8.40
		$m = 11$	1.80	2.57	5.01	2.29	4.37	8.53
		$m = 13$	1.84	2.56	4.97	2.32	4.33	8.41

Table 5.6: Times (in ms) of the Seminumeric method and Montgomery-Ladder with scalars randomly chosen

↓ Algorithm / Curve →			P-192		P-224		P-256	
			$l = 96$	$l = 192$	$l = 112$	$l = 224$	$l = 128$	$l = 256$
w-numeric (affine)	$w = 3$		**1.28**	2.49	**1.48**	3.12	2.00	3.84
	$w = 4$		**1.28**	**2.40**	1.56	**3.01**	1.92	3.76
	$w = 5$		1.45	2.56	1.72	3.04	2.08	3.76
w-NAF-numeric (affine)	$w = 3$		1.32	2.56	1.64	3.16	2.00	3.89
	$w = 4$		**1.28**	2.48	1.56	3.12	1.93	3.85
	$w = 5$		**1.28**	2.48	1.60	3.08	1.96	3.69
Frac-w-NAF-numeric (affine)	$w = 3$	$m = 1$	1.44	2.73	1.76	3.48	2.12	4.21
	$w = 4$	$m = 1$	1.44	2.76	1.72	3.36	2.08	4.16
		$m = 3$	1.45	2.72	1.76	3.33	2.09	4.13
		$m = 5$	1.44	2.72	1.72	3.32	2.08	4.09
	$w = 5$	$m = 1$	1.40	2.72	1.72	3.40	2.13	4.05
		$m = 3$	1.44	2.73	1.76	3.41	2.12	4.09
		$m = 5$	1.44	2.76	1.69	3.40	2.08	4.08
		$m = 7$	1.44	2.73	1.72	3.32	2.08	4.08
		$m = 9$	1.44	2.72	1.80	3.36	2.16	4.08
		$m = 11$	1.49	2.76	1.77	3.33	2.17	4.09
		$m = 13$	1.48	2.76	1.76	3.36	2.16	4.13
w-numeric (Jacobian projective)	$w = 3$		1.32	2.52	1.60	3.33	1.92	3.64
	$w = 4$		1.32	2.48	1.64	3.16	1.84	3.52
	$w = 5$		1.48	2.61	1.81	3.20	1.96	3.60
w-NAF-numeric (Jacobian projective)	$w = 3$		1.36	2.60	1.68	3.28	1.88	3.64
	$w = 4$		1.32	2.53	1.65	3.25	**1.80**	3.61
	$w = 5$		1.32	2.48	1.68	3.17	1.84	**3.48**
Frac-w-NAF-numeric (Jacobian projective)	$w = 3$	$m = 1$	1.48	2.76	1.85	3.61	2.05	4.04
	$w = 4$	$m = 1$	1.48	2.80	1.76	3.48	1.96	3.92
		$m = 3$	1.48	2.73	1.76	3.52	2.00	3.92
		$m = 5$	1.48	2.72	1.81	3.49	2.00	3.92
	$w = 5$	$m = 1$	1.48	2.76	1.84	3.48	2.00	3.84
		$m = 3$	1.49	2.76	1.88	3.52	2.04	3.88
		$m = 5$	1.52	2.76	1.80	3.53	2.01	3.89
		$m = 7$	1.48	2.76	1.80	3.45	1.96	3.85
		$m = 9$	1.52	2.77	1.88	3.48	2.04	3.92
		$m = 11$	1.52	2.76	1.92	3.52	2.04	3.88
		$m = 13$	1.48	2.81	1.92	3.49	2.08	3.88
Montgomery scalar multiplcation (affine)			1.72	3.45	2.12	4.20	2.56	5.17
Montgomery scalar multiplcation (standard projective)			**1.56**	**3.08**	**1.92**	**3.84**	**2.16**	**4.24**

Table 5.7: Times (in ms) of the Seminumeric method and Montgomery-Ladder with scalars randomly chosen

↓ Algorithm / Curve →			P-384			P-521		
			$l = 128$	$l = 192$	$l = 384$	$l = 128$	$l = 256$	$l = 521$
	$w = 3$		2.60	3.72	7.41	3.32	6.37	12.92
w-numeric	$w = 4$		2.44	3.60	7.21	3.12	6.25	12.33
(affine)	$w = 5$		2.65	3.80	7.09	3.40	6.36	12.28
	$w = 3$		2.52	3.80	7.65	3.20	6.32	13.05
w-NAF-numeric	$w = 4$		2.40	3.60	7.37	3.01	6.08	12.40
(affine)	$w = 5$		2.44	3.65	7.21	3.12	6.13	12.17
	$w = 3$	$m = 1$	2.69	4.00	8.01	3.36	6.61	13.56
		$m = 1$	2.61	3.84	7.73	3.21	6.49	13.04
	$w = 4$	$m = 3$	2.60	3.84	7.69	3.20	6.52	12.97
		$m = 5$	2.57	3.77	7.61	3.21	6.49	12.96
		$m = 1$	2.65	3.89	7.68	3.28	6.53	12.92
Frac-w-NAF-numeric		$m = 3$	2.68	3.92	7.69	3.36	6.61	12.96
(affine)		$m = 5$	2.69	3.92	7.68	3.25	6.57	12.92
	$w = 5$	$m = 7$	2.68	3.93	7.73	3.24	6.48	12.96
		$m = 9$	2.69	3.93	7.69	3.37	6.60	12.93
		$m = 11$	2.68	3.92	7.73	3.40	6.57	12.96
		$m = 13$	2.73	3.92	7.69	3.40	6.57	12.92
	$w = 3$		2.24	3.21	6.36	2.84	5.33	10.92
w-numeric	$w = 4$		**2.04**	3.09	6.08	2.64	5.29	10.24
(Jacobian projective)	$w = 5$		2.28	3.24	**6.00**	2.89	5.36	10.24
	$w = 3$		2.16	3.20	6.45	2.69	5.24	10.80
w-NAF-numeric	$w = 4$		**2.04**	**3.00**	6.13	**2.48**	**5.04**	10.20
(Jacobian projective)	$w = 5$		2.08	3.04	6.01	2.64	5.09	**10.08**
	$w = 3$	$m = 1$	2.32	3.45	6.85	2.88	5.57	11.52
		$m = 1$	2.20	3.29	6.56	2.72	5.53	11.08
	$w = 4$	$m = 3$	2.24	3.28	6.57	2.76	5.49	10.92
		$m = 5$	2.24	3.24	6.56	2.72	5.48	10.92
		$m = 1$	2.28	3.32	6.53	2.85	5.56	10.96
Frac-w-NAF-numeric		$m = 3$	2.32	3.33	6.57	2.84	5.56	10.93
(Jacobian projective)		$m = 5$	2.32	3.36	6.57	2.80	5.57	10.92
	$w = 5$	$m = 7$	2.36	3.40	6.60	2.76	5.53	10.96
		$m = 9$	2.36	3.36	6.57	2.84	5.61	10.92
		$m = 11$	2.36	3.41	6.60	2.89	5.52	11.00
		$m = 13$	2.40	3.40	6.61	2.92	5.56	10.92
Montgomery scalar multiplcation (affine)			3.48	5.20	10.36	4.53	9.09	18.61
Montgomery scalar multiplcation (standard projective)			**2.53**	**3.76**	**7.61**	**3.17**	**6.45**	**13.08**

Table 5.8: Times (in ms) of the Seminumeric method and Montgomery-ladder method with addition chains of the scalars chosen randomly

↓ Algorithm / Curve →			P-192		P-224		P-256	
			$l = 96$	$l = 192$	$l = 112$	$l = 224$	$l = 128$	$l = 256$
	$w = 3$		1.24	2.32	1.48	3.04	1.92	3.76
w-numeric	$w = 4$		1.24	2.36	1.52	2.96	1.88	3.64
(affine)	$w = 5$		1.36	2.48	1.68	2.97	2.05	3.73
	$w = 3$		1.20	2.36	1.52	3.01	1.84	3.61
w-NAF-numeric	$w = 4$		**1.16**	2.28	1.44	2.92	1.76	3.57
(affine)	$w = 5$		**1.16**	2.28	1.48	2.84	1.80	3.44
	$w = 3$	$m = 1$	**1.16**	2.24	1.52	2.88	1.84	3.61
		$m = 1$	1.20	2.28	1.45	2.84	1.76	3.53
	$w = 4$	$m = 3$	1.20	2.24	1.44	2.85	1.80	3.52
		$m = 5$	1.20	2.24	**1.40**	**2.80**	1.77	3.48
		$m = 1$	**1.16**	**2.20**	1.49	2.84	1.80	3.48
Frac-w-NAF-numeric		$m = 3$	1.20	2.24	1.48	2.84	1.81	3.44
(affine)		$m = 5$	1.20	2.24	1.48	2.84	1.76	3.49
	$w = 5$	$m = 7$	1.20	2.28	1.44	2.85	1.76	3.48
		$m = 9$	1.24	2.29	1.52	**2.80**	1.81	3.48
		$m = 11$	1.24	2.28	1.52	2.85	1.84	3.49
		$m = 13$	1.24	2.29	1.56	2.88	1.84	3.52
	$w = 3$		1.28	2.44	1.52	3.24	1.84	3.56
w-numeric	$w = 4$		1.28	2.40	1.60	3.08	1.77	3.44
(Jacobian projective)	$w = 5$		1.45	2.52	1.76	3.16	1.92	3.53
	$w = 3$		1.24	2.41	1.56	3.08	1.72	3.36
w-NAF-numeric	$w = 4$		1.20	2.37	1.52	2.97	1.68	3.32
(Jacobian projective)	$w = 5$		1.20	2.37	1.52	2.97	1.68	**3.16**
	$w = 3$	$m = 1$	1.20	2.28	1.52	3.01	1.69	3.32
		$m = 1$	**1.16**	2.32	1.48	2.92	**1.64**	3.32
	$w = 4$	$m = 3$	1.21	2.33	1.48	2.96	1.68	3.29
		$m = 5$	1.20	2.32	1.52	2.96	1.68	3.28
		$m = 1$	1.20	2.24	1.52	2.92	1.72	3.17
Frac-w-NAF-numeric		$m = 3$	1.20	2.33	1.52	2.97	1.72	3.20
(Jacobian projective)		$m = 5$	1.24	2.28	1.48	2.96	**1.64**	3.20
	$w = 5$	$m = 7$	1.25	2.28	1.53	2.92	1.68	3.21
		$m = 9$	1.24	2.32	1.56	2.96	1.72	3.24
		$m = 11$	1.24	2.32	1.56	2.96	1.72	3.24
		$m = 13$	1.28	2.32	1.61	2.93	1.77	3.25
Montgomery scalar multiplcation (affine)			1.73	3.44	2.13	4.25	2.60	5.21
Montgomery scalar multiplcation (standard projective)			**1.56**	**3.09**	**1.96**	**3.96**	**2.16**	**4.36**

Table 5.9: Times (in ms) of the Seminumeric method and Montgomery-ladder method with addition chains of the scalars chosen randomly

↓ Algorithm / Curve →			P-384			P-521		
			$l = 128$	$l = 192$	$l = 384$	$l = 128$	$l = 256$	$l = 521$
	$w = 3$		2.52	3.68	7.33	3.24	6.17	12.60
w-numeric	$w = 4$		2.41	3.48	7.01	3.08	6.09	11.96
(affine)	$w = 5$		2.56	3.72	7.01	3.37	6.29	12.08
	$w = 3$		2.44	3.56	7.17	3.00	6.04	12.41
w-NAF-numeric	$w = 4$		2.24	3.40	6.88	2.85	5.80	11.76
(affine)	$w = 5$		2.29	3.40	6.72	2.96	5.80	11.60
	$w = 3$	$m = 1$	2.37	3.48	7.01	3.00	5.85	12.08
		$m = 1$	2.25	3.32	6.80	2.84	5.77	11.65
	$w = 4$	$m = 3$	2.28	3.33	6.76	2.84	5.77	11.48
		$m = 5$	2.24	3.28	6.77	2.84	5.73	11.48
		$m = 1$	2.32	3.40	6.72	2.96	5.89	11.52
Frac-w-NAF-numeric		$m = 3$	2.37	3.44	6.72	2.96	5.85	11.53
(affine)		$m = 5$	2.36	3.44	6.73	2.92	5.85	11.45
	$w = 5$	$m = 7$	2.41	3.44	6.65	2.88	5.81	11.56
		$m = 9$	2.36	3.45	6.76	3.00	5.85	11.48
		$m = 11$	2.36	3.44	6.76	3.01	5.81	11.52
		$m = 13$	2.44	3.44	6.73	3.00	5.84	11.44
	$w = 3$		2.12	3.13	6.12	2.72	5.21	10.52
w-numeric	$w = 4$		2.04	3.01	5.89	2.60	5.09	10.13
(Jacobian projective)	$w = 5$		2.24	3.17	5.93	2.84	5.21	10.00
	$w = 3$		2.00	2.96	5.97	2.57	4.96	10.20
w-NAF-numeric	$w = 4$		**1.88**	2.80	5.77	**2.36**	4.80	9.64
(Jacobian projective)	$w = 5$		1.96	2.84	5.61	2.49	4.80	9.57
	$w = 3$	$m = 1$	2.00	2.93	5.88	2.48	4.84	10.12
		$m = 1$	**1.88**	2.84	5.73	2.37	**4.76**	9.57
	$w = 4$	$m = 3$	1.92	2.80	5.65	2.40	4.80	9.49
		$m = 5$	1.89	**2.73**	**5.60**	2.37	**4.76**	9.41
		$m = 1$	2.00	2.85	5.61	2.48	4.84	9.49
Frac-w-NAF-numeric		$m = 3$	1.96	2.80	5.61	2.49	4.84	**9.40**
(Jacobian projective)		$m = 5$	2.00	2.85	5.64	2.40	4.84	9.45
	$w = 5$	$m = 7$	2.00	2.88	5.65	2.41	4.80	9.49
		$m = 9$	2.04	2.84	5.61	2.48	4.88	9.53
		$m = 11$	2.01	2.93	5.65	2.52	4.81	9.49
		$m = 13$	2.04	2.92	**5.60**	2.56	4.81	9.45
Montgomery scalar multiplcation (affine)			3.44	5.20	10.44	4.53	9.05	18.49
Montgomery scalar multiplcation (standard projective)			**2.52**	**3.81**	**7.53**	**3.20**	**6.41**	**13.04**

Table 5.10: Times (in ms) of the square-root computation

↓ Algorithm / Curve →			P-192	P-224	P256	P-384	P-521
w-numeric	$w=3$		0.16	–	**0.16**	0.40	0.56
	$w=4$		0.16	–	**0.16**	0.36	**0.52**
	$w=5$		0.17	–	**0.16**	**0.32**	**0.52**
w-NAF-numeric	$w=3$		0.20	–	0.32	0.49	0.84
	$w=4$		0.20	–	0.32	0.52	0.85
	$w=5$		0.24	–	0.28	0.52	0.84
Frac-w-NAF-numeric	$w=3$	$m=1$	0.20	–	0.32	0.52	0.84
	$w=4$	$m=1$	0.20	–	0.28	0.52	0.80
		$m=3$	0.24	–	0.28	0.56	0.84
		$m=5$	0.20	–	0.32	0.52	0.84
	$w=5$	$m=1$	0.20	–	0.32	0.52	0.84
		$m=3$	0.24	–	0.32	0.56	0.84
		$m=5$	0.24	–	0.32	0.52	0.84
		$m=7$	0.20	–	0.32	0.52	0.84
		$m=9$	0.20	–	0.37	0.52	0.88
		$m=11$	0.24	–	0.32	0.56	0.84
		$m=13$	0.24	–	0.32	0.52	0.85
Tonelli-Shanks			**0.15**	**5.43**	0.19	0.39	0.54

Table 5.11: Speedups obtained by different randomized batch-verification algorithms with scalars randomly chosen

(Fixed-base double scalar multiplication is *not* used)

Batch Verification Algorithm	Randomization Algorithm	t	P-192		P-224		P-256	
			None*	$l = 96$	None*	$l = 112$	None*	$l = 128$
Algorithm N	Numeric	3	2.47	1.36	0.61	0.51	2.62	1.47
		4	2.87	1.47	0.63	0.52	3.15	1.63
		5	2.92	1.49	0.64	0.53	3.35	1.68
		6	2.45	1.35	0.62	0.51	3.00	1.59
		7	1.70	1.09	0.56	0.48	2.23	1.34
		8	1.04	0.77	0.48	0.41	1.43	1.00
Algorithm N′	Numeric	3	2.63	1.51	0.62	0.53	2.74	1.59
		4	3.37	1.81	0.65	0.56	3.54	1.88
		5	4.05	1.93	0.67	0.57	4.31	2.04
		6	4.68	2.13	0.69	0.59	5.03	2.23
		7	5.26	2.19	0.70	0.59	5.70	2.32
		8	5.82	2.33	0.71	0.60	6.36	2.46
Algorithm S2′	Seminumeric	3	2.95	1.40	2.95	1.45	2.96	1.47
		4	3.82	1.56	3.85	1.64	3.87	1.67
		5	4.53	1.67	4.59	1.76	4.66	1.80
		6	5.05	1.74	5.16	1.84	5.30	1.89
		7	4.95	1.73	5.16	1.84	5.45	1.90
		8	4.03	1.60	4.31	1.72	4.76	1.81
Algorithm S3	Seminumeric	3	2.93	1.39	2.94	1.45	2.95	1.47
		4	3.78	1.56	3.81	1.63	3.83	1.66
		5	4.46	1.66	4.53	1.75	4.59	1.79
		6	5.01	1.73	5.13	1.83	5.26	1.88
		7	5.06	1.74	5.27	1.85	5.51	1.91
		8	4.89	1.72	5.17	1.84	5.54	1.92
Algorithm $S3_{gcd}$	Seminumeric	3	2.94	1.39	2.94	1.45	2.95	1.47
		4	3.80	1.56	3.83	1.63	3.85	1.66
		5	4.54	1.67	4.60	1.76	4.66	1.80
		6	5.18	1.75	5.26	1.85	5.39	1.90
		7	5.49	1.79	5.65	1.89	5.86	1.95
		8	5.41	1.78	5.65	1.90	6.00	1.97
Algorithm SP	Seminumeric	3	2.95	1.40	2.95	1.45	2.93	1.47
		4	3.86	1.57	3.87	1.64	3.89	1.67
		5	4.71	1.69	4.73	1.78	4.77	1.81
		6	5.32	1.77	5.36	1.86	5.47	1.91
		7	5.68	1.81	5.76	1.91	5.94	1.96
		8	5.55	1.79	5.69	1.90	6.00	1.97
		9	4.96	1.73	5.13	1.83	5.53	1.91
		10	3.80	1.56	3.93	1.65	4.48	1.77
Algorithm SP_{gcd}	Seminumeric	3	3.00	1.41	2.95	1.45	2.93	1.47
		4	3.87	1.57	3.87	1.64	3.90	1.67
		5	4.73	1.70	4.76	1.78	4.80	1.82
		6	5.35	1.77	5.41	1.87	5.50	1.91
		7	5.87	1.83	5.96	1.93	6.11	1.98
		8	5.71	1.81	5.80	1.91	6.15	1.98
		9	5.55	1.79	5.65	1.90	6.13	1.98
		10	4.09	1.61	4.23	1.70	4.83	1.82

* without randomization

Table 5.12: Speedups obtained by different randomized batch-verification algorithms with scalars randomly chosen

(Fixed-base double scalar multiplication is *not* used)

Batch Verification Algorithm	Randomization Algorithm	t	P-384			P-521		
			None*	$l = 128$	$l = 192$	None*	$l = 128$	$l = 256$
Algorithm N	Numeric	3	2.65	1.78	1.53	2.68	1.95	1.52
		4	3.27	2.04	1.72	3.34	2.28	1.71
		5	3.63	2.17	1.82	3.78	2.48	1.82
		6	3.46	2.11	1.77	3.80	2.49	1.83
		7	2.85	1.87	1.60	3.28	2.26	1.70
		8	1.99	1.46	1.29	2.41	1.81	1.43
Algorithm N′	Numeric	3	2.72	1.86	1.76	2.73	2.02	1.84
		4	3.52	2.23	1.87	3.53	2.45	1.90
		5	4.27	2.48	2.38	4.29	2.77	2.57
		6	4.99	2.74	2.22	5.01	3.08	2.26
		7	5.65	2.90	2.80	5.68	3.30	3.09
		8	6.29	3.09	2.45	6.32	3.54	2.49
Algorithm S2′	Seminumeric	4	3.90	2.15	1.78	3.92	2.45	1.76
		5	4.77	2.39	1.94	4.82	2.77	1.93
		6	5.52	2.57	2.05	5.63	3.02	2.04
		7	5.92	2.65	2.10	6.16	3.17	2.11
		8	5.57	2.58	2.06	5.99	3.12	2.09
Algorithm S3	Seminumeric	3	2.96	1.83	1.55	2.97	2.04	1.54
		4	3.88	2.14	1.77	3.91	2.44	1.76
		5	4.72	2.38	1.93	4.78	2.76	1.92
		6	5.48	2.56	2.04	5.60	3.01	2.04
		7	5.93	2.65	2.10	6.16	3.17	2.11
		8	6.20	2.70	2.14	6.57	3.27	2.15
Algorithm $S3_{gcd}$	Seminumeric	3	2.96	1.83	1.55	2.97	2.04	1.54
		4	3.89	2.15	1.77	3.92	2.45	1.76
		5	4.77	2.39	1.94	4.82	2.77	1.93
		6	5.58	2.58	2.06	5.67	3.03	2.05
		7	6.22	2.71	2.14	6.40	3.23	2.14
		8	6.58	2.77	2.18	6.87	3.34	2.19
Algorithm SP	Seminumeric	3	2.97	1.83	1.55	2.97	2.04	1.54
		4	3.92	2.16	1.78	3.93	2.45	1.77
		5	4.83	2.41	1.95	4.86	2.78	1.93
		6	5.60	2.58	2.06	5.67	3.03	2.05
		7	6.17	2.70	2.13	6.31	3.20	2.13
		8	6.41	2.74	2.16	6.62	3.28	2.16
		9	6.11	2.69	2.13	6.40	3.23	2.14
		10	5.23	2.50	2.01	5.60	3.01	2.04
Algorithm SP_{gcd}	Seminumeric	3	2.97	1.83	1.55	2.97	2.04	1.54
		4	3.92	2.16	1.78	3.93	2.45	1.77
		5	4.85	2.41	1.95	4.88	2.79	1.93
		6	5.63	2.59	2.06	5.69	3.04	2.05
		7	6.32	2.73	2.15	6.43	3.24	2.14
		8	6.57	2.77	2.18	6.74	3.31	2.17
		9	6.71	2.80	2.19	6.96	3.37	2.20
		10	5.62	2.59	2.06	5.99	3.12	2.09

* without randomization

Table 5.13: Speedups obtained by different randomized batch-verification algorithms with addition chains of the scalars randomly chosen

(Fixed-base double scalar multiplication is *not* used)

Batch Verification Algorithm	Randomization Algorithm	t	P-192		P-224		P-256	
			None*	$l = 96$	None*	$l = 112$	None*	$l = 128$
Algorithm N	Numeric	3	2.47	1.43	0.61	0.51	2.62	1.54
		4	2.87	1.55	0.63	0.53	3.15	1.72
		5	2.92	1.57	0.64	0.53	3.35	1.77
		6	2.45	1.42	0.62	0.52	3.00	1.67
		7	1.70	1.13	0.56	0.48	2.23	1.40
		8	1.04	0.79	0.48	0.42	1.43	1.03
Algorithm N′	Numeric	3	2.63	1.54	0.62	0.53	2.74	1.61
		4	3.37	1.81	0.65	0.56	3.54	1.88
		5	4.05	1.96	0.67	0.57	4.31	2.06
		6	4.68	2.13	0.69	0.59	5.03	2.23
		7	5.26	2.21	0.70	0.59	5.70	2.35
		8	5.82	2.33	0.71	0.60	6.36	2.46
Algorithm S2′	Seminumeric	3	2.95	1.47	2.95	1.49	2.96	1.54
		4	3.82	1.66	3.85	1.69	3.87	1.76
		5	4.53	1.78	4.59	1.82	4.66	1.90
		6	5.05	1.85	5.16	1.90	5.30	2.00
		7	4.95	1.84	5.16	1.90	5.45	2.02
		8	4.03	1.69	4.31	1.77	4.76	1.92
Algorithm S3	Seminumeric	3	2.93	1.46	2.94	1.49	2.95	1.54
		4	3.78	1.65	3.81	1.68	3.83	1.75
		5	4.46	1.77	4.53	1.81	4.59	1.89
		6	5.01	1.85	5.13	1.90	5.26	1.99
		7	5.06	1.85	5.27	1.92	5.51	2.03
		8	4.89	1.83	5.17	1.90	5.54	2.03
Algorithm $S3_{gcd}$	Seminumeric	3	2.94	1.47	2.94	1.49	2.95	1.54
		4	3.80	1.65	3.83	1.69	3.85	1.75
		5	4.54	1.78	4.60	1.82	4.66	1.90
		6	5.18	1.87	5.26	1.92	5.39	2.01
		7	5.49	1.91	5.65	1.97	5.86	2.08
		8	5.41	1.90	5.65	1.97	6.00	2.09
Algorithm SP	Seminumeric	3	2.95	1.47	2.95	1.49	2.93	1.53
		4	3.86	1.66	3.87	1.69	3.89	1.76
		5	4.71	1.80	4.73	1.84	4.77	1.92
		6	5.32	1.89	5.36	1.93	5.47	2.02
		7	5.68	1.93	5.76	1.98	5.94	2.09
		8	5.55	1.91	5.69	1.97	6.00	2.09
		9	4.96	1.84	5.13	1.90	5.53	2.03
		10	3.80	1.65	3.93	1.71	4.48	1.87
Algorithm SP_{gcd}	Seminumeric	3	3.00	1.48	2.95	1.49	2.93	1.53
		4	3.87	1.67	3.87	1.69	3.90	1.76
		5	4.73	1.81	4.76	1.85	4.80	1.92
		6	5.35	1.89	5.41	1.94	5.50	2.03
		7	5.87	1.95	5.96	2.00	6.11	2.11
		8	5.71	1.93	5.80	1.98	6.15	2.11
		9	5.55	1.91	5.65	1.97	6.13	2.11
		10	4.09	1.70	4.23	1.76	4.83	1.93

* without randomization

Table 5.14: Speedups obtained by different randomized batch-verification algorithms with addition chains of the scalars randomly chosen

(Fixed-base double scalar multiplication is *not* used)

Batch Verification Algorithm	Randomization Algorithm	t	P-384			P-521		
			None*	$l = 128$	$l = 192$	None*	$l = 128$	$l = 256$
Algorithm N	Numeric	3	2.65	1.83	1.61	2.68	1.99	1.57
		4	3.27	2.11	1.81	3.34	2.34	1.78
		5	3.63	2.25	1.92	3.78	2.54	1.90
		6	3.46	2.18	1.87	3.80	2.55	1.90
		7	2.85	1.93	1.68	3.28	2.31	1.76
		8	1.99	1.49	1.34	2.41	1.84	1.48
Algorithm N′	Numeric	3	2.72	1.87	1.79	2.73	2.03	1.86
		4	3.52	2.23	1.87	3.53	2.45	1.90
		5	4.27	2.50	2.41	4.29	2.79	2.60
		6	4.99	2.74	2.22	5.01	3.08	2.26
		7	5.65	2.92	2.83	5.68	3.32	3.12
		8	6.29	3.09	2.45	6.32	3.54	2.49
Algorithm S2′	Seminumeric	3	2.97	1.89	1.62	2.97	2.07	1.58
		4	3.90	2.23	1.87	3.92	2.49	1.82
		5	4.77	2.49	2.05	4.82	2.83	1.99
		6	5.52	2.68	2.17	5.63	3.09	2.12
		7	5.92	2.77	2.23	6.16	3.24	2.19
		8	5.57	2.69	2.18	5.99	3.20	2.17
Algorithm S3	Seminumeric	3	2.96	1.89	1.62	2.97	2.07	1.58
		4	3.88	2.22	1.86	3.91	2.49	1.82
		5	4.72	2.47	2.04	4.78	2.82	1.99
		6	5.48	2.67	2.17	5.60	3.08	2.11
		7	5.93	2.77	2.23	6.16	3.24	2.19
		8	6.20	2.83	2.27	6.57	3.35	2.24
Algorithm $S3_{gcd}$	Seminumeric	3	2.96	1.89	1.62	2.97	2.07	1.58
		4	3.89	2.23	1.87	3.92	2.49	1.82
		5	4.77	2.49	2.05	4.82	2.83	1.99
		6	5.58	2.69	2.18	5.67	3.10	2.12
		7	6.22	2.83	2.27	6.40	3.31	2.22
		8	6.58	2.91	2.32	6.87	3.43	2.27
Algorithm SP	Seminumeric	3	2.97	1.89	1.62	2.97	2.07	1.58
		4	3.92	2.24	1.87	3.93	2.50	1.82
		5	4.83	2.51	2.06	4.86	2.84	2.00
		6	5.60	2.70	2.18	5.67	3.10	2.12
		7	6.17	2.82	2.27	6.31	3.28	2.21
		8	6.41	2.87	2.30	6.62	3.36	2.24
		9	6.11	2.81	2.26	6.40	3.31	2.22
		10	5.23	2.61	2.13	5.60	3.08	2.11
Algorithm SP_{gcd}	Seminumeric	3	2.97	1.89	1.62	2.97	2.07	1.58
		4	3.92	2.24	1.87	3.93	2.50	1.82
		5	4.85	2.51	2.06	4.88	2.85	2.00
		6	5.63	2.70	2.19	5.69	3.11	2.13
		7	6.32	2.85	2.29	6.43	3.32	2.22
		8	6.57	2.90	2.32	6.74	3.40	2.26
		9	6.71	2.93	2.34	6.96	3.45	2.28
		10	5.62	2.70	2.19	5.99	3.19	2.17

* without randomization

Table 5.15: Speedups obtained by different randomized batch-verification algorithms with scalars randomly chosen

(Fixed-base double scalar multiplication is *used*)

Batch Verification Algorithm	Randomization Algorithm	t	P-192		P-224		P-256	
			None*	$l = 96$	None*	$l = 112$	None*	$l = 128$
Algorithm N	Numeric	3	2.25	1.24	0.55	0.46	2.29	1.29
		4	2.34	1.20	0.51	0.42	2.47	1.28
		5	2.23	1.13	0.47	0.39	2.44	1.23
		6	1.77	0.98	0.44	0.36	2.07	1.10
		7	1.19	0.76	0.38	0.32	1.49	0.90
		8	0.70	0.52	0.31	0.27	0.92	0.65
Algorithm N′	Numeric	3	2.40	1.38	0.55	0.47	2.40	1.39
		4	2.75	1.47	0.52	0.45	2.77	1.48
		5	3.08	1.47	0.50	0.43	3.14	1.48
		6	3.39	1.54	0.49	0.41	3.48	1.54
		7	3.67	1.53	0.48	0.40	3.80	1.55
		8	3.95	1.58	0.47	0.40	4.11	1.59
Algorithm S2′	Seminumeric	3	2.68	1.27	2.65	1.30	2.58	1.29
		4	3.12	1.28	3.09	1.31	3.03	1.31
		5	3.45	1.27	3.42	1.31	3.39	1.31
		6	3.66	1.26	3.65	1.30	3.66	1.30
		7	3.46	1.21	3.51	1.25	3.63	1.27
		8	2.73	1.08	2.84	1.13	3.08	1.17
Algorithm $S3_{gcd}$	Seminumeric	3	2.67	1.27	2.64	1.30	2.58	1.28
		4	3.08	1.27	3.06	1.31	3.00	1.30
		5	3.40	1.27	3.38	1.31	3.34	1.30
		6	3.63	1.25	3.63	1.30	3.64	1.30
		7	3.53	1.21	3.59	1.26	3.67	1.27
		8	3.31	1.16	3.42	1.21	3.58	1.24
Algorithm $S3_{gcd}$	Seminumeric	3	2.68	1.27	2.64	1.30	2.58	1.28
		4	3.10	1.27	3.07	1.31	3.02	1.30
		5	3.46	1.27	3.43	1.31	3.40	1.31
		6	3.75	1.27	3.73	1.31	3.73	1.31
		7	3.84	1.25	3.85	1.29	3.91	1.30
		8	3.67	1.21	3.74	1.25	3.88	1.27
Algorithm SP	Seminumeric	3	2.68	1.27	2.65	1.30	2.56	1.28
		4	3.15	1.28	3.11	1.32	3.05	1.31
		5	3.59	1.29	3.53	1.33	3.48	1.32
		6	3.86	1.28	3.80	1.32	3.78	1.32
		7	3.96	1.26	3.92	1.30	3.96	1.31
		8	3.76	1.22	3.76	1.25	3.88	1.27
		9	3.29	1.14	3.31	1.18	3.49	1.21
		10	2.47	1.01	2.49	1.05	2.77	1.10
Algorithm SP_{gcd}	Seminumeric	3	2.73	1.28	2.65	1.30	2.56	1.28
		4	3.16	1.28	3.11	1.32	3.05	1.31
		5	3.61	1.29	3.56	1.33	3.50	1.33
		6	3.88	1.28	3.83	1.32	3.80	1.32
		7	4.10	1.27	4.06	1.31	4.07	1.32
		8	3.87	1.23	3.83	1.26	3.98	1.28
		9	3.67	1.19	3.65	1.22	3.87	1.25
		10	2.66	1.04	2.67	1.08	2.99	1.13

* without randomization

Table 5.16: Speedups obtained by different randomized batch-verification algorithms with scalars randomly chosen

(Fixed-base double scalar multiplication is *used*)

Batch Verification Algorithm	Randomization Algorithm	t	P-384			P-521		
			None*	$l = 128$	$l = 192$	None*	$l = 128$	$l = 256$
Algorithm N	Numeric	3	2.33	1.57	1.35	2.30	1.68	1.31
		4	2.57	1.60	1.35	2.62	1.79	1.34
		5	2.65	1.58	1.32	2.76	1.81	1.33
		6	2.39	1.46	1.23	2.63	1.72	1.26
		7	1.90	1.24	1.06	2.17	1.49	1.12
		8	1.28	0.94	0.83	1.54	1.16	0.92
Algorithm N′	Numeric	3	2.40	1.63	1.55	2.35	1.73	1.58
		4	2.77	1.75	1.47	2.77	1.92	1.49
		5	3.12	1.81	1.73	3.13	2.02	1.87
		6	3.45	1.89	1.53	3.46	2.13	1.56
		7	3.75	1.93	1.86	3.77	2.19	2.05
		8	4.05	1.99	1.57	4.06	2.27	1.60
Algorithm S2′	Seminumeric	3	2.61	1.61	1.37	2.55	1.75	1.32
		4	3.07	1.69	1.40	3.07	1.92	1.38
		5	3.47	1.74	1.41	3.52	2.02	1.41
		6	3.81	1.77	1.42	3.89	2.09	1.41
		7	3.93	1.76	1.40	4.08	2.10	1.40
		8	3.59	1.66	1.32	3.84	2.00	1.34
Algorithm S3	Seminumeric	3	2.61	1.61	1.37	2.55	1.75	1.32
		4	3.05	1.69	1.39	3.06	1.91	1.38
		5	3.44	1.73	1.41	3.49	2.01	1.40
		6	3.79	1.77	1.41	3.87	2.08	1.41
		7	3.94	1.76	1.40	4.08	2.10	1.40
		8	3.99	1.74	1.38	4.21	2.10	1.38
Algorithm $S3_{gcd}$	Seminumeric	3	2.61	1.61	1.37	2.55	1.75	1.32
		4	3.06	1.69	1.39	3.07	1.92	1.38
		5	3.48	1.74	1.41	3.52	2.02	1.41
		6	3.85	1.78	1.42	3.92	2.09	1.41
		7	4.13	1.80	1.42	4.24	2.14	1.42
		8	4.24	1.79	1.40	4.41	2.15	1.40
Algorithm SP	Seminumeric	3	2.61	1.61	1.37	2.55	1.75	1.33
		4	3.08	1.69	1.40	3.08	1.92	1.38
		5	3.52	1.75	1.42	3.55	2.03	1.41
		6	3.87	1.78	1.42	3.91	2.09	1.41
		7	4.10	1.79	1.42	4.18	2.12	1.41
		8	4.12	1.77	1.39	4.24	2.11	1.39
		9	3.84	1.69	1.33	4.00	2.02	1.34
		10	3.22	1.54	1.24	3.43	1.84	1.25
Algorithm SP_{gcd}	Seminumeric	3	2.61	1.61	1.37	2.55	1.75	1.33
		4	3.08	1.69	1.40	3.08	1.92	1.38
		5	3.54	1.76	1.42	3.56	2.04	1.41
		6	3.89	1.79	1.43	3.93	2.10	1.42
		7	4.20	1.81	1.43	4.26	2.14	1.42
		8	4.23	1.78	1.40	4.33	2.13	1.39
		9	4.21	1.76	1.38	4.35	2.10	1.37
		10	3.46	1.59	1.27	3.66	1.91	1.28

* without randomization

Table 5.17: Speedups obtained by different randomized batch-verification algorithms with addition chains of the scalars randomly chosen

(Fixed-base double scalar multiplication is *used*)

Batch Verification Algorithm	Randomization Algorithm	t	P-192		P-224		P-256	
			None*	$l = 96$	None*	$l = 112$	None*	$l = 128$
Algorithm N	Nnumeric	3	2.25	1.30	0.55	0.46	2.29	1.35
		4	2.34	1.27	0.51	0.43	2.47	1.34
		5	2.23	1.20	0.47	0.40	2.44	1.29
		6	1.77	1.03	0.44	0.37	2.07	1.15
		7	1.19	0.79	0.38	0.33	1.49	0.93
		8	0.70	0.54	0.31	0.27	0.92	0.67
Algorithm N′	Nnumeric	3	2.40	1.40	0.55	0.47	2.40	1.41
		4	2.75	1.47	0.52	0.45	2.77	1.48
		5	3.08	1.49	0.50	0.43	3.14	1.50
		6	3.39	1.54	0.49	0.41	3.48	1.54
		7	3.67	1.54	0.48	0.40	3.80	1.56
		8	3.95	1.58	0.47	0.40	4.11	1.59
Algorithm S2′	Seminumeric	3	2.68	1.34	2.65	1.34	2.58	1.35
		4	3.12	1.35	3.09	1.36	3.03	1.37
		5	3.45	1.35	3.42	1.36	3.39	1.39
		6	3.66	1.34	3.65	1.35	3.66	1.38
		7	3.46	1.28	3.51	1.30	3.63	1.35
		8	2.73	1.15	2.84	1.17	3.08	1.24
Algorithm S3	Seminumeric	3	2.67	1.33	2.64	1.34	2.58	1.34
		4	3.08	1.35	3.06	1.35	3.00	1.37
		5	3.40	1.34	3.38	1.35	3.34	1.38
		6	3.63	1.34	3.63	1.34	3.64	1.38
		7	3.53	1.29	3.59	1.31	3.67	1.35
		8	3.31	1.24	3.42	1.26	3.58	1.31
Algorithm $S3_{gcd}$	Seminumeric	3	2.68	1.33	2.64	1.34	2.58	1.34
		4	3.10	1.35	3.07	1.35	3.02	1.37
		5	3.46	1.35	3.43	1.36	3.40	1.39
		6	3.75	1.35	3.73	1.36	3.73	1.39
		7	3.84	1.33	3.85	1.34	3.91	1.38
		8	3.67	1.29	3.74	1.30	3.88	1.35
Algorithm SP	Seminumeric	3	2.68	1.34	2.65	1.34	2.56	1.34
		4	3.15	1.36	3.11	1.36	3.05	1.38
		5	3.59	1.37	3.53	1.37	3.48	1.40
		6	3.86	1.37	3.80	1.37	3.78	1.40
		7	3.96	1.35	3.92	1.35	3.96	1.39
		8	3.76	1.30	3.76	1.30	3.88	1.35
		9	3.29	1.22	3.31	1.22	3.49	1.28
		10	2.47	1.07	2.49	1.08	2.77	1.16
Algorithm SP_{gcd}	Seminumeric	3	2.73	1.35	2.65	1.34	2.56	1.34
		4	3.16	1.36	3.11	1.36	3.05	1.38
		5	3.61	1.38	3.56	1.38	3.50	1.40
		6	3.88	1.37	3.83	1.37	3.80	1.40
		7	4.10	1.36	4.06	1.36	4.07	1.40
		8	3.87	1.31	3.83	1.31	3.98	1.36
		9	3.67	1.27	3.65	1.27	3.87	1.33
		10	2.66	1.11	2.67	1.11	2.99	1.20

* without randomization

Table 5.18: Speedups obtained by different randomized batch-verification algorithms with addition chains of the scalars randomly chosen

(Fixed-base double scalar multiplication is *used*)

Batch Verification Algorithm	Randomization Algorithm	t	P-384			P-521		
			None*	$l = 128$	$l = 192$	None*	$l = 128$	$l = 256$
Algorithm N	Numeric	3	2.33	1.61	1.41	2.30	1.71	1.35
		4	2.57	1.66	1.43	2.62	1.83	1.40
		5	2.65	1.64	1.40	2.76	1.86	1.39
		6	2.39	1.51	1.29	2.63	1.76	1.31
		7	1.90	1.28	1.12	2.17	1.53	1.17
		8	1.28	0.96	0.86	1.54	1.18	0.95
Algorithm N′	Numeric	3	2.40	1.65	1.58	2.35	1.75	1.60
		4	2.77	1.75	1.47	2.77	1.92	1.49
		5	3.12	1.83	1.76	3.13	2.04	1.89
		6	3.45	1.89	1.53	3.46	2.13	1.56
		7	3.75	1.94	1.88	3.77	2.20	2.07
		8	4.05	1.99	1.57	4.06	2.27	1.60
Algorithm S2′	Seminumeric	3	2.61	1.66	1.43	2.55	1.78	1.36
		4	3.07	1.75	1.47	3.07	1.95	1.43
		5	3.47	1.81	1.49	3.52	2.07	1.45
		6	3.81	1.85	1.50	3.89	2.13	1.46
		7	3.93	1.84	1.48	4.08	2.15	1.45
		8	3.59	1.73	1.40	3.84	2.05	1.39
Algorithm S3	Seminumeric	3	2.61	1.66	1.43	2.55	1.78	1.36
		4	3.05	1.75	1.46	3.06	1.95	1.42
		5	3.44	1.80	1.48	3.49	2.06	1.45
		6	3.79	1.84	1.50	3.87	2.13	1.46
		7	3.94	1.84	1.48	4.08	2.15	1.45
		8	3.99	1.82	1.46	4.21	2.15	1.44
Algorithm $S3_{gcd}$	Seminumeric	3	2.61	1.66	1.43	2.55	1.78	1.36
		4	3.06	1.75	1.47	3.07	1.95	1.43
		5	3.48	1.81	1.49	3.52	2.07	1.45
		6	3.85	1.86	1.51	3.92	2.14	1.47
		7	4.13	1.88	1.51	4.24	2.19	1.47
		8	4.24	1.87	1.49	4.41	2.20	1.46
Algorithm SP	Seminumeric	3	2.61	1.66	1.43	2.55	1.78	1.36
		4	3.08	1.76	1.47	3.08	1.96	1.43
		5	3.52	1.83	1.50	3.55	2.08	1.46
		6	3.87	1.86	1.51	3.91	2.14	1.47
		7	4.10	1.87	1.51	4.18	2.17	1.46
		8	4.12	1.85	1.48	4.24	2.16	1.44
		9	3.84	1.76	1.42	4.00	2.07	1.39
		10	3.22	1.60	1.31	3.43	1.89	1.29
Algorithm SP_{gcd}	Seminumeric	3	2.61	1.66	1.43	2.55	1.78	1.36
		4	3.08	1.76	1.47	3.08	1.96	1.43
		5	3.54	1.83	1.50	3.56	2.08	1.46
		6	3.89	1.87	1.51	3.93	2.15	1.47
		7	4.20	1.90	1.52	4.26	2.20	1.47
		8	4.23	1.87	1.49	4.33	2.18	1.45
		9	4.21	1.84	1.47	4.35	2.16	1.43
		10	3.46	1.66	1.35	3.66	1.95	1.33

* without randomization

$3.48 + 0.16 = 3.64$ and $10.08 + 0.52 = 10.60$ for the two curves P-256 and P-521. In this case of randomly chosen addition chains, the best overheads incurred by the seminumeric method are 3.16 and 9.40. For half-length randomizers, the best total overheads of the numeric method are $1.56 + 0.16 = 1.72$ and $4.60 + 0.52 = 5.12$ for the two curves. The same overheads for the seminumeric method are slightly better: 1.80 and 5.04. This is the expected pattern as evident from our theoretical estimates. Both the numeric and the seminumeric methods run significantly faster than Montgomery ladders.

Tables 5.11–5.14 list the speedup figures obtained in the case where memory to store precomputed data is not available, that is, individual verification does not use fixed-base double scalar multiplication. Tables 5.15–5.18 deal with the case where individual verification uses fixed-base double scalar multiplication. This case calls for significant storage overhead and so is separately shown. Since the precomputed tables speeds up individual verification noticeably, the maximum speedup obtained by $\mathrm{SP}_{\mathrm{gcd}}$ over individual verification now drops to about 50% for batch size seven.

In ECDSA$^{\#}$, there is a possibility of using multiple scalar multiplication. Table 5.2–5.5 list the times for computing the sum $\xi_1 R_1 + \xi_2 R_2$ using a single double-and-add loop. These times are much less than two separate scalar-multiplication times even by the best windowed method.

5.5 Adaptation of Randomization Methods to Koblitz Curves

Our randomization methods and the batch-verification methods of Chapters 3 and 4 can be ported *mutatis mutandis* to other popular curves.

5.5.1 Ordinary Elliptic Curves Defined over Binary Fields

As an illustrative example, we take the family of Koblitz curves recommended by NIST [48]. These curves are defined over binary fields $\mathbb{F}_{2^d}$ by the equation

$$y^2 + xy = x^3 + ax^2 + 1, \text{ with } a = 0 \text{ or } 1. \tag{5.6}$$

Montgomery-Ladder Formulas

We now represent elliptic-curve points in the standard projective coordinates. Let $P_1 = (h_1, k_1, l_1)$ and $P_2 = (h_2, k_2, l_2)$ be two non-zero multiples of R. Suppose that $P_1 \neq \pm P_2$ and $P_1 - P_2 = (h_4, k_4, l_4)$. We assume that only the x- and z-coordinates of these points are available. We can compute these coordinates of $P_1 + P_2 = (h_3, k_3, l_3)$ using the following formulas:

$$l_3 = (h_1 k_2)^2 + (h_2 k_1)^2, \qquad x_3 = l_3 h_4 + (h_1 k_2)(h_2 k_1).$$

We compute point doubling $2P_1 = (h_5, k_5, l_5)$ as:

$$h_5 = (h_1^2 + l_1^2)^2, \qquad l_5 = h_1^2 l_1^2.$$

We can easily modify Algorithm 5.1 to the case of projective coordinates.

Numeric-Computation Formulas

Now, the relevant problem is the computation of the two values of y from the equation $y^2 + ry + (r^3 + ar^2 + 1) = 0$. We first replace y by ry to convert the equation to the form $y^2 + y + \alpha = 0$, where $\alpha = \frac{r^3 + ar^2 + 1}{r^2}$. The converted equation is solvable if and only if the absolute trace $\mathrm{Tr}(\alpha)$ is zero. In that case, if d is odd, a solution for y is $\alpha^{2^1} + \alpha^{2^3} + \alpha^{2^5} + \cdots + \alpha^{2^{d-2}}$, and the other solution is 1 plus the first solution. These solutions can be efficiently obtained using a half-trace calculation [24].

Seminumeric-Computation Formulas

Let $R = (r, y)$ with y treated as a symbol satisfying the Koblitz-curve equation $y^2 + ry = r^3 + ar^2 + 1$.

Theorem 5.5.1. *Any non-zero multiple of R can be expressed as* $\left(h, k + \left(\frac{y}{r}\right) h\right)$.

Proof. First, notice that P itself can be so expressed with $h = r$ and $k = 0$. Next, suppose that $P_1 = (h_1, k_1 + \left(\frac{y}{r}\right) h_1)$ and $P_2 = (h_2, k_2 + \left(\frac{y}{r}\right) h_2)$ are two distinct non-zero

multiples of R with $P_3 = P_1 + P_2 \neq \mathcal{O}$. The point-addition formula on Koblitz curves implies that $P_3 = (h_3, k_3 + \left(\frac{y}{r}\right) h_3)$, where

$$\begin{aligned} h_3 &= \left(\frac{k_1+k_2}{h_1+h_2}\right)^2 + \left(\frac{k_1+k_2}{h_1+h_2}\right) + h_1 + h_2 + a + \left(\frac{r^3+ar^2+b}{r^2}\right) \\ &= \frac{h_1(h_2^2+k_2)+h_2(h_1^2+k_1)}{h_1^2+h_2^2}, \\ k_3 &= \left(\frac{k_1+k_2}{h_1+h_2}\right)(h_1+h_3)+h_3+k_1. \end{aligned}$$

The double P_4 of P_1, if non-zero, can be expressed as $(h_4, k_4 + \left(\frac{y}{r}\right) h_4)$, where:

$$h_4 = h_1^2 + \frac{b}{h_1^2}, \qquad k_4 = h_1^2 + \left(h_1 + \frac{k_1}{h_1} + 1\right) h_3.$$

The opposite of $(h, k + \left(\frac{y}{r}\right) h)$ is $(h, (k+h) + \left(\frac{y}{r}\right) h)$. □

For Koblitz curves, the τ-NAF point-multiplication algorithm is computationally very efficient. This motivates using the following theorem.

Theorem 5.5.2. *The second-power Frobenius endomorphism on a point of the form* $\left(h, k + \left(\frac{y}{r}\right) h\right)$ *gives a point in the same form.*

Proof. Let $P_1 = (h_1, k_1 + \left(\frac{y}{r}\right) h_1)$. The point $P_5 = (h_1^2, (k_1 + \left(\frac{y}{r}\right) h_1)^2)$ can again be expressed as $(h_5, k_5 + \left(\frac{y}{r}\right) h_5)$, where:

$$h_5 = h_1^2, \qquad k_5 = k_1^2 + \left(\frac{r^3+ar^2+b}{r^2}\right) h_1^2.$$ □

The above two theorems indicate that for all relevant points of the form $(h, k + \left(\frac{y}{r}\right) h)$, it suffices to store the values of h and k alone. The second term $\left(\frac{y}{r}\right) h$ in the y-coordinate carries no extra information, and does not hamper the arithmetic operations on the points. Indeed, the point negation, doubling, and the second addition formulas are now exactly the same as the numeric formulas for Koblitz curves, without any extra operation. If $a + \frac{r^3+ar^2+b}{r^2} = \frac{r^3+b}{r^2}$ is precomputed, the first formula for computing h_3 does not lead to an increased operation count. Application of the second-power Frobenius endomorphism (computation of k_5), however, now involves a multiplication of h_5 with the precomputed field element $\frac{r^3+ar^2+b}{r^2}$, followed by an addition of this product to k_1^2.

After the h and k values of ξR are computed by any addition-chain method, one obtains the point $\xi R = \left(h, k + \left(\frac{h}{r}\right) y\right)$.

5.5.2 Explicit Formulas for Koblitz Curves

Let $P_1 = (h_1, k_1 + \left(\frac{y}{r}\right) h_1, l_1)$ and $P_2 = (h_2, k_2 + \left(\frac{y}{r}\right) h_2, l_2)$ be two non-zero multiplies of $P = (r, y, 1) \in E(\mathbb{F}_{2^d})$ with $P_1 \neq \pm P_2$, and let $P_3 = P_1 + P_2 = (h_3, k_3 + \left(\frac{y}{r}\right) h_3, l_3)$ and $P_4 = 2P_1 = (h_4, k_4 + \left(\frac{y}{r}\right) h_4, l_4)$. The field element $f' = (r^3 + ar^2 + b)/r^2$ is precomputed. López-Dahab coordinates are used. Affine coordinates give us better practical performance. Here, we present the formulas for López-Dahab coordinates for theoretical interests only.

Point Addition

$$z_{12} = l_1^2,\ z_{22} = l_2^2,\ A_1 = h_1 \cdot l_2,\ A_2 = h_2 \cdot l_1,\ c = A_1 + A_2,\ b_1 = A_1^2,\ b_2 = A_2^2,$$
$$D = b_1 + b_2,\ e_{10} = k_1 \cdot z_{22},\ e_{20} = k_2 \cdot z_{12},\ F_0 = e_{10} + e_{20},\ g_0 = c \cdot F_0,\ l_3 = l_1 \cdot l_2 \cdot D,$$
$$h_3 = A_1 \cdot (e_{20} + b_2) + A_2 \cdot (e_{10} + b_1),\ k_3 = (A_1 \cdot g_0 + e_{10} \cdot D) \cdot D + (g_0 + k_3) \cdot h_3.$$

τ Operation

$$h_4 = h_1^2,\ l_4 = l_1^2,\ k_4 = k_1^2 + h_4 \cdot l_4 \cdot f'.$$

5.5.3 Comparison of Montgomery Ladders and Seminumeric Method

For Koblitz curves, we use standard projective coordinates in the Montgomery-ladder method, and affine coordinates in the τ-*NAF* windowed variant of the seminumeric method. These gave us the best respective running times. In fact, affine coordinates outperformed López-Dahab (LD) coordinates [36] in our implementations.

To analyze the Montgomery-ladder implementation, we take $P_1 = (x_1, y_1, z_1)$, $P_2 = (x_2, y_2, z_2)$, and $P_1 - P_2 = (r, r + y, 1) \in E(\mathbb{F}_{2^d})$ in standard projective coordinates. As earlier, we do not use or compute the y-coordinates. We only compute the x- and z-coordinates of $P_1 + P_2$ and $2P_1$ according to the Montgomery-ladder formulas given in [16, 36]:

$$\begin{aligned} z(P_1 + P_2) &= (x_1 z_2)^2 + (x_2 z_1)^2, \\ x(P_1 + P_2) &= z(P_1 + P_2) \times r + x_1 x_2 z_1 z_2, \end{aligned}$$

$$\begin{aligned} x(2P_1) &= x_1^4 + bz_1^4, \text{ and} \\ z(2P_1) &= x_1^2 z_1^2 \end{aligned}$$

Following the implementation of [24], we need $5M + 5S + 2A$ to compute M_{Mont}.

Now, we analyze the seminumeric algorithm in affine coordinates. All non-zero multiples of $(r, y) \in E(\mathbb{F}_{2^d})$ are of the form $(\beta_x, \beta_y + \frac{\beta_x}{r} y)$ with $\beta_x, \beta_y \in \mathbb{F}_{2^d}$. Let $P_1 = (x_1, y_1 + \frac{x_1}{r} y)$ and $P_2 = (x_2, y_2 + \frac{x_2}{r} y)$ be two such multiples with $P_1 \neq \pm P_2$, and y satisfies the equation $y^2 + ry = r^3 + ar^2 + b$ with r known. We treat y as a symbol for the rest of this section. The following formulas are derived assuming that $b = 1$ and that $B = (r^3 + ar^2 + b)/r^2$ is precomputed. The x- and y-coordinates of $P_3 = P_1 + P_2 = (x_3, y_3 + \frac{x_3}{r} y)$ and $P_4 = \tau(P_1) = (x_4, y_4 + \frac{x_4}{r} y)$ are computed as follows:

$$\begin{aligned} \lambda &= \tfrac{y_1 + y_2}{x_1 + x_2}, & x_4 &= x_1^2, \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + B, & y_4 &= y_1^2 + B x_4. \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1. \end{aligned}$$

$A_{Semi} = 2M + 1S + 1I + 6A$ and $\tau_{Semi} = 1M + 2S$ field operations are needed for each point addition and application of τ, respectively (with B precomputed). Here, point addition does not need any extra multiplication in affine or LD coordinates compared to the formulas given in [35], but the application of τ needs one extra multiplication (in LD coordinates, two extra multiplications are needed). If the addition chain for the scalar multiplier is computed by the τ-*NAF* representation with w-bit windows [16, 24, 70], then the density of non-zero digits is on an average $\frac{1}{w+1}$. For each of these non-zero digits, A_{Semi} is required, and τ_{Semi} is required for each non-zero digit in the addition chain. In case of τ-*NAF*, we use special τ-chains in the precomputation stage [24, 70], where 3-, 4- and 5-bit windows need

$$\begin{aligned} \Pi_3 &= 1\tau_{Semi} + 1A_{Semi}, \\ \Pi_4 &= 3\tau_{Semi} + 3A_{Semi}, \text{ and} \\ \Pi_5 &= 6\tau_{Semi} + 7A_{Semi} \text{ curve operations, respectively.} \end{aligned}$$

For Koblitz-curve scalar multiplication, we have to pay a special attention to the length of the addition chains. Let c be the co-factor of the Koblitz curve given by Eqn (5.6). Then, we have $\#E(\mathbb{F}_{2^d}) = cn$. Let $\mu = (-1)^{1-a}$, and $\alpha = (\alpha_1 + \tau\alpha_2) \in \mathbb{Z}[\tau]$. The *norm* of α is given by

$$N(\alpha) = {\alpha_1}^2 + \mu\alpha_1\alpha_2 + 2{\alpha_2}^2. \tag{5.7}$$

The length of the τ-*NAF* representation of α is approximately $\log_2(N(\alpha))$. After the partial modular reduction [70], the length of the addition chain reduces to a maximum of $d+a$. This reduction takes place only if $N(\alpha) \geqslant \frac{4}{7}n$. Therefore, we make our analysis on the basis of whether $N(\alpha) \geqslant \frac{4}{7}n$ or not.

- Case 1: $N(\alpha) < \frac{4}{7}n$

  Let $\alpha = (\alpha_1 + \tau\alpha_2)$ with $\alpha_2 = 0$ be the scalar multiplier, and $l = \log_2 \alpha$ the bit length of the multiplier. The length of the addition chain obtained by the τ-*NAF* representation is approximately $2l$. In contrast, the binary (and NAF) representations produce addition chains of approximate length l. The Montgomery-ladder scalar multiplication needs $C_{Mont} = lM_{Mont} = l(5M+5S)$ field operations (ignoring additions and subtractions). For w-bit windowed τ-*NAF*, the required operation count in the seminumeric method is

$$\begin{aligned} C_{\tau NAF} &= \Pi_w + 2l\left(\tau_{Semi} + \frac{1}{w+1}A_{Semi}\right) \\ &= \Pi_w + 2l\left((1M+2S) + \frac{2M+1S+1I}{w+1}\right). \end{aligned}$$

  Our experimental environment shows the relations between M, S and I as $1S = 0.88M$ and $1I = 4.75M$. Using these relations and putting $w = 5$, we simplify the above operation counts as

$$C_{Mont} = (9.40M)l,\ C_{\tau NAF} = (6.79M)l + (69.97M).$$

  From these expressions, it follows that the seminumeric method is faster than Montgomery ladders for $l \geqslant 30$.

- Case 2: $N(\alpha) \geqslant \frac{4}{7}n$

  In this case, the length of the τ-NAF addition chain remains nearly $d+a$ (irrespective of the length l of the scalar multiplier), whereas the length of the binary addition chain used by Montgomery ladders increases with l. As l increases, the running time of the seminumeric algorithm remains nearly constant, and the running time of the Montgomery ladder increases linearly with l. Similar operation counts as done in Case 1 now shows that the seminumeric algorithm is faster than Montgomery ladders for

$$l \geqslant 0.36d + 7.44. \tag{5.8}$$

On the other hand, the condition $N(\alpha) \geqslant \frac{4}{7}n$ requires $l \geqslant 0.5d$ approximately. Therefore, the inequality $l \geqslant 0.36d + 7.44$ is always satisfied. In fact, we now have

$$\frac{C_{\tau NAF}}{C_{Mont}} \approx 0.36\frac{d}{l}.$$

For $l = d/2$, the seminumeric algorithms takes about 72% of the running time of Montgomery ladders, and for $l = d$, about 36%.

5.5.4 Experimental Comparison

We have used the same experimental setup as described in Section 5.4.4. The average times of randomization achieved by the seminumeric and the Montgomery-ladder algorithms are listed in Tables 5.22 and 5.23 for NIST Koblitz curves. Here, w is the window size, and l is the bit length of the scalar multiplier (randomizer in the batch-verification application). As mentioned in Section 5.1.1, we have chosen l to be 128, $d/2$ and d. The seminumeric algorithm is found to be faster than the Montgomery-ladder algorithm, particularly for large randomizers.

Tables 5.20 and 5.21 list the overheads associated with the numeric randomization method. In order to compare the performances of the numeric method and the seminumeric method, we add the best possible numeric scalar multiplication time to the best possible time of root finding using a half-trace computation. Both the numeric and the seminumeric methods run much faster than Montgomery ladders.

The seminumeric algorithm is found to be faster than the Montgomery-ladder algorithm, particularly for large randomizers. For Koblitz curves, the speedup is about 100% and 20% for 128-bit and half-length randomizers. This pattern is consistent with the theoretical estimates given above.

Tables 5.24 and 5.25 list the speedup figures over individual verification. The scalars are randomly chosen, and individual verification does not use huge precomputation tables needed for fixed-base double scalar multiplication.

We now discuss two variations in our experiments. First, we study randomly generated addition chains instead of computing the addition chains from randomly chosen scalars. The speedup values are listed in Tables 5.26 and 5.27. For Koblitz

curves, we do not experience noticeable increases in speedup values by randomly choosing the addition chains. This is because the times to compute the addition chains from scalars are negligible compared to scalar-multiplication times.

We also study individual verification by the fixed-base double scalar-multiplication method which is applicable if the batch-verification algorithms run on platforms with enough available memory for storing huge precomputed tables. Here, the precomputation times benefit individual verification when the number t of signatures is at least seven. For smaller values of t, speedup factors of about two are registered. In Tables 5.28 and 5.29, the scalars are randomly chosen, whereas in Tables 5.30 and 5.31, the addition chains are randomly chosen.

In ECDSA$^{\#}$, there is a possibility of using multiple scalar multiplication. Tables 5.22 and 5.23 list the times for computing the sum $\xi_1 R_1 + \xi_2 R_2$ using a single double-and-add loop. These times are much larger than two separate scalar-multiplication times by the best windowed τ-NAF method. This is because in the windowed τ-NAF method, times required for scalar multiplication by half-length and full-length scalars are approximately the same, as discussed in Section 5.5.3. Moreover, the τ operation is much more efficient than point doubling. Nevertheless, the randomization overhead being substantial, we do not obtain much speedup from randomized ECDSA batch verification on Koblitz curves.

Table 5.19: Times (in ms) of half-trace root-finding algorithm for NIST Koblitz curves

Curve	Full memory	Half memory	Quarter memory
K-163	2.68	3.16	3.32
K-233	4.80	5.29	5.44
K-283	6.60	7.08	8.31
K-409	13.32	14.16	15.12
K-571	24.31	26.12	30.83

Table 5.20: Times (in ms) of numeric scalar multiplication for NIST Koblitz curves with scalars randomly chosen

↓ Algorithm / Curve →		K-163		K-233		K-283	
		$l = 80$	$l = 163$	$l = 112$	$l = 233$	$l = 128$	$l = 283$
w-τ-NAF-Seminumeric (affine)	$w = 3$	72.52	73.37	149.24	151.76	216.00	237.00
	$w = 4$	**68.61**	**68.84**	137.99	141.30	201.00	220.00
	$w = 5$	69.69	69.60	**136.29**	**139.41**	**196.00**	**216.00**
w-τ-Seminumeric (LD projective)	$w = 3$	137.22	138.67	285.72	289.75	424.00	457.00
	$w = 4$	130.00	129.62	262.38	268.47	381.00	416.00
	$w = 5$	131.59	130.96	259.58	264.63	380.00	420.00
Multiple scalar multiplcation $\xi_1 R_1 + \xi_2 R_2$ (affine)		**124.77**	**248.00**	**280.62**	**579.39**	**417.00**	**908.62**
Multiple scalar multiplcation $\xi_1 R_1 + \xi_2 R_2$ (LD projective)		213.73	429.00	457.75	942.48	677.00	1486.22

↓ Algorithm / Curve →		K-409			K-571		
		$l = 128$	$l = 192$	$l = 408$	$l = 128$	$l = 256$	$l = 571$
w-τ-NAF-Seminumeric (affine)	$w = 3$	368.72	545.05	582.23	573.00	1140.00	1249.00
	$w = 4$	336.85	498.68	528.88	516.00	1033.00	1144.00
	$w = 5$	**328.17**	**477.93**	**506.56**	**517.00**	**968.00**	**1072.00**
w-τ-Seminumeric (LD projective)	$w = 3$	717.47	1055.56	1130.47	1160.00	2305.00	2496.00
	$w = 4$	656.45	971.50	1031.74	1044.00	2092.00	2333.00
	$w = 5$	635.69	925.09	982.23	1036.00	1941.00	2168.00
Multiple scalar multiplcation $\xi_1 R_1 + \xi_2 R_2$ (affine)		**697.65**	**1066.86**	**2266.63**	**1115.32**	**2233.56**	**4968.33**
Multiple scalar multiplcation $\xi_1 R_1 + \xi_2 R_2$ (LD projective)		1138.53	1750.48	3716.65	1882.13	3770.47	8359.44

Table 5.21: Times (in ms) of numeric scalar multiplication for NIST Koblitz curves with addition chains of the scalars randomly chosen

↓ Algorithm / Curve →		K-163		K-233		K-283	
		$l = 80$	$l = 163$	$l = 112$	$l = 233$	$l = 128$	$l = 283$
w-τ-NAF-Seminumeric (affine)	$w = 3$	71.92	72.66	148.66	151.15	216.00	241.00
	$w = 4$	**68.02**	**68.29**	137.37	140.85	201.00	**216.00**
	$w = 5$	69.14	69.05	**135.60**	**138.81**	**196.00**	**216.00**
w-τ-Seminumeric (LD projective)	$w = 3$	136.52	137.89	285.48	289.38	424.00	461.00
	$w = 4$	129.12	128.85	262.43	268.09	381.00	416.00
	$w = 5$	131.10	130.32	259.19	264.42	380.00	416.00

↓ Algorithm / Curve →		K-409			K-571		
		$l = 128$	$l = 192$	$l = 408$	$l = 128$	$l = 256$	$l = 571$
w-τ-NAF-Seminumeric (affine)	$w = 3$	367.70	543.64	580.77	569.00	1136.00	1245.00
	$w = 4$	335.88	496.99	527.06	520.00	1036.00	1144.00
	$w = 5$	**327.05**	**476.25**	**504.73**	**517.00**	**965.00**	**1072.00**
w-τ-Seminumeric (LD projective)	$w = 3$	716.18	1053.79	1128.61	1156.00	2297.00	2488.00
	$w = 4$	655.07	969.86	1029.79	1040.00	2084.00	2325.00
	$w = 5$	634.58	923.24	980.33	1028.00	1937.00	2160.00

Table 5.22: Times (in ms) of Seminumeric and Montgomery-Ladder scalar multiplication for NIST Koblitz curves with scalars randomly chosen

↓ Algorithm / Curve →		K-163		K-233		K-283	
		$l = 80$	$l = 163$	$l = 112$	$l = 233$	$l = 128$	$l = 283$
w-τ-NAF-Seminumeric (affine)	$w = 3$	93.50	94.78	193.93	197.87	285.77	311.29
	$w = 4$	**89.89**	**90.13**	183.06	187.71	267.12	290.37
	$w = 5$	91.17	91.42	**181.76**	**186.33**	**265.04**	**286.64**
w-τ-Seminumeric (LD projective)	$w = 3$	172.75	174.72	362.86	369.35	539.95	588.59
	$w = 4$	166.50	166.38	342.00	350.38	507.40	549.35
	$w = 5$	168.87	168.46	340.09	347.64	502.48	539.98
Montgomery Ladder Method (affine)		**91.18**	**185.86**	206.16	429.87	306.36	681.33
Montgomery Ladder Method (standard projective)		92.42	188.62	**197.62**	**411.97**	**292.70**	**651.62**

↓ Algorithm / Curve →		K-409			K-571		
		$l = 128$	$l = 192$	$l = 408$	$l = 128$	$l = 256$	$l = 571$
w-τ-NAF-Seminumeric (affine)	$w = 3$	488.90	724.33	772.53	773.31	1542.68	1722.48
	$w = 4$	458.08	678.93	720.16	730.27	1429.38	1596.97
	$w = 5$	**450.54**	**659.68**	**699.45**	**718.25**	**1375.88**	**1530.30**
w-τ-Seminumeric (LD projective)	$w = 3$	931.78	1374.97	1470.51	1511.68	3038.72	3388.42
	$w = 4$	876.90	1300.49	1381.52	1438.17	2836.02	3166.75
	$w = 5$	858.46	1256.85	1335.55	1412.19	2711.68	3024.38
Montgomery Ladder Method (affine)		526.35	792.37	1684.39	847.85	1706.69	3819.01
Montgomery Ladder Method (standard projective)		**504.49**	**758.94**	**1614.39**	**824.02**	**1659.40**	**3714.01**

Table 5.23: Times (in ms) of Seminumeric and Montgomery-Ladder scalar multiplication for NIST Koblitz curves with addition chains of the scalars randomly chosen

↓ Algorithm / Curve →		K-163		K-233		K-283	
		$l = 80$	$l = 163$	$l = 112$	$l = 233$	$l = 128$	$l = 283$
w-τ-NAF-Seminumeric (affine)	$w = 3$	92.01	93.30	193.48	197.44	285.36	310.65
	$w = 4$	**88.56**	**88.87**	182.72	187.41	266.64	289.82
	$w = 5$	89.97	90.13	**181.48**	**185.96**	**264.37**	**285.89**
w-τ-Seminumeric (LD projective)	$w = 3$	170.41	172.55	363.67	370.26	540.01	588.83
	$w = 4$	164.78	164.61	342.70	350.95	507.48	549.13
	$w = 5$	166.90	166.61	340.81	348.39	502.25	539.88
Montgomery Ladder Method (affine)		**91.20**	**186.08**	208.10	434.38	306.45	681.76
Montgomery Ladder Method (standard projective)		92.62	189.01	**199.87**	**416.61**	**293.02**	**651.91**

↓ Algorithm / Curve →		K-409			K-571		
		$l = 128$	$l = 192$	$l = 408$	$l = 128$	$l = 256$	$l = 571$
w-τ-NAF-Seminumeric (affine)	$w = 3$	485.81	719.63	767.26	772.74	1542.01	1721.74
	$w = 4$	454.77	674.06	714.74	729.60	1429.03	1596.69
	$w = 5$	**447.21**	**654.72**	**693.92**	**717.44**	**1375.43**	**1529.79**
w-τ-Seminumeric (LD projective)	$w = 3$	926.63	1367.25	1462.38	1508.38	3033.44	3383.73
	$w = 4$	871.61	1292.42	1372.81	1434.80	2831.61	3161.99
	$w = 5$	854.07	1249.99	1327.72	1408.87	2707.75	3020.82
Montgomery Ladder Method (affine)		527.30	793.66	1686.82	848.50	1708.12	3823.07
Montgomery Ladder Method (standard projective)		**504.49**	**759.17**	**1614.30**	**824.02**	**1660.08**	**3715.66**

Table 5.24: Speedups obtained by different randomized batch-verification algorithms with scalars randomly chosen

(Fixed based double scalar multiplication is *not* used)

Batch Verification Algorithm	Randomization Algorithm	t	K-163		K-233		K-283	
			None*	$l = 80$	None*	$l = 112$	None*	$l = 128$
Algorithm N	Numeric	2	1.79	0.95	1.83	0.97	1.85	1.00
		3	2.23	1.06	2.37	1.10	2.44	1.16
		4	2.13	1.03	2.40	1.10	2.55	1.18
		5	1.58	0.89	1.89	0.98	2.09	1.07
		6	0.98	0.66	1.21	0.76	1.39	0.85
		7	0.53	0.42	0.68	0.51	0.79	0.58
Algorithm N′	Numeric	2	1.88	0.97	1.90	0.98	1.90	1.02
		3	2.71	1.15	2.76	1.18	2.78	1.23
		4	3.49	1.27	3.58	1.30	3.61	1.37
		5	4.22	1.36	4.36	1.39	4.39	1.47
		6	4.89	1.42	5.09	1.46	5.14	1.54
		7	5.53	1.47	5.78	1.51	5.85	1.60
Algorithm S2′	Seminumeric	2	1.95	0.86	1.96	0.86	1.97	0.89
		3	2.82	0.99	2.87	1.00	2.89	1.04
		4	3.48	1.06	3.59	1.08	3.66	1.13
		5	3.31	1.05	3.61	1.08	3.78	1.14
		6	2.81	0.99	3.22	1.04	3.48	1.11
		7	1.50	0.76	1.84	0.84	2.09	0.92
Algorithm S3	Seminumeric	2	1.91	0.85	1.93	0.85	1.95	0.89
		3	2.70	0.98	2.74	0.98	2.77	1.03
		4	3.23	1.04	3.32	1.05	3.39	1.10
		5	3.15	1.03	3.35	1.05	3.50	1.11
		6	2.89	1.00	3.17	1.03	3.42	1.10
		7	2.12	0.89	2.43	0.94	2.69	1.01
Algorithm $S3_{gcd}$	Seminumeric	2	1.91	0.85	1.93	0.85	1.93	0.88
		3	2.57	0.96	2.61	0.97	2.63	1.01
		4	3.18	1.03	3.27	1.04	3.35	1.10
		5	3.11	1.03	3.30	1.05	3.45	1.11
		6	2.88	1.00	3.07	1.02	3.38	1.10
		7	2.12	0.89	2.40	0.94	2.65	1.01
Algorithm SP	Seminumeric	2	1.97	0.86	1.97	0.86	1.98	0.89
		3	2.91	1.00	2.93	1.01	2.94	1.05
		4	3.55	1.07	3.63	1.08	3.68	1.13
		5	4.13	1.12	4.22	1.13	4.30	1.18
		6	4.14	1.12	4.38	1.14	4.54	1.20
		7	2.37	0.93	2.69	0.98	2.94	1.05
		8	2.34	0.93	2.67	0.97	2.92	1.05
		9	1.61	0.79	1.87	0.84	2.09	0.92
		10	0.63	0.45	1.01	0.61	1.16	0.68
Algorithm SP_{gcd}	Seminumeric	2	1.97	0.86	1.97	0.86	1.98	0.89
		3	2.91	1.00	2.93	1.01	2.94	1.05
		4	3.75	1.09	3.81	1.09	3.84	1.14
		5	4.55	1.15	4.64	1.15	4.70	1.21
		6	4.61	1.15	4.85	1.17	5.00	1.23
		7	2.60	0.96	2.95	1.01	3.21	1.08
		8	2.69	0.98	3.06	1.02	3.36	1.10
		9	2.01	0.87	2.37	0.93	2.64	1.01
		10	0.71	0.49	1.02	0.61	1.17	0.68

* without randomization

Table 5.25: Speedups obtained by different randomized batch-verification algorithms with scalars randomly chosen

(Fixed based double scalar multiplication is *not* used)

Batch Verification Algorithm	Randomization Algorithm	t	K-409			K-571		
			None*	$l = 128$	$l = 192$	None*	$l = 128$	$l = 256$
Algorithm N′	Numeric	2	1.88	1.17	1.00	1.90	1.30	1.02
		3	2.57	1.40	1.16	2.64	1.61	1.20
		4	2.84	1.48	1.21	3.01	1.74	1.28
		5	2.50	1.38	1.15	2.80	1.67	1.24
		6	1.78	1.13	0.97	2.10	1.40	1.08
		7	1.05	0.79	0.70	1.31	0.99	0.82
Algorithm N′	Numeric	2	1.93	1.19	1.01	1.93	1.32	1.03
		3	2.83	1.48	1.21	2.84	1.69	1.24
		4	3.70	1.68	1.35	3.72	1.96	1.39
		5	4.53	1.84	1.44	4.57	2.17	1.49
		6	5.33	1.95	1.52	5.38	2.34	1.57
		7	6.09	2.05	1.57	6.17	2.48	1.63
Algorithm S2′	Seminumeric	2	1.98	1.05	0.86	1.98	1.19	0.87
		3	2.92	1.27	1.01	2.94	1.48	1.02
		4	3.75	1.41	1.09	3.80	1.67	1.11
		5	4.07	1.45	1.11	4.25	1.75	1.14
		6	3.96	1.43	1.11	4.28	1.76	1.14
		7	2.61	1.21	0.97	3.05	1.51	1.03
Algorithm S2	Seminumeric	2	1.95	1.04	0.86	1.96	1.18	0.87
		3	2.81	1.25	0.99	2.84	1.46	1.01
		4	3.50	1.37	1.07	3.57	1.63	1.09
		5	3.75	1.41	1.09	3.91	1.69	1.11
		6	3.72	1.40	1.09	4.00	1.71	1.12
		7	3.10	1.30	1.03	3.44	1.60	1.07
Algorithm $S3_{gcd}$	Seminumeric	2	1.95	1.04	0.86	1.96	1.18	0.87
		3	2.69	1.22	0.98	2.72	1.42	0.99
		4	3.46	1.36	1.06	3.53	1.62	1.08
		5	3.69	1.40	1.08	3.85	1.68	1.11
		6	3.70	1.40	1.09	3.96	1.70	1.12
		7	3.05	1.29	1.02	3.37	1.58	1.07
Algorithm SP	Seminumeric	2	1.98	1.05	0.87	1.99	1.19	0.87
		3	2.95	1.28	1.01	2.96	1.49	1.02
		4	3.75	1.41	1.09	3.79	1.67	1.10
		5	4.41	1.49	1.14	4.49	1.79	1.16
		6	4.81	1.53	1.16	4.98	1.87	1.19
		7	3.43	1.36	1.06	3.79	1.67	1.10
		8	3.45	1.36	1.06	3.84	1.68	1.11
		9	2.56	1.20	0.96	2.96	1.49	1.02
		10	1.51	0.90	0.76	1.82	1.13	0.84
Algorithm SP_{gcd}	Seminumeric	2	1.98	1.05	0.87	1.99	1.19	0.87
		3	2.95	1.28	1.01	2.96	1.49	1.02
		4	3.88	1.42	1.10	3.91	1.69	1.11
		5	4.77	1.53	1.16	4.82	1.84	1.18
		6	5.24	1.57	1.19	5.39	1.92	1.21
		7	3.71	1.40	1.09	4.06	1.72	1.13
		8	3.92	1.43	1.10	4.35	1.77	1.15
		9	3.20	1.32	1.04	3.65	1.64	1.09
		10	1.51	0.90	0.76	1.82	1.13	0.84

* without randomization

Table 5.26: Speedups obtained by different randomized batch-verification algorithms with addition chains of the scalars randomly chosen

(Fixed based double scalar multiplication is *not* used)

Batch Verification Algorithm	Randomization Algorithm	t	K-163		K-233		K-283	
			None*	$l = 80$	None*	$l = 112$	None*	$l = 128$
Algorithm N	Numeric	2	1.79	0.95	1.83	0.97	1.85	1.00
		3	2.23	1.06	2.37	1.10	2.44	1.16
		4	2.13	1.04	2.40	1.11	2.55	1.18
		5	1.58	0.89	1.89	0.98	2.09	1.07
		6	0.98	0.66	1.21	0.76	1.39	0.85
		7	0.53	0.42	0.68	0.51	0.79	0.58
Algorithm N′	Numeric	2	1.88	0.97	1.90	0.99	1.90	1.02
		3	2.71	1.16	2.76	1.18	2.78	1.23
		4	3.49	1.28	3.58	1.31	3.61	1.37
		5	4.22	1.37	4.36	1.40	4.39	1.47
		6	4.89	1.43	5.09	1.46	5.14	1.54
		7	5.53	1.48	5.78	1.52	5.85	1.60
Algorithm S3,	Seminumeric	2	1.95	0.86	1.96	0.86	1.97	0.89
		3	2.82	1.00	2.87	1.00	2.89	1.04
		4	3.48	1.07	3.59	1.08	3.66	1.13
		5	3.31	1.06	3.61	1.08	3.78	1.14
		6	2.81	1.00	3.22	1.04	3.48	1.11
		7	1.50	0.76	1.84	0.84	2.09	0.92
Algorithm S3	Seminumeric	2	1.91	0.86	1.93	0.85	1.95	0.89
		3	2.70	0.99	2.74	0.98	2.77	1.03
		4	3.23	1.05	3.32	1.05	3.39	1.10
		5	3.15	1.04	3.35	1.05	3.50	1.11
		6	2.89	1.01	3.17	1.04	3.42	1.11
		7	2.12	0.90	2.43	0.94	2.69	1.02
Algorithm $S3_{gcd}$	Seminumeric	2	1.91	0.86	1.93	0.86	1.93	0.88
		3	2.57	0.97	2.61	0.97	2.63	1.01
		4	3.18	1.04	3.27	1.05	3.35	1.10
		5	3.11	1.04	3.30	1.05	3.45	1.11
		6	2.88	1.01	3.07	1.02	3.38	1.10
		7	2.12	0.90	2.40	0.94	2.65	1.01
Algorithm SP	Seminumeric	2	1.97	0.87	1.97	0.86	1.98	0.89
		3	2.91	1.01	2.93	1.01	2.94	1.05
		4	3.55	1.08	3.63	1.08	3.68	1.13
		5	4.13	1.13	4.22	1.13	4.30	1.18
		6	4.14	1.13	4.38	1.14	4.54	1.20
		7	2.37	0.94	2.69	0.98	2.94	1.05
		8	2.34	0.93	2.67	0.97	2.92	1.05
		9	1.61	0.79	1.87	0.84	2.09	0.92
		10	0.63	0.45	1.01	0.61	1.16	0.68
Algorithm SP_{gcd}	Seminumeric	2	1.97	0.87	1.97	0.86	1.98	0.89
		3	2.91	1.01	2.93	1.01	2.94	1.05
		4	3.75	1.10	3.81	1.09	3.84	1.15
		5	4.55	1.16	4.64	1.15	4.70	1.21
		6	4.61	1.16	4.85	1.17	5.00	1.23
		7	2.60	0.97	2.95	1.01	3.21	1.08
		8	2.69	0.98	3.06	1.02	3.36	1.10
		9	2.01	0.88	2.37	0.93	2.64	1.01
		10	0.71	0.49	1.02	0.61	1.17	0.68

* without randomization

Table 5.27: Speedups obtained by different randomized batch-verification algorithms with addition chains of the scalars randomly chosen

(Fixed based double scalar multiplication is *not* used)

Batch Verification Algorithm	Randomization Algorithm	t	K-409			K-571		
			None*	$l = 128$	$l = 192$	None*	$l = 128$	$l = 256$
Algorithm N	Numeric	2	1.88	1.17	1.00	1.90	1.30	1.02
		3	2.57	1.40	1.16	2.64	1.61	1.21
		4	2.84	1.48	1.22	3.01	1.74	1.28
		5	2.50	1.38	1.15	2.80	1.67	1.24
		6	1.78	1.13	0.97	2.10	1.40	1.08
		7	1.05	0.79	0.70	1.31	0.99	0.82
Algorithm N′	Numeric	2	1.93	1.19	1.01	1.93	1.32	1.03
		3	2.83	1.48	1.21	2.84	1.69	1.25
		4	3.70	1.69	1.35	3.72	1.96	1.39
		5	4.53	1.84	1.45	4.57	2.17	1.49
		6	5.33	1.96	1.52	5.38	2.34	1.57
		7	6.09	2.05	1.58	6.17	2.48	1.63
Algorithm S2′	Seminumeric	2	1.98	1.06	0.87	1.98	1.19	0.87
		3	2.92	1.28	1.01	2.94	1.48	1.02
		4	3.75	1.41	1.10	3.80	1.67	1.11
		5	4.07	1.46	1.12	4.25	1.76	1.14
		6	3.96	1.44	1.11	4.28	1.76	1.14
		7	2.61	1.21	0.97	3.05	1.51	1.03
Algorithm S3	Seminumeric	2	1.95	1.05	0.86	1.96	1.18	0.87
		3	2.81	1.26	1.00	2.84	1.46	1.01
		4	3.50	1.38	1.07	3.57	1.63	1.09
		5	3.75	1.41	1.10	3.91	1.69	1.11
		6	3.72	1.41	1.09	4.00	1.71	1.12
		7	3.10	1.31	1.03	3.44	1.60	1.07
Algorithm $S3_{gcd}$	Seminumeric	2	1.95	1.05	0.86	1.96	1.18	0.87
		3	2.69	1.23	0.98	2.72	1.42	0.99
		4	3.46	1.37	1.07	3.53	1.62	1.08
		5	3.69	1.40	1.09	3.85	1.68	1.11
		6	3.70	1.41	1.09	3.96	1.70	1.12
		7	3.05	1.30	1.03	3.37	1.58	1.07
Algorithm SP	Seminumeric	2	1.98	1.06	0.87	1.99	1.19	0.87
		3	2.95	1.28	1.02	2.96	1.49	1.02
		4	3.75	1.41	1.10	3.79	1.67	1.10
		5	4.41	1.50	1.15	4.49	1.79	1.16
		6	4.81	1.54	1.17	4.98	1.87	1.19
		7	3.43	1.36	1.07	3.79	1.67	1.10
		8	3.45	1.37	1.07	3.84	1.68	1.11
		9	2.56	1.20	0.96	2.96	1.49	1.02
		10	1.51	0.91	0.76	1.82	1.13	0.84
Algorithm SP_{gcd}	Seminumeric	2	1.98	1.06	0.87	1.99	1.19	0.87
		3	2.95	1.28	1.02	2.96	1.49	1.02
		4	3.88	1.43	1.11	3.91	1.69	1.11
		5	4.77	1.54	1.17	4.82	1.85	1.18
		6	5.24	1.58	1.19	5.39	1.92	1.21
		7	3.71	1.41	1.09	4.06	1.72	1.13
		8	3.92	1.44	1.11	4.35	1.77	1.15
		9	3.20	1.33	1.04	3.65	1.64	1.09
		10	1.51	0.91	0.77	1.82	1.13	0.84

* without randomization

Table 5.28: Speedups obtained by different randomized batch-verification algorithms with scalars randomly chosen

(Fixed based double scalar multiplication is *used*)

Batch Verification Algorithm	Randomization Algorithm	t	K-163		K-233		K-283	
			None*	$l = 80$	None*	$l = 112$	None*	$l = 128$
Algorith N	Numeric	2	1.79	0.95	1.83	0.97	1.85	1.00
		3	2.23	1.06	2.37	1.10	2.44	1.16
		4	2.13	1.03	2.40	1.10	2.55	1.18
		5	1.58	0.89	1.89	0.98	2.09	1.07
		6	0.98	0.66	1.21	0.76	1.39	0.85
		7	0.53	0.42	0.68	0.51	0.79	0.58
Algorith N′	Numeric	2	4.07	2.10	4.57	2.37	4.60	2.47
		3	4.81	2.05	5.43	2.31	5.47	2.42
		4	5.51	2.01	6.25	2.27	6.29	2.39
		5	6.16	1.99	7.02	2.24	7.07	2.36
		6	6.77	1.97	7.75	2.22	7.82	2.35
		7	7.34	1.95	8.44	2.21	8.52	2.33
Algorith S2′	Seminumeric	2	4.22	1.86	4.72	2.07	4.75	2.15
		3	5.01	1.76	5.63	1.96	5.68	2.05
		4	5.50	1.68	6.27	1.88	6.38	1.97
		5	4.84	1.53	5.82	1.74	6.09	1.83
		6	3.89	1.37	4.91	1.58	5.29	1.69
		7	1.99	1.01	2.69	1.22	3.05	1.34
Algorith S3	Seminumeric	2	4.15	1.85	4.64	2.06	4.70	2.14
		3	4.81	1.74	5.39	1.93	5.45	2.02
		4	5.10	1.64	5.80	1.83	5.92	1.92
		5	4.61	1.51	5.39	1.70	5.64	1.79
		6	4.00	1.39	4.84	1.58	5.21	1.68
		7	2.81	1.18	3.54	1.37	3.92	1.48
Algorith $S3_{gcd}$	Seminumeric	2	4.16	1.85	4.64	2.06	4.65	2.13
		3	4.57	1.71	5.12	1.90	5.18	1.98
		4	5.03	1.63	5.71	1.82	5.84	1.91
		5	4.55	1.50	5.33	1.69	5.55	1.78
		6	3.98	1.38	4.68	1.56	5.15	1.67
		7	2.81	1.18	3.50	1.37	3.86	1.47
Algorith SP	Seminumeric	2	4.27	1.87	4.75	2.08	4.77	2.16
		3	5.16	1.78	5.75	1.98	5.78	2.06
		4	5.62	1.69	6.34	1.88	6.41	1.97
		5	6.04	1.63	6.80	1.81	6.92	1.90
		6	5.74	1.55	6.67	1.73	6.90	1.82
		7	3.15	1.24	3.94	1.43	4.29	1.53
		8	3.01	1.19	3.77	1.38	4.12	1.48
		9	2.02	0.98	2.58	1.16	2.87	1.26
		10	0.78	0.55	1.37	0.82	1.56	0.91
Algorith SP_{gcd}	Seminumeric	2	4.27	1.87	4.75	2.08	4.77	2.16
		3	5.16	1.78	5.75	1.98	5.78	2.06
		4	5.93	1.72	6.64	1.91	6.69	2.00
		5	6.66	1.68	7.48	1.86	7.58	1.95
		6	6.38	1.59	7.40	1.78	7.61	1.87
		7	3.45	1.28	4.30	1.47	4.67	1.58
		8	3.45	1.25	4.33	1.44	4.74	1.55
		9	2.52	1.09	3.26	1.28	3.62	1.38
		10	0.88	0.60	1.37	0.82	1.57	0.92

* without randomization

Table 5.29: Speedups obtained by different randomized batch-verification algorithms with scalars randomly chosen

(Fixed based double scalar multiplication is *used*)

Batch Verification Algorithm	Randomization Algorithm	t	K-409			K-571		
			None*	$l = 128$	$l = 192$	None*	$l = 128$	$l = 256$
Algorithm N′	Numeric	2	4.66	2.89	2.47	4.85	3.33	2.61
		3	5.17	2.82	2.34	5.48	3.35	2.50
		4	5.05	2.63	2.16	5.52	3.20	2.34
		5	4.11	2.27	1.88	4.74	2.83	2.09
		6	2.75	1.75	1.50	3.36	2.23	1.72
		7	1.56	1.17	1.04	2.00	1.52	1.25
Algorithm N′	Numeric	2	4.76	2.93	2.50	4.94	3.37	2.64
		3	5.69	2.97	2.44	5.90	3.50	2.58
		4	6.58	2.99	2.40	6.83	3.60	2.55
		5	7.43	3.01	2.37	7.72	3.67	2.52
		6	8.26	3.03	2.35	8.59	3.74	2.50
		7	9.04	3.04	2.33	9.41	3.78	2.49
Algorithm S2′	Seminumeric	2	4.89	2.60	2.14	5.06	3.04	2.23
		3	5.87	2.55	2.02	6.09	3.07	2.11
		4	6.67	2.50	1.94	6.98	3.07	2.03
		5	6.68	2.38	1.83	7.19	2.97	1.93
		6	6.14	2.22	1.72	6.83	2.80	1.82
		7	3.87	1.79	1.44	4.66	2.30	1.57
Algorithm S3	Seminumeric	2	4.81	2.58	2.12	5.00	3.02	2.22
		3	5.66	2.51	2.00	5.90	3.02	2.09
		4	6.24	2.44	1.90	6.56	2.98	1.99
		5	6.15	2.31	1.79	6.61	2.86	1.88
		6	5.77	2.17	1.68	6.38	2.73	1.79
		7	4.60	1.93	1.52	5.24	2.44	1.64
Algorithm $S3_{gcd}$	Seminumeric	2	4.82	2.58	2.12	5.00	3.02	2.22
		3	5.41	2.46	1.97	5.65	2.96	2.06
		4	6.16	2.43	1.89	6.48	2.97	1.98
		5	6.06	2.30	1.78	6.50	2.84	1.88
		6	5.74	2.17	1.68	6.31	2.71	1.78
		7	4.53	1.92	1.52	5.14	2.42	1.63
Algorithm SP	Seminumeric	2	4.90	2.61	2.14	5.08	3.05	2.23
		3	5.94	2.57	2.03	6.15	3.09	2.12
		4	6.68	2.50	1.94	6.96	3.07	2.03
		5	7.25	2.45	1.87	7.59	3.03	1.96
		6	7.46	2.38	1.80	7.95	2.98	1.89
		7	5.09	2.02	1.57	5.78	2.55	1.68
		8	4.95	1.95	1.52	5.67	2.48	1.63
		9	3.57	1.67	1.34	4.25	2.13	1.47
		10	2.06	1.23	1.04	2.55	1.58	1.18
Algorithm SP_{gcd}	Seminumeric	2	4.90	2.61	2.14	5.08	3.05	2.23
		3	5.94	2.57	2.03	6.15	3.09	2.12
		4	6.91	2.53	1.96	7.17	3.10	2.04
		5	7.84	2.51	1.91	8.15	3.12	1.99
		6	8.12	2.44	1.84	8.60	3.06	1.93
		7	5.51	2.08	1.61	6.20	2.63	1.72
		8	5.63	2.05	1.58	6.41	2.61	1.69
		9	4.47	1.84	1.45	5.24	2.36	1.57
		10	2.07	1.24	1.04	2.55	1.58	1.18

* without randomization

Table 5.30: Speedups obtained by different randomized batch-verification algorithms with addition chains of the scalars randomly chosen

(Fixed based double scalar multiplication is *used*)

Batch Verification Algorithm	Randomization Algorithm	t	K-163		K-233		K-283	
			None*	$l = 80$	None*	$l = 112$	None*	$l = 128$
Algorithm N	Numeric	2	3.88	2.06	4.40	2.33	4.45	2.42
		3	3.97	1.89	4.66	2.16	4.80	2.28
		4	3.36	1.64	4.18	1.93	4.45	2.06
		5	2.32	1.30	3.05	1.59	3.37	1.73
		6	1.36	0.91	1.84	1.16	2.12	1.30
		7	0.70	0.56	0.99	0.74	1.15	0.85
Algorithm N′	Numeric	2	4.07	2.11	4.57	2.38	4.60	2.47
		3	4.81	2.06	5.43	2.32	5.47	2.42
		4	5.51	2.02	6.25	2.28	6.29	2.39
		5	6.16	2.00	7.02	2.25	7.07	2.36
		6	6.77	1.98	7.75	2.23	7.82	2.35
		7	7.34	1.97	8.44	2.21	8.52	2.33
Algorithm S2′	Seminumeric	2	4.22	1.88	4.72	2.07	4.75	2.15
		3	5.01	1.78	5.63	1.97	5.68	2.05
		4	5.50	1.70	6.27	1.88	6.38	1.97
		5	4.84	1.55	5.82	1.74	6.09	1.84
		6	3.89	1.39	4.91	1.58	5.29	1.69
		7	1.99	1.01	2.69	1.22	3.05	1.34
Algorithm S3	Seminumeric	2	4.15	1.86	4.64	2.06	4.70	2.14
		3	4.81	1.75	5.39	1.94	5.45	2.02
		4	5.10	1.66	5.80	1.83	5.92	1.92
		5	4.61	1.52	5.39	1.70	5.64	1.79
		6	4.00	1.40	4.84	1.58	5.21	1.68
		7	2.81	1.19	3.54	1.37	3.92	1.48
Algorithm $S3_{gcd}$	Seminumeric	2	4.16	1.86	4.64	2.06	4.65	2.13
		3	4.57	1.72	5.12	1.90	5.18	1.98
		4	5.03	1.65	5.71	1.82	5.84	1.92
		5	4.55	1.52	5.33	1.69	5.55	1.79
		6	3.98	1.40	4.68	1.56	5.15	1.68
		7	2.81	1.19	3.50	1.37	3.86	1.47
Algorithm SP	Seminumeric	2	4.27	1.88	4.75	2.08	4.77	2.16
		3	5.16	1.80	5.75	1.98	5.78	2.07
		4	5.62	1.71	6.34	1.88	6.41	1.97
		5	6.04	1.65	6.80	1.82	6.92	1.91
		6	5.74	1.56	6.67	1.73	6.90	1.83
		7	3.15	1.25	3.94	1.43	4.29	1.53
		8	3.01	1.20	3.77	1.38	4.12	1.48
		9	2.02	0.99	2.58	1.16	2.87	1.26
		10	0.78	0.55	1.37	0.82	1.56	0.91
Algorithm SP_{gcd}	Seminumeric	2	4.27	1.88	4.75	2.08	4.77	2.16
		3	5.16	1.80	5.75	1.98	5.78	2.07
		4	5.93	1.74	6.64	1.91	6.69	2.00
		5	6.66	1.69	7.48	1.86	7.58	1.95
		6	6.38	1.61	7.40	1.78	7.61	1.87
		7	3.45	1.29	4.30	1.48	4.67	1.58
		8	3.45	1.27	4.33	1.45	4.74	1.55
		9	2.52	1.10	3.26	1.28	3.62	1.38
		10	0.88	0.60	1.37	0.82	1.57	0.92

* without randomization

Table 5.31: Speedups obtained by different randomized batch-verification algorithms with addition chains of the scalars randomly chosen

(Fixed based double scalar multiplication is *used*)

Batch Verification Algorithm	Randomization Algorithm	t	K-409			K-571		
			None*	$l = 128$	$l = 192$	None*	$l = 128$	$l = 256$
Algorithm N	Numeric	2	4.66	2.90	2.47	4.85	3.33	2.62
		3	5.17	2.83	2.34	5.48	3.35	2.50
		4	5.05	2.64	2.16	5.52	3.20	2.35
		5	4.11	2.27	1.89	4.74	2.83	2.10
		6	2.75	1.75	1.50	3.36	2.23	1.72
		7	1.56	1.17	1.05	2.00	1.52	1.26
Algorithm N′	Numeric	2	4.76	2.94	2.50	4.94	3.37	2.64
		3	5.69	2.97	2.44	5.90	3.50	2.59
		4	6.58	3.00	2.40	6.83	3.60	2.55
		5	7.43	3.02	2.38	7.72	3.67	2.53
		6	8.26	3.04	2.36	8.59	3.74	2.51
		7	9.04	3.05	2.34	9.41	3.78	2.49
Algorithm S3′	Seminumeric	2	4.89	2.61	2.15	5.06	3.05	2.23
		3	5.87	2.57	2.03	6.09	3.07	2.11
		4	6.67	2.51	1.95	6.98	3.07	2.03
		5	6.68	2.39	1.84	7.19	2.97	1.93
		6	6.14	2.23	1.72	6.83	2.81	1.82
		7	3.87	1.80	1.44	4.66	2.30	1.57
Algorithm S3	Seminumeric	2	4.81	2.59	2.13	5.00	3.02	2.22
		3	5.66	2.52	2.01	5.90	3.02	2.09
		4	6.24	2.45	1.91	6.56	2.99	1.99
		5	6.15	2.32	1.80	6.61	2.86	1.88
		6	5.77	2.18	1.69	6.38	2.73	1.79
		7	4.60	1.94	1.53	5.24	2.44	1.64
Algorithm $S3_{gcd}$	Seminumeric	2	4.82	2.59	2.13	5.00	3.02	2.22
		3	5.41	2.47	1.98	5.65	2.96	2.06
		4	6.16	2.44	1.90	6.48	2.97	1.98
		5	6.06	2.31	1.79	6.50	2.84	1.88
		6	5.74	2.18	1.69	6.31	2.72	1.78
		7	4.53	1.93	1.52	5.14	2.42	1.63
Algorithm SP	Seminumeric	2	4.90	2.62	2.15	5.08	3.05	2.23
		3	5.94	2.58	2.04	6.15	3.09	2.12
		4	6.68	2.51	1.95	6.96	3.07	2.03
		5	7.25	2.46	1.88	7.59	3.03	1.96
		6	7.46	2.39	1.81	7.95	2.98	1.89
		7	5.09	2.03	1.58	5.78	2.55	1.69
		8	4.95	1.96	1.53	5.67	2.48	1.64
		9	3.57	1.68	1.35	4.25	2.14	1.47
		10	2.06	1.24	1.04	2.55	1.59	1.18
Algorithm SP_{gcd}	Seminumeric	2	4.90	2.62	2.15	5.08	3.05	2.23
		3	5.94	2.58	2.04	6.15	3.09	2.12
		4	6.91	2.55	1.97	7.17	3.11	2.04
		5	7.84	2.52	1.92	8.15	3.12	1.99
		6	8.12	2.45	1.85	8.60	3.07	1.93
		7	5.51	2.09	1.62	6.20	2.63	1.72
		8	5.63	2.06	1.59	6.41	2.61	1.69
		9	4.47	1.85	1.46	5.24	2.36	1.57
		10	2.07	1.24	1.04	2.55	1.58	1.18

* without randomization

5.6 Chapter Summary

In this chapter, a numeric alternative and a seminumeric alternative to Montgomery ladders are proposed for randomizing the batch verification of ECDSA and ECDSA$^{\#}$ signatures. We theoretically and experimentally establish the superiority of the numeric and seminumeric methods over Montgomery ladders. This study is particularly relevant in the context of several attacks that can be mounted on unrandomized batch-verification schemes.

Chapter 6

Batch Verification of EdDSA Signatures

In Chapters 3 and 4, several algorithms are proposed for the batch verification of ECDSA signatures. In this chapter, we make a comparative study of these methods for the Edwards curve digital signature algorithm (EdDSA). We describe the adaptation of Algorithms N, N′, S2′, S3 and SP for EdDSA signatures. The randomization methods are also explained in detail. More precisely, we study seminumeric scalar multiplication and Montgomery ladders during randomization of EdDSA signatures. Each EdDSA signature verification involves a square-root computation. One may instead use an ECDSA-like verification procedure which avoids the expensive square-root computation. We study both these variants of EdDSA verification. Experimental results show that for small batch sizes the Algorithms S2′ and SP yield speedup comparable to what is achieved by Algorithm N′ which is originally proposed as the default EdDSA batch-verification algorithm.

6.1 Edwards Curve Digital Signature Algorithm (EdDSA)

Bernstein et al. in [8] propose the Edwards Curve Digital Signature Algorithm (EdDSA). This signature scheme is based on the group structure of the twisted

Edwards curve over a prime field $\mathbb{F}_p$ defined as

$$E : -x^2 + y^2 = 1 + dx^2y^2, \tag{6.1}$$

where d is not a square element in $\mathbb{F}_p$ and $d \notin \{0, -1\}$.

To set up EdDSA signatures, one fixes the following domain parameters:

$$\begin{aligned} b &= \text{an integer} \geqslant 10, \\ H &= \text{a cryptographic hash function whose output is } 2b \text{ bits long}, \\ p &= \text{a prime congruent to 1 modulo 4}, \\ d &= \text{a non-square element in } \mathbb{F}_p,\ d \neq 0, -1, \\ \lambda &= \text{a prime in the range } \left[2^{b-4}, 2^{b-3}\right], \\ B &= \text{a point of the curve that acts as the base point, } B \neq (0,1). \end{aligned}$$

These domain parameters are the same for all the entities participating in a network. The Edwards-curve group is an additive group, where the sum of two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on the curve is the point $P_3 = P_1 + P_2 = (x_3, y_3)$ that can be computed using the twisted Edwards-curve addition formula as given in [6]:

$$(x_3, y_3) = (x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 + x_1x_2}{1 - dx_1x_2y_1y_2} \right) \tag{6.2}$$

Now, we describe the three parts of the EdDSA signature scheme. The signer creates his/her key pair using Algorithm 6.1. Let M be a message. Algorithm 6.2 generates the EdDSA signature (R', S) on the message M. The validity of the signature is checked by Algorithm 6.3.

Algorithm 6.1 EdDSA Key Generation

INPUT: Domain Parameters.

OUTPUT: Public key A, private key k.

- Choose a random b-bit string as k.
- Compute $H(k) = (h_0, h_1, \ldots, h_{2b-1})$.
- Compute $a = 2^{b-2} + \sum_{3 \leqslant i \leqslant b-3} 2^i h_i \in \{2^{b-2}, 2^{b-2} + 8, \ldots, 2^{b-1} - 8\}$.
- Compute $A = aB$.

Algorithm 6.2 EdDSA Signature Generation

INPUT: Domain Parameters, message M, private key k, and $H(k) = (h_0, h_1, \ldots, h_{2b-1})$.

OUTPUT: The EdDSA signature (R', S) on M.

- Compute $r = H(h_b, \ldots, h_{2b-1}, M) \in \{0, 1, \ldots, 2^{2b} - 1\}$.
- Compute $R = rB \in E$.
- R' = (a bit identifying the x-coordinate of R) $||$ (the y-coordinate of R).
- Compute $S = r + H(R', A, M)a \pmod{\lambda}$.

Algorithm 6.3 EdDSA Signature Verification

INPUT: Domain Parameters, message M, public key A, and signature (R', S).

OUTPUT: Accept or reject.

- Compute $H(R', A, M)$.
- Compute R from R' (using a square-root computation as described in the text).
- Accept the signature if and only if the equation $SB = R + H(R', A, M)A$ holds.

Algorithm 6.4 Alternative EdDSA Signature Verification

INPUT: Domain Parameters, message M, public key A, and signature (R', S).

OUTPUT: Accept or reject.

- Compute $H(R', A, M)$.
- Extract the y-coordinate R_y of R from R'.
- Accept the signature if and only if the equation $R_y = y\,(SB - H(R', A, M)A)$ holds.

In the verification Algorithm 6.3, we have to compute R from R' which contains the y-coordinate of the point R and an identifier bit for the x-coordinate of R. From the known y-coordinate, we first compute two x-coordinates by $x \equiv \pm\sqrt{\frac{y^2-1}{dy^2+1}} \pmod{p}$, and then solve the sign problem using the identifier bit present in R'.

We can avoid the square-root computation during verification. We propose an

alternative signature-verification Algorithm 6.4 which is a straightforward adaptation of the ECDSA signature-verification algorithm. The correctness of Algorithm 6.4 can be easily proved as follows. We have $S = (r + H(R',A,M)a) \pmod{\lambda}$, that is, $r = S - H(R',A,M)a$. Multiplying both sides by B, we get $rB = SB - H(R',A,M)aB$, that is $R = SB - H(R',A,M)A$. Therefore $y(R) = y(SB - H(R',A,M)A)$. To the best of our knowledge, this way of verifying EdDSA signatures has not been reported or studied in the literature. Like ECDSA signatures, it seems that this EdDSA verification does not reduce the security of EdDSA signatures.

Bernstein et al. [10] introduce Edwards curve over binary fields. Bernstein also presents the concept of batch scalar multiplications over binary Edwards curve in binary field $\mathbb{F}_{2^{251}}$ [5]. Porting all our algorithms to binary Edwards curves is fairly straightforward. We do not deal with these curves in this thesis.

6.2 Batch Verification of EdDSA

Like ECDSA, only the y-coordinate of an Edwards-curve point is sent in an EdDSA signature. An extra bit to identify the correct x-coordinate is included in the signature. All the batch-verification algorithms studied in connection with ECDSA apply equally well to EdDSA signatures. Suppose that we want to verify a batch $(M_1,R'_1,S_1),(M_2,R'_2,S_2),\ldots,(M_t,R'_t,S_t)$ of t EdDSA signatures. Let R_i be the point corresponding to R'_i. We combine the individual verification equations for the t signatures as:

$$\left(\sum_{i=1}^{t} S_i\right) B - \sum_{i=1}^{t} H(R'_i,A_i,M_i)A_i = \sum_{i=1}^{t} R_i. \tag{6.3}$$

If all the signatures are from the same signer, that is, $A_1 = A_2 = \cdots = A_t = A$, then Eqn (6.3) simplifies to:

$$\left(\sum_{i=1}^{t} S_i\right) B - \left(\sum_{i=1}^{t} H(R'_i,A_i,M_i)\right) A = \sum_{i=1}^{t} R_i. \tag{6.4}$$

Eqn (6.4) requires only two scalar multiplications. Unlike ECDSA, an EdDSA signature contains an extra bit of information to identify the x-coordinate of R uniquely (after solving a quadratic equation). We can compute the full Edwards-curve point R_i from R'_i for all i. This calls for t square-root computations modulo p. This algorithm is similar to Algorithm N$'$ of Chapter 3 and is called Algorithm EdN$'$ here. If the extra

bit is not available in the EdDSA signature (or is ignored) to uniquely distinguish the x-coordinate, we have to try all the 2^t combinations of points to verify the batch. We call this naive method Algorithm EdN. The original EdDSA paper [8] recommends Algorithm EdN$'$ as the default batch-verification algorithm.

6.2.1 Adaptation of Algorithm S2$'$

We can remove the overhead of square-root computations altogether. The adaptation of Algorithm S2$'$ can solve this problem. Let us call this adapted version Algorithm EdS2$'$. We first divide the t Edwards-curve points $R_1, R_2, \ldots, R_t$ in two groups. Then, we rewrite Eqn (6.3) as:

$$\left(\sum_{i=1}^{\lceil \frac{t}{2} \rceil} R_i\right) = \left(\sum_{i=1}^{t} S_i\right) B - \left(\sum_{i=1}^{t} H(R'_i, A_i, M_i)\right) A - \left(\sum_{i=\lceil \frac{t}{2} \rceil + 1}^{t} R_i\right). \tag{6.5}$$

We treat the x-coordinates of the points R_i as symbols and compute the symbolic sum of the two sides of Eqn (6.5). Let the symbolic sum on the left-hand side of Eqn (6.5) be Q_1, and that on the right-hand side be Q_2. For a valid batch, Q_1 and Q_2 are two symbolic representations of the same point. We have $y(Q_1) \in \mathbb{F}_p\left[x_1, x_2, \ldots, x_{\lceil \frac{t}{2} \rceil}\right]$ and $y(Q_2) \in \mathbb{F}_p\left[x_{\lceil \frac{t}{2} \rceil + 1}, x_{\lceil \frac{t}{2} \rceil + 2}, \ldots, x_t\right]$. Let

$$\phi = y(Q_1) - y(Q_2),$$

so ϕ is a polynomial in $\mathbb{F}_p[x_1, x_2, \ldots, x_t]$. In ϕ, the maximum degree of any x_i is 1. We write ϕ as $ux_1 + v$, where $u, v \in \mathbb{F}_p[x_2, \ldots, x_t]$. Multiplying ϕ with $ux_1 - v$, we get

$$(ux_1 - v)\phi = (ux_1 - v)(ux_1 + v) = u^2 x_1^2 - v^2.$$

Substituting x_1^2 by $\frac{y_1^2 - 1}{dy_1^2 + 1}$, we get $\phi' = (ux_1 - v)\phi = u^2\left(\frac{y_1^2 - 1}{dy_1^2 + 1}\right) - v^2$. To keep the degrees of all remaining x_i to $\leqslant 1$, a substitution phase follows this elimination, in which we replace x_i^2 by $\frac{y_i^2 - 1}{dy_i^2 + 1}$ for all $i = 2, 3, \ldots, t$. Using the same procedure, we eliminate all the symbolic x-coordinates $x_2, x_3, \ldots, x_t$ one by one. At the end, if we obtain the zero polynomial, we accept the batch of signatures, else we reject it.

6.2.2 Edwards-Curve Summation Polynomials and Adaptation of Algorithm SP

Here, we mention the adaptation necessary to make Algorithm SP of Chapter 4 work for EdDSA batch verification. The two base cases f_2 and f_3 of Edwards-curve summation polynomials, and the recurrence relation to compute the summation polynomial f_t for $t \geqslant 4$ are:

$$\begin{aligned} f_2(y_1, y_2) &= y_1 - y_2, \\ f_3(y_1, y_2, y_3) &= (V - d^2 U y_1^2 y_2^2) y_3^2 - 2y_1 y_2 (V + dU) y_3 + (V y_1^2 y_2^2 - U), \\ &\quad \text{where } U = (y_1^2 - 1)(y_2^2 - 1) \text{ and } V = (dy_1^2 + 1)(dy_2^2 + 1), \\ f_t(y_1, y_2, \ldots, y_t) &= \mathrm{Res}_Y(f_{t-k}(y_1, \ldots, y_{t-k-1}, Y), f_{k+2}(y_{t-k}, \ldots, y_t, Y)) \\ &\quad \text{for } t \geqslant 4 \text{ and for any } k \text{ in the range } 1 \leqslant k \leqslant t-3. \end{aligned}$$

The summation polynomial f_t evaluated at the t arguments $y_1, y_2, \ldots, y_t$ is zero if and only if there exists an x_i in $\overline{\mathbb{F}}_p$ for each y_i, where $1 \leqslant i \leqslant t$, such that $-x_i^2 + y_i^2 = 1 + dx_i^2 y_i^2$ and $\sum_{i=1}^{t}(x_i, y_i) = \mathscr{O}$. If the batch-verification condition of Eqn (6.3) or (6.4) is expressed as $\sum_{i=1}^{t}(x_i, y_i) + (-\alpha, \beta) = \mathscr{O}$, it therefore suffices to check whether $f_{t+1}(y_1, y_2, \ldots, y_t, \beta) = 0$. We call this adapted version Algorithm EdSP. To restrict our attention to curve points defined over $\mathbb{F}_p$ only, we need to carry out the sanity check introduced in Chapter 4. The sanity check for Edwards curves follows the same procedure as for elliptic curves (that is, we check whether the Legendre symbol $\left(\frac{(y_i^2-1)/(dy_i^2+1)}{p}\right)$ is equal to 1).

6.2.3 Adaptation of Algorithm S3

Algorithm S3 can be adapted to EdDSA signatures as follows. Let us call this adapted version Algorithm EdS3. We first divide the t Edwards-curve points $R_1, R_2, \ldots, R_t$ in two groups. Similar to Algorithm S2′, we rewrite Eqn (6.3) as:

$$\left(\sum_{i=1}^{\lceil \frac{t}{2} \rceil} R_i\right) = \left(\sum_{i=1}^{t} S_i\right) B - \left(\sum_{i=1}^{t} H(R_i', A_i, M_i)\right) A - \left(\sum_{i=\lceil \frac{t}{2} \rceil + 1}^{t} R_i\right). \tag{6.6}$$

We treat the x-coordinates of the points R_i as symbols and compute the symbolic sum of the two sides of Eqn (6.6). Let the symbolic sum on the left-hand side of Eqn (6.6)

be Q_1, and that on the right-hand side be Q_2. For a valid batch, Q_1 and Q_2 are two symbolic representations of the same point. We have $y(Q_1) \in \mathbb{F}_p\left[x_1, x_2, \ldots, x_{\lceil \frac{t}{2} \rceil}\right]$ and $y(Q_2) \in \mathbb{F}_p\left[x_{\lceil \frac{t}{2} \rceil + 1}, x_{\lceil \frac{t}{2} \rceil + 2}, \ldots, x_t\right]$. Let

$$\phi_1 = Y - y(Q_1) \text{ and } \phi_2 = Y - y(Q_2),$$

so ϕ_1 and ϕ_2 are polynomials in $\mathbb{F}_p[x_1, x_2, \ldots, x_t]$. In ϕ_1 and ϕ_2, the maximum degree of any x_i is 1. Then, we use the elimination method of Algorithm S2′ on ϕ_1 and ϕ_2 to create two polynomials in Y; let us call them $F_t^{(1)}$ and $F_t^{(2)}$. We take the resultant of these two polynomials with respect to the variable Y and accept the batch if and only if the resultant is zero. We call this Algorithm EdS3.

If we replace the last resultant computation of Algorithms EdSP and EdS3 by a gcd computation, we arrive at the practically faster variants Algorithms EdSP_{gcd} and EdS3_{gcd}.

6.3 Randomization of EdDSA Batch-Verification Algorithms

EdDSA signatures can be randomized easily by methods similar to the randomization methods for ECDSA. For randomly chosen multipliers $\xi_1, \xi_2, \ldots, \xi_t$, we now verify whether the following equality holds:

$$\left(\sum_{i=1}^{t} \xi_i S_i\right) B - \sum_{i=1}^{t} \xi_i H(R_i', A_i, M_i) A_i = \sum_{i=1}^{t} \xi_i R_i. \tag{6.7}$$

For the case of the same signer, that is, $A_1 = A_2 = \cdots = A_t = A$, Eqn (6.7) simplifies to:

$$\left(\sum_{i=1}^{t} \xi_i S_i\right) B - \left(\sum_{i=1}^{t} \xi_i H(R_i', A_i, M_i)\right) A = \sum_{i=1}^{t} \xi_i R_i. \tag{6.8}$$

The default batch-verification algorithm for EdDSA is EdN′, in which we explicitly and uniquely compute the points R_i by square-root computations modulo p. Subsequently, their multiples $\xi_i R_i$ can be computed *numerically*. We finally check whether the condition of Eqn (6.7) or (6.8) holds. The process does not involve any symbolic or summation-polynomial computation. In a variant denoted by EdN, we assume that R_i cannot be uniquely determined, so we need to try all possible

combinations of the signs of x_i. For each combination, randomization proceeds numerically as in the case of EdN$'$.

We may, however, ignore the presence of the extra bit in R_i' identifying the correct value of x_i. By doing so, we can adapt the randomized Algorithms EdS2$'$ and EdSP to work for EdDSA. This is motivated by a need to avoid costly square-root computations of Algorithm EdN$'$.

In order to apply Algorithm EdS2$'$ to the batch-verification Eqn (6.7) or (6.8), it suffices to compute the y-coordinates of all $\xi_i R_i$. As in the case of ECDSA, we can uniquely compute $y(\xi_i R_i)$ from the knowledge of ξ_i and $y(R_i)$ alone. More precisely, let $R = (x,y)$ be a point on the Edwards curve. Any multiple uR of R can be expressed as (hx,k), where $h,k \in \mathbb{F}_p$ are fully determined by (u and) the y-coordinate of R. R itself is so expressed with $h = 1$ and $k = y$. The sum of two multiples $P_1 = (h_1 x, k_1)$ and $P_2 = (h_2 x, y_2)$ of R is $P_1 + P_2 = (h_3 x, k_3)$, where

$$\begin{aligned} h_3 &= (h_1 k_2 + h_2 k_1)/(1 + d h_1 h_2 k_1 k_2 f), \\ k_3 &= (k_1 k_2 + h_1 h_2 f)/(1 - d h_1 h_2 k_1 k_2 f), \end{aligned}$$

with f precomputed as $f = x^2 = (y^2 - 1)/(dy^2 + 1) \in \mathbb{F}_p$. For Edwards curves, the doubling formula is the same as the addition formula. That is, the double of $P_1 = (h_1 x, k_1)$ is $2P_1 = (h_4 x, k_4)$, where

$$\begin{aligned} h_4 &= 2h_1 k_1/(1 + d h_1^2 k_1^2 f), \\ k_4 &= (k_1^2 + h_1^2 f)/(1 - d h_1^2 k_1^2 f). \end{aligned}$$

We henceforth refer to this computation of $y(\xi_i R_i)$ as the *seminumeric* randomization method.

We can also use *Montgomery ladders* [42] to compute $y(\xi_i R_i)$. For deriving the Montgomery-ladder formulas, let $P_1 = (h_1, k_1)$ and $P_2 = (h_2, k_2)$ be two points on the curve. For point addition, we need the y-coordinate of the point $P_1 - P_2$ as follows.

$$y(P_1 + P_2) = \frac{2k_1 k_2 (1 + d h_1^2 h_2^2)}{1 - d h_1^2 h_2^2 (k_1 k_2)^2} - y(P_1 - P_2).$$

Here, $h_i^2 = (k_i^2 - 1)/(d k_i^2 + 1)$ for $i = 1, 2$. Finally, point doubling uses the formula

$$y(2P_1) = \frac{k_1^2 + h_1^2}{1 - d h_1^2 k_1^2},$$

where $h_1^2 = (k_1^2 - 1)/(dk_1^2 + 1)$. These formulas can be easily converted to projective coordinates.

Let us now theoretically compare the performance of the seminumeric method with that of the Montgomery-ladder method. Let $P_1 = (\alpha_1 x, \beta_1, \gamma_1)$ and $P_2 = (\alpha_2 x, \beta_2, \gamma_2)$ be two points on the curve in standard projective coordinates. We precompute the value $f_x = x^2 = \frac{y^2-1}{dy^2+1}$. The seminumeric method computes the sum $P_3 = P_1 + P_2 = (\alpha_3 x, \beta_3, \gamma_3)$ and the double $P_4 = 2P_1 = (\alpha_4 x, \beta_4, \gamma_4)$ as given below:

Point Addition

$A = \gamma_1 \cdot \gamma_2,\ B = A^2,\ C = \alpha_1 \cdot \alpha_2,\ C_1 = C \cdot f_x,\ D = \beta_1 \cdot \beta_2,\ E = d \cdot C_1 \cdot D,\ F = B - E,$
$G = B + E,\ \alpha_3 = A \cdot F \cdot ((\alpha_1 + \beta_1) \cdot (\alpha_2 + \beta_2) - C - D),\ \beta_3 = A \cdot G \cdot (D + C_1),$
$\gamma_3 = F \cdot G.$

Point Doubling

$B = (\alpha_1 + \beta_1)^2,\ C = \alpha_1^2,\ C_1 = C \cdot f_x,\ D = \beta_1^2,\ E_1 = D - C_1,\ E_2 = C + D,\ H = \gamma_1^2,$
$J = 2 \cdot H - E_1,\ \alpha_4 = (B - E_2) \cdot J,\ \beta_4 = E_1 \cdot (C_1 + D),\ \gamma_4 = E_1 \cdot J.$

Each of seminumeric point addition and point doubling requires one extra field multiplication than the optimized implementation given in [9]. More precisely, seminumeric point addition and doubling take $(11M + 1S)$ and $(4M + 4S)$ field operations respectively (ignoring the negligible time consumed by multiplication by d and field addition).

The Montgomery-ladder method requires $(18M + 6S)$ field operations for each addition and doubling combined in each iteration. Let $P_1 = (y_1, z_1)$ and $P_2 = (y_2, z_2)$ be two points on the Edwards curve with $P_2 - P_1 = (y, 1)$. Let $P_1 + P_2 = P_3 = (y_3, z_3)$ and $2P_2 = P_4 = (y_4, z_4)$. The formulas for computing P_3 and P_4 in standard projective coordinates are given below:

$y_{12} = y_1^2;\ y_{22} = y_2^2;\ z_{12} = z_1^2;\ z_{22} = z_2^2;\ y_{11} = (y_{12} - z_{12});\ y_{21} = (y_{22} - z_{22});$
$d_{y_1} = (d \cdot y_{12} + z_{12});\ d_{y_2} = (d \cdot y_{22} + z_{22});\ t_1 = d_{y_1} \cdot d_{y_2};\ t_2 = y_{11} \cdot y_{21};$
$t_3 = y_1 \cdot y_2;\ t_4 = z_1 \cdot z_2;\ N = 2 \cdot (t_1 + d \cdot t_2) \cdot t_3 \cdot t_4;\ D = t_4^2 \cdot t_1 - d_2 \cdot t_2 \cdot t_3^2;$
$y_3 = N - y \cdot D;\ z_3 = D;\ y_4 = (y_{22} \cdot d_{y_2} + y_{21} \cdot z_{22});\ z_4 = (d_{y_2} \cdot z_{22} - d \cdot y_{22} \cdot y_{21})$

We can use any windowed variant of point multiplication in the seminumeric point multiplication method. On the contrary, no effective windowed variant is known for Montgomery ladders. Moreover, the practical ladder described in [43] is efficient only for constant multipliers, which is not the case with randomized batch verification. We therefore use only the binary ladder.

Let us use l-bit randomizers. If we use the w-NAF method in the seminumeric computation, the precomputation stage needs $(4M+4S)+(2^{w-1}-1)(11M+1S)$ field operations, and $\left(\frac{l}{w+1}\right)(11M+1S)$ field operations are required to perform the scalar multiplication. The seminumeric scalar multiplication is faster than the Montgomery-ladder method if

$$(4M+4S)+(2^{w-1}-1)(11M+1S)+(4M+4S)l+\left(\frac{l(11M+1S)}{w+1}\right)\leqslant l(18M+6S).$$

Putting $w=4$ and assuming $1M\approx 1S$, we deduce that for $l\geqslant 8$ the seminumeric method is faster than the Montgomery-ladder method.

6.4 Experimental Results

The algorithms are implemented in a 2.33 GHz Xeon server running Ubuntu Linux Version 2012 LTS. The algorithms are implemented using the GP/PARI calculator (version 2.5.0 compiled by the GNU C compiler 4.6.2). We have used the symbolic-computation facilities of the calculator in our programs. All other functions (like scalar multiplication and square-root computation) are written as subroutines with minimal function-call overheads. Since the algorithms are evaluated in terms of the numbers of field operations, this gives a fair comparison of experimental data with the theoretical estimates. We have implemented windowed, w-NAF and frac-w-NAF methods for square-root computations and for numeric and seminumeric randomization methods. We have used affine and standard projective coordinates. We have performed all the experiments on the Edwards curve Ed25519 [8].

Table 6.1 lists the overheads associated with all the batch-verification algorithms. Table 6.2 shows the square-root computation times obtained by various windowed algorithms. The times needed to carry out double and Montgomery-ladder scalar multiplications are supplied in Table 6.3. If storage overhead in the computing

Table 6.1: Overhead (in ms) of different batch-verification algorithms for EdDSA

Batch Size	Algorithm						
t	EdN	EdN$'$	EdS2$'$	EdS3	EdS3$_{\text{gcd}}$	EdSP	EdSP$_{\text{gcd}}$
2	0.08	0.03	0.05	0.05	0.05	0.05	0.05
3	0.24	0.04	0.11	0.16	0.16	0.11	0.09
4	0.63	0.05	0.26	0.30	0.30	0.14	0.13
5	1.56	0.07	0.53	0.61	0.57	0.28	0.27
6	3.70	0.08	0.97	0.99	0.93	1.40	1.09
7	8.64	0.10	2.03	1.84	1.63	2.70	1.82

Table 6.2: Times (in ms) of square-root computations in the underlying field

↓ Algorithm			Times (in ms)
w-numeric (affine)	$w=3$		0.36
	$w=4$		**0.28**
	$w=5$		**0.28**
w-NAF-numeric (affine)	$w=3$		**0.28**
	$w=4$		**0.28**
	$w=5$		0.32
Frac-w-NAF-numeric (affine)	$w=3$	$m=1$	0.33
	$w=4$	$m=1$	**0.28**
		$m=3$	0.32
		$m=5$	0.32
	$w=5$	$m=1$	0.32
		$m=3$	0.32
		$m=5$	0.32
		$m=7$	0.36
		$m=9$	0.32
		$m=11$	0.32
		$m=13$	0.36

Table 6.3: Times (in ms) of the double scalar multiplication and Montgomery-ladder randomization method

Scalars are randomly chosen				
	Double scalar multiplication		Montgomery-ladder multiplication	
Coordinate system	$l = 128$	$l = 255$	$l = 128$	$l = 255$
Affine	3.44	6.78	2.97	5.81
Standard projective	2.03	3.98	1.96	3.84
Additon chains of the scalars are randomly chosen				
	Double scalar multiplication		Montgomery-ladder multiplication	
Coordinate system	$l = 128$	$l = 255$	$l = 128$	$l = 255$
Affine	3.12	6.26	2.81	5.55
Standard projective	1.74	3.43	1.81	3.56

Table 6.4: Times (in ms) of fixed-base double scalar multiplication

Coordinate	t					
	2	3	4	5	6	7
Affine	14.33	18.07	21.75	25.43	29.14	32.80
Standard projective	9.07	11.72	14.37	17.02	19.67	22.30

platform is permitted, one can do fixed-base double scalar multiplication, the results for which are given in Table 6.4. We present the times required for the numeric and seminumeric scalar multiplications in Tables 6.5 and 6.6. In Table 6.5, we use random scalars, so we have to compute the addition chain for each scalar multiplication. The timing figures presented in Table 6.5 include the addition-chain computation times. In Table 6.6, on the other hand, the addition chains are randomly generated. The best results obtained are highlighted and used in speedup computations.

The overall speedup figures obtained by our batch-verification algorithms are listed in Tables 6.7–6.10. In the speedup tables, we include the results using both the default signature-verification Algorithm 6.3 and the ECDSA-like signature-verification Algorithm 6.4.

For batch sizes in the range $2 \leqslant t \leqslant 7$, the speedup obtained by Algorithms EdS2′ and EdSP is competitive with that obtained by the default batch-verification Algorithm EdN′. Algorithms EdS2′ and EdSP outperform Algorithm EdN′ if we use the default Algorithm 6.3 for individual verification. On the other hand, if we use the ECDSA-like verification Algorithm 6.4 for individual verification, Symbolic-

Table 6.5: Times (in ms) of the numeric and seminumeric randomization methods (The scalars are randomly chosen)

↓ Algorithm			Numeric Methods		SemiNumeric Methods	
			$l = 128$	$l = 255$	$l = 128$	$l = 255$
w-numeric (affine)	$w = 3$		2.28	4.49	2.36	4.60
	$w = 4$		2.28	4.37	2.40	4.57
	$w = 5$		2.48	4.40	2.56	4.64
w-NAF-numeric (affine)	$w = 3$		2.32	4.60	2.40	4.81
	$w = 4$		2.24	4.44	2.29	4.60
	$w = 5$		2.28	4.33	2.36	4.45
Frac-w-NAF-numeric (affine)	$w = 3$	$m = 1$	2.48	4.77	2.57	4.97
	$w = 4$	$m = 1$	2.49	4.76	2.52	4.92
		$m = 3$	2.44	4.77	2.53	4.93
		$m = 5$	2.44	4.77	2.52	4.89
	$w = 5$	$m = 1$	2.45	4.65	2.53	4.81
		$m = 3$	2.48	4.64	2.52	4.85
		$m = 5$	2.48	4.68	2.57	4.85
		$m = 7$	2.49	4.72	2.57	4.88
		$m = 9$	2.48	4.73	2.56	4.88
		$m = 11$	2.52	4.72	2.56	4.93
		$m = 13$	2.52	4.72	2.64	4.89
w-numeric (Jacobian projective)	$w = 3$		**1.28**	2.44	1.44	2.72
	$w = 4$		**1.28**	**2.40**	1.44	**2.64**
	$w = 5$		1.40	2.44	1.52	2.72
w-NAF-numeric (Jacobian projective)	$w = 3$		1.32	2.68	1.44	2.92
	$w = 4$		**1.28**	2.57	**1.40**	2.77
	$w = 5$		1.32	2.44	1.44	2.68
Frac-w-NAF-numeric (Jacobian projective)	$w = 3$	$m = 1$	1.49	2.92	1.64	3.12
	$w = 4$	$m = 1$	1.48	2.88	1.60	3.12
		$m = 3$	1.44	2.84	1.64	3.12
		$m = 5$	1.49	2.80	1.64	3.12
	$w = 5$	$m = 1$	1.44	2.80	1.64	3.08
		$m = 3$	1.48	2.81	1.64	3.08
		$m = 5$	1.48	2.81	1.64	3.08
		$m = 7$	1.48	2.84	1.64	3.09
		$m = 9$	1.52	2.84	1.69	3.12
		$m = 11$	1.53	2.81	1.64	3.08
		$m = 13$	1.52	2.85	1.64	3.12

Table 6.6: Times (in ms) of the numeric and seminumeric randomization methods (The addition chains are randomly chosen)

↓ Algorithm			Numeric Methods		SemiNumeric Methods	
			$l = 128$	$l = 255$	$l = 128$	$l = 255$
w-numeric (affine)	$w = 3$		2.24	4.45	2.32	4.48
	$w = 4$		2.24	4.29	2.32	4.41
	$w = 5$		2.40	4.37	2.53	4.53
w-NAF-numeric (affine)	$w = 3$		2.16	4.20	2.24	4.53
	$w = 4$		2.08	4.08	2.21	4.36
	$w = 5$		2.13	3.96	2.24	4.24
Frac-w-NAF-numeric (affine)	$w = 3$	$m = 1$	2.16	4.12	2.24	4.37
	$w = 4$	$m = 1$	2.12	4.13	2.20	4.33
		$m = 3$	2.08	4.09	2.20	4.37
		$m = 5$	2.12	4.08	2.25	4.28
	$w = 5$	$m = 1$	2.08	4.00	2.20	4.16
		$m = 3$	2.12	4.05	2.25	4.25
		$m = 5$	2.12	4.04	2.24	4.21
		$m = 7$	2.12	4.08	2.28	4.20
		$m = 9$	2.12	4.09	2.29	4.29
		$m = 11$	2.16	4.04	2.32	4.21
		$m = 13$	2.20	4.05	2.28	4.24
w-numeric (Jacobian projective)	$w = 3$		1.20	2.36	1.36	2.68
	$w = 4$		1.20	2.32	1.36	2.56
	$w = 5$		1.36	2.40	1.48	2.65
w-NAF-numeric (Jacobian projective)	$w = 3$		1.20	2.33	1.32	2.60
	$w = 4$		**1.12**	2.20	**1.28**	**2.44**
	$w = 5$		**1.12**	**2.13**	**1.28**	**2.44**
Frac-w-NAF-numeric (Jacobian projective)	$w = 3$	$m = 1$	1.17	2.28	1.33	2.56
	$w = 4$	$m = 1$	1.13	2.24	**1.28**	2.48
		$m = 3$	**1.12**	2.24	**1.28**	**2.44**
		$m = 5$	1.16	2.20	**1.28**	2.48
	$w = 5$	$m = 1$	1.16	2.16	1.32	**2.44**
		$m = 3$	1.21	2.16	1.32	2.48
		$m = 5$	1.20	2.17	1.32	2.48
		$m = 7$	1.20	2.20	1.32	2.48
		$m = 9$	1.21	2.24	1.32	2.52
		$m = 11$	1.20	2.20	1.36	2.56
		$m = 13$	1.20	2.20	1.41	2.49

Table 6.7: Speedup (over individual verification) obtained by different batch-verification methods with randomly chosen scalars
(Fixed-base double scalar multiplication is *not* used)

Batch Verification Algorithm	Randomization Algorithm	Batch Size	Algorithm 6.3		Algorithm 6.4	
			None*	$l = 128$	None*	$l = 128$
EdN	Numeric	2	1.84	1.19	1.72	1.11
		3	2.53	1.44	2.36	1.34
		4	2.97	1.57	2.78	1.47
		5	3.07	1.60	2.87	1.49
		6	2.73	1.50	2.55	1.40
		7	2.05	1.27	1.91	1.18
EdN′	Numeric	2	1.86	1.29	1.74	1.21
		3	2.63	1.86	2.46	1.74
		4	3.31	1.85	3.09	1.73
		5	3.91	2.25	3.65	2.10
		6	4.45	2.17	4.16	2.02
		7	4.94	2.47	4.61	2.31
EdS2′	Seminumeric	2	2.11	1.25	1.98	1.17
		3	3.12	1.54	2.92	1.44
		4	4.02	1.73	3.75	1.62
		5	4.72	1.85	4.41	1.73
		6	5.16	1.91	4.82	1.79
		7	4.96	1.89	4.64	1.76
EdS3	Seminumeric	2	2.11	1.25	1.98	1.17
		3	3.09	1.53	2.88	1.43
		4	3.98	1.72	3.72	1.61
		5	4.64	1.84	4.34	1.72
		6	5.14	1.91	4.80	1.79
		7	5.12	1.91	4.79	1.78
EdS3_{gcd}	Seminumeric	2	2.11	1.25	1.98	1.17
		3	3.09	1.53	2.88	1.43
		4	3.98	1.72	3.72	1.61
		5	4.68	1.84	4.37	1.72
		6	5.21	1.92	4.86	1.79
		7	5.32	1.94	4.97	1.81
EdSP	Seminumeric	2	2.11	1.25	1.98	1.17
		3	3.12	1.54	2.92	1.44
		4	4.14	1.75	3.86	1.64
		5	5.00	1.89	4.67	1.77
		6	4.75	1.85	4.44	1.73
		7	4.46	1.81	4.17	1.69
EdSP_{gcd}	Seminumeric	2	2.11	1.25	1.98	1.17
		3	3.14	1.55	2.93	1.44
		4	4.15	1.75	3.87	1.64
		5	5.01	1.89	4.68	1.77
		6	5.04	1.90	4.71	1.77
		7	5.14	1.91	4.80	1.79

* without randomization

Table 6.8: Speedup (over individual verification) obtained by different batch-verification methods with randomly chosen addition chains
(Fixed-base double scalar multiplication is *not* used)

Batch Verification Algorithm	Randomization Algorithm	Batch Size	Algorithm 6.3		Algorithm 6.4	
			None*	$l = 128$	None*	$l = 128$
EdN	Numeric	2	1.84	1.24	1.72	1.16
		3	2.53	1.52	2.36	1.42
		4	2.97	1.67	2.78	1.56
		5	3.07	1.70	2.87	1.59
		6	2.73	1.59	2.55	1.49
		7	2.05	1.33	1.91	1.24
EdN′	Numeric	2	1.86	1.35	1.74	1.26
		3	2.63	1.94	2.46	1.81
		4	3.31	1.98	3.09	1.85
		5	3.91	2.39	3.65	2.24
		6	4.45	2.34	4.16	2.18
		7	4.94	2.66	4.61	2.49
EdS2′	Seminumeric	2	2.11	1.29	1.98	1.21
		3	3.12	1.61	2.92	1.51
		4	4.02	1.82	3.75	1.70
		5	4.72	1.95	4.41	1.82
		6	5.16	2.02	4.82	1.89
		7	4.96	1.99	4.64	1.86
EdS3	Seminumeric	2	2.11	1.29	1.98	1.21
		3	3.09	1.60	2.88	1.50
		4	3.98	1.81	3.72	1.69
		5	4.64	1.94	4.34	1.81
		6	5.14	2.02	4.80	1.89
		7	5.12	2.02	4.79	1.88
EdS3_{gcd}	Seminumeric	2	2.11	1.29	1.98	1.21
		3	3.09	1.60	2.88	1.50
		4	3.98	1.81	3.72	1.69
		5	4.68	1.95	4.37	1.82
		6	5.21	2.03	4.86	1.90
		7	5.32	2.05	4.97	1.91
EdSP	Seminumeric	2	2.11	1.29	1.98	1.21
		3	3.12	1.61	2.92	1.51
		4	4.14	1.84	3.86	1.72
		5	5.00	2.00	4.67	1.87
		6	4.75	1.96	4.44	1.83
		7	4.46	1.91	4.17	1.78
EdSP_{gcd}	Seminumeric	2	2.11	1.29	1.98	1.21
		3	3.14	1.62	2.93	1.51
		4	4.15	1.85	3.87	1.72
		5	5.01	2.00	4.68	1.87
		6	5.04	2.00	4.71	1.87
		7	5.14	2.02	4.80	1.89

* without randomization

Table 6.9: Speedup (over individual verification) obtained by different batch-verification methods with randomly chosen scalars
(Fixed-base double scalar multiplication is *used*)

Batch Verification Algorithm	Randomization Algorithm	Batch Size	Algorithm 6.3		Algorithm 6.4	
			None*	$l = 128$	None*	$l = 128$
EdN	Numeric	2	2.08	1.34	1.96	1.26
		3	2.48	1.41	2.32	1.32
		4	2.70	1.43	2.51	1.32
		5	2.65	1.38	2.45	1.28
		6	2.28	1.25	2.10	1.15
		7	1.66	1.03	1.53	0.95
EdN′	Numeric	2	2.11	1.46	1.98	1.37
		3	2.58	1.83	2.41	1.70
		4	3.01	1.68	2.79	1.56
		5	3.38	1.94	3.12	1.80
		6	3.72	1.81	3.43	1.67
		7	4.02	2.01	3.69	1.85
EdS2′	Seminumeric	2	2.39	1.41	2.25	1.33
		3	3.07	1.52	2.87	1.41
		4	3.65	1.57	3.39	1.46
		5	4.08	1.60	3.77	1.48
		6	4.31	1.60	3.97	1.47
		7	4.04	1.53	3.71	1.41
EdS3	Seminumeric	2	2.39	1.41	2.25	1.33
		3	3.03	1.51	2.83	1.41
		4	3.62	1.57	3.36	1.45
		5	4.01	1.59	3.71	1.47
		6	4.30	1.60	3.96	1.47
		7	4.17	1.55	3.83	1.43
EdS3_{gcd}	Seminumeric	2	2.39	1.41	2.25	1.33
		3	3.03	1.51	2.83	1.41
		4	3.62	1.57	3.36	1.45
		5	4.05	1.59	3.74	1.47
		6	4.35	1.60	4.01	1.48
		7	4.32	1.57	3.98	1.45
EdSP	Seminumeric	2	2.39	1.41	2.25	1.33
		3	3.07	1.52	2.87	1.41
		4	3.76	1.59	3.49	1.48
		5	4.32	1.64	4.00	1.51
		6	3.97	1.55	3.66	1.43
		7	3.63	1.47	3.34	1.35
EdSP_{gcd}	Seminumeric	2	2.39	1.41	2.25	1.33
		3	3.09	1.52	2.88	1.42
		4	3.77	1.60	3.50	1.48
		5	4.33	1.64	4.00	1.51
		6	4.21	1.59	3.88	1.46
		7	4.18	1.56	3.84	1.43

* without randomization

Table 6.10: Speedup (over individual verification) obtained by different batch-verification methods with randomly chosen addition chains
(Fixed-base double scalar multiplication is *used*)

Batch Verification Algorithm	Randomization Algorithm	Batch Size	Algorithm 6.3		Algorithm 6.4	
			None*	$l = 128$	None*	$l = 128$
EdN	Numeric	2	2.08	1.40	1.96	1.32
		3	2.48	1.49	2.32	1.39
		4	2.70	1.52	2.51	1.41
		5	2.65	1.47	2.45	1.36
		6	2.28	1.33	2.10	1.22
		7	1.66	1.08	1.53	0.99
EdN′	Numeric	2	2.11	1.53	1.98	1.44
		3	2.58	1.91	2.41	1.78
		4	3.01	1.80	2.79	1.67
		5	3.38	2.07	3.12	1.91
		6	3.72	1.95	3.43	1.80
		7	4.02	2.16	3.69	1.99
EdS2′	Seminumeric	2	2.39	1.46	2.25	1.38
		3	3.07	1.58	2.87	1.48
		4	3.65	1.65	3.39	1.54
		5	4.08	1.69	3.77	1.56
		6	4.31	1.69	3.97	1.56
		7	4.04	1.62	3.71	1.49
EdS3	Seminumeric	2	2.39	1.46	2.25	1.38
		3	3.03	1.57	2.83	1.47
		4	3.62	1.65	3.36	1.53
		5	4.01	1.68	3.71	1.55
		6	4.30	1.69	3.96	1.55
		7	4.17	1.64	3.83	1.51
EdS3_{gcd}	Seminumeric	2	2.39	1.46	2.25	1.38
		3	3.03	1.57	2.83	1.47
		4	3.62	1.65	3.36	1.53
		5	4.05	1.68	3.74	1.55
		6	4.35	1.70	4.01	1.56
		7	4.32	1.67	3.98	1.53
EdSP	Seminumeric	2	2.39	1.46	2.25	1.38
		3	3.07	1.58	2.87	1.48
		4	3.76	1.68	3.49	1.56
		5	4.32	1.73	4.00	1.60
		6	3.97	1.63	3.66	1.51
		7	3.63	1.55	3.34	1.43
EdSP_{gcd}	Seminumeric	2	2.39	1.46	2.25	1.38
		3	3.09	1.59	2.88	1.48
		4	3.77	1.68	3.50	1.56
		5	4.33	1.73	4.00	1.60
		6	4.21	1.67	3.88	1.54
		7	4.18	1.64	3.84	1.51

* without randomization

computation algorithms outperforms Algorithm EdN′ for batch sizes $t \leqslant 7$, and Algorithm EdSP is faster than Algorithm EdN′ for batch sizes $\leqslant 5$. The use of fixed-base double scalar multiplication, if applicable, benefits individual verification much in the same way as it does for NIST prime curves.

In short, replacing square-root computations by symbolic or resultant computations does not degrade the batch-verification process, so long as we restrict only to small batches of signatures. However, the overhead of the default batch-verification algorithm EdN′ increases linearly with the batch size, whereas that of EdS2′, EdS3 or EdSP increases exponentially. Consequently, EdN′ must eventually take over the exponential algorithms (not demonstrated in the experimental results though).

6.5 Chapter Summary

In this chapter, we port several batch-verification algorithms proposed for ECDSA to EdDSA signatures. We also address the issues of randomizing the batch-verification process. Our experimental results demonstrate that the default batch-verification algorithm proposed for EdDSA can be slightly improved by using the new developments based on symbolic and resultant computations, at least for small batch sizes.

Chapter 7

Conclusion

7.1 Work Reported in the Thesis

In this thesis, we have proposed several batch-verification algorithms for ECDSA and EdDSA signatures. To the best of our knowledge, these are the first algorithms proposed for standard ECDSA signatures. The main technical novelty of this work is the effective use of

- symbolic computations, and
- summation polynomials

for this problem.

We have started with two batch-verification algorithms based on taking square-roots in the underlying fields. We call these naive algorithms (N and N$'$). To avoid expensive square-root computations, we have proposed a batch-verification Algorithm S1 which replaces square-root computations by symbolic manipulations. Algorithm S1 also provide the theoretical basis for our next batch-verification Algorithm S2 which also uses symbolic manipulations. Theoretically and practically, Algorithm S2 is more efficient than Algorithm S1. We have also introduced two faster variants of Algorithms S1 and S2, called Algorithms S1$'$ and S2$'$. This efficiency is achieved by taking a one-level divide-and-conquer approach.

In Algorithm SP, we have replaced symbolic manipulations by computations of elliptic-curve summation polynomials. We have provided an alternative method of computing the summation polynomials based on symbolic manipulations. This leads to our third symbolic-computation algorithm termed Algorithm S3. Algorithms $\mathrm{SP}_{\mathrm{gcd}}$ and $\mathrm{S3}_{\mathrm{gcd}}$ are two practically faster variants which replace the last resultant computation (the most expensive operation in Algorithms SP and S3) by a gcd computation. Algorithms SP and $\mathrm{SP}_{\mathrm{gcd}}$ are theoretically the fastest batch-verification algorithms, and use $\mathrm{O}(2^t)$ field operations to verify a batch of t ECDSA signatures.

To avoid forgery attacks, we have randomized all these algorithms. In this thesis, we have discussed three randomization methods: a numeric method, a seminumeric method and the Montgomery-ladder method. Our theoretical estimates and practical experiments show that the proposed seminumeric randomization method is the most suitable for *standard* ECDSA signatures.

We have also ported all our algorithms to Koblitz curves and Edward Curves, and provided all necessary technical details, analyses and experimental data in the thesis.

7.2 Future Directions

Our study introduces some interesting open problems in the context of standard ECDSA batch verification. These problems are listed below:

- Whether batch verification of standard ECDSA signatures can be carried out in $\mathrm{o}(m)$ time (or even in polynomial time in t) remains the most important open problem.
- Randomization of our batch-verification algorithms reduces the speedup factors considerably. Whether it is possible to reduce the overhead of the randomization stage without degrading the security stands as another important open problem.
- Our algorithms—particularly the randomized versions—are effective only when all signatures in a batch belong to the same signer. The question of how we can efficiently handle the situation where all the signatures come from different signers remains unanswered.

Publications from this Work

- **Journal**

  1. *S. Karati*, A. Das, D. Roychowdhury, B. Bellur, D. Bhattacharya and A. Iyer, *New algorithms for batch verification of standard ECDSA signatures*, Journal of Cryptographic Engineering, Springer, DOI 10.1007/s13389-014-0082-x, Online ISSN 2190-8516.

- **Conference**

  1. *S. Karati*, A. Das, D. Roychowdhury, B. Bellur, D. Bhattacharya and A. Iyer, *Batch verification of ECDSA signatures*, 5th International Conference on Cryptology in Africa (AfricaCrypt 2012), Lecture Notes in Computer Science #7374, pp 1–18, Jul 10–12, 2012, Ifrane, Morocco.
  2. *S. Karati* and A. Das, *Faster batch verification of standard ECDSA signatures using summation polynomials*, 12th International Conference on Applied Cryptography and Network Security (ACNS 2014), Lecture Notes in Computer Science #8479, pp 438–456, Jun 10–13, 2014, Lausanne, Switzerland.
  3. *S. Karati*, A. Das and D. Roychoudhury, *Randomized batch verification of standard ECDSA signatures*, to appear in the Fourth International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE 2014), Oct 18–22, 2014, Pune, India.
  4. *S. Karati* and A. Das, *Batch verification of EdDSA signatures*, to appear in the Fourth International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE 2014), Oct 18–22, 2014, Pune, India.

Bibliography

[1] ANSI, *Public key cryptography for the financial services industry, the elliptic curve digital signature algorithm (ECDSA)*. http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.62%3A2005&sourcekeyword=_inurl:webstore.ansi.org%23inurl:sku%3Dansi%2Bx9&source=google&adgroup=x9&gclid=CPSKj8rd9MACFQKSjgodS54AUg, 2005.

[2] A. ANTIPA, D. BROWN, R. GALLANT, R. LAMBERT, R. STRUIK, AND S. VANSTONE, *Accelerated verification of ECDSA signatures*, in SAC, vol. 3897 of Lecture Notes in Computer Science, Springer, 2006, pp. 307–318.

[3] L. BATINA, N. MENTENS, K. SAKIYAMA, B. PRENEEL, AND I. VERBAUWHEDE, *Low-cost elliptic curve cryptography for wireless sensor networks*, in Security and Privacy in Ad-Hoc and Sensor Networks, vol. 4357 of Lecture Notes in Computer Science, Springer, 2006, pp. 6–17.

[4] M. BELLARE, J. A. GARAY, AND T. RABIN, *Fast batch verification for modular exponentiation and digital signatures*, in EUROCRYPT, vol. 1403 of Lecture Notes in Computer Science, Springer, 1998, pp. 236–250.

[5] D. J. BERNSTEIN, *Batch binary Edwards*, in CRYPTO, vol. 5677 of Lecture Notes in Computer Science, Spinger, 2009, pp. 317–336.

[6] D. J. BERNSTEIN, P. BIRKNER, M. JOYE, T. LANGE, AND C. PETERS, *Twisted Edwards curves*, in AFRICACRYPT, vol. 5023 of Lecture Notes in Computer Science, Springer, 2008, pp. 389–405.

[7] D. J. BERNSTEIN, J. DOUMEN, T. LANGE, AND J.-J. OOSTERWIJK, *Faster batch forgery identification*, in INDOCRYPT, vol. 7668 of Lecture Notes in Computer Science, Springer, 2012, pp. 454–473.

[8] D. J. BERNSTEIN, N. DUIF, T. LANGE, P. SCHWABE, AND B.-Y. YANG, *High-speed high-security signatures*, Journal of Cryptographic Engineering, 2 (2012), pp. 77–89.

[9] D. J. BERNSTEIN AND T. LANGE, *Explicit-formulas database*. http://www.hyperelliptic.org/EFD/index.html, 2007.

[10] D. J. BERNSTEIN, T. LANGE, AND R. R. FARASHAHI, *Binary Edwards curves*, in CHES, vol. 5154 of Lecture Notes in Computer Science, Springer, 2008, pp. 244–265.

[11] I. F. BLAKE, G. SEROUSSI, AND N. SMART, *Elliptic Curves in Cryptography*, Cambridge University Press, 1999.

[12] E. BRIER, M. JOYE, AND T. E. D. WIN, *Weierstraß elliptic curves and side-channel attacks*, in PKC, vol. 2274 of Lecture Notes in Computer Science, Springer, 2002, pp. 335–345.

[13] W. S. BROWN, *The subresultant PRS algorithm*, ACM transactions on mathematical software, 4 (1978), pp. 237–24.

[14] J. H. CHEON AND J. H. YI, *Fast batch verification of multiple signatures*, in PKC, vol. 4450 of Lecture Notes in Computer Science, Springer, 2007, pp. 442–457.

[15] H. COHEN AND K. BELABAS, *PARI/GP*. http://pari.math.u-bordeaux.fr/, 2003–2013.

[16] H. COHEN, G. FREY, R. AVANZI, C. DOCHE, T. LANGE, K. NGUYEN, AND F. VERCAUTEREN, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, CRC Press, 2006.

[17] G. E. COLLINS, *Subresultants and reduced polynomial remainder sequences*, Jl of ACM, 14 (1967), pp. 128–142.

[18] A. DAS, D. R. CHOUDHURY, D. BHATTACHARYA, S. RAJAVELU, R. SHOREY, AND T. THOMAS, *Authentication schemes for VANETs: A survey*, International Journal of Vehicle Information and Communication Systems, 3 (2013), pp. 1–27.

[19] W. DIFFIE AND M. E. HELLMAN, *New directions in cryptography*, IEEE Transactions on Information Theory, 22 (1976), pp. 644–654.

[20] T. ELGAMAL, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Transactions on Information Theory, 31 (1985), pp. 469–472.

[21] A. FIAT, *Batch RSA*, Journal of Cryptology, 10 (1997), pp. 75–88.

[22] W. FISCHER, C. GIRAUD, E. W. KNUDSEN, AND J.-P. SEIFERT, *Parallel scalar multiplication on general elliptic curves over F_p hedged against non-differential side-channel attacks.* http://eprint.iacr.org/2002/007, 2002.

[23] S. GOLDWASSER AND S. MICALI, *Probabilistic encryption & how to play mental poker keeping secret all partial information*, in STOC '82, ACM, 1982, pp. 365–377.

[24] D. HANKERSON, J. LÓPEZ, AND A. MENEZES, *Software implementation of elliptic curve cryptography over binary fields.*, in CHES, vol. 1965 of Lecture Notes in Computer Science, Springer, 2000, pp. 1–24.

[25] D. HANKERSON, A. MENEZES, AND S. VANSTONE, *Guide to Elliptic-Curve Cryptography*, Springer, 2004.

[26] L. HARN, *Batch verifying multiple DSA-type digital signatures*, Electronics Letters, 34 (1998), pp. 870–871.

[27] ——, *Batch verifying multiple RSA digital signatures*, Electronics Letters, 34 (1998), pp. 1219–1220.

[28] J. HOEVEN AND G. LECERF, *On the complexity of multivariate blockwise polynomial multiplication*, in ISSAC, ACM, 2012, pp. 211–218.

[29] M.-S. HWANG, I.-C. LIN, AND K.-F. HWANG, *Cryptanalysis of the batch verifying multiple RSA digital signatures*, Informatica, 11 (2000), pp. 15–19.

[30] IEEE, *IEEE 1363-2000: Standard specifications for public key cryptography*. http://http://grouper.ieee.org/groups/1363/P1363/, 2004.

[31] ISO/IEC, *Information technology – Security techniques – Cryptographic techniques based on elliptic curves – Part 1: General*. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=44734, 2008.

[32] ——, *Information technology – Security techniques – Cryptographic techniques based on elliptic curves – Part 5: Elliptic curve generation*. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=46541, 2009.

[33] D. JOHNSON, A. MENEZES, AND S. A. VANSTONE, *The elliptic curve digital signature algorithm (ECDSA)*, International Journal of Information Security, 1 (2001), pp. 36–63.

[34] M. JOYE, *Security analysis of RSA-type cryptosystems*, PhD thesis, UCL Crypto Group, Belgium, 1997.

[35] T. LANGE, *A note on López-Dahab coordinates*, IACR Cryptology ePrint Archive 2004/323, (2004).

[36] J. LÓPEZ AND R. DAHAB, *Fast multiplication on elliptic curves over GF*(2^m) *without precomputation*, in CHES, vol. 1717 of Lecture Notes in Computer Science, Springer, 1999, pp. 316–327.

[37] A. MENEZES, S. VANSTONE, AND T. OKAMOTO, *Reducing elliptic curve logarithms to logarithms in a finite field*, in STOC, ACM, 1991, pp. 80–89.

[38] A. J. MENEZES, P. C. VAN OORSCHOT, AND S. A. VANSTONE, *Some computational aspects of root finding in GF(*q^m*)*, in ISSAC, vol. 358 of Lecture Notes in Computer Science, Springer, 1989, pp. 259–270.

[39] M. MIGNOTTE, *Mathematics for Computer Algebra*, Springer, 1992.

[40] B. MÖLLER, *Improved techniques for fast exponentiation*, in ICISC, vol. 2587 of Lecture Notes in Computer Science, Springer, 2002, pp. 298–312.

[41] ——, *Fractional windows revisited: Improved signed-digit representations for efficient exponentiation*, in ICISC, vol. 3506 of Lecture Notes in Computer Science, Springer, 2004, pp. 137–153.

[42] P. L. MONTGOMERY, *Speeding up Pollard and elliptic curve methods of factorization*, in Mathematics of Computation, vol. 48, 1987, pp. 243–264.

[43] ——, *Evaluating recurrences of form* $X_{m+n} = f(X_m, X_n, X_{m-n})$ *via Lucas chains.*, Microsoft research article, (1992), pp. 1–19.

[44] F. MORAIN AND J. OLIVOS, *Speeding up the computations on an elliptic curve using addition-subtraction chains*, Theoretical Informatics and Applications, 24 (1990), pp. 531–543.

[45] D. NACCACHE, D. M'RAIHI, D. RAPHEALI, AND S. VAUDENAY, *Can DSA be improved: Complexity trade-offs with the digital signature standard*, in EUROCRYPT, vol. 950 of Lecture Notes in Computer Science, Spinger, 1994, pp. 85–94.

[46] NIST, *The digital signature standard*, Communications of the ACM, 35 (1992), pp. 36–40.

[47] ——, *Secure hash standard (FIPS PUB 180-1)*. http://techheap.packetizer.com/cryptography/hash/fip180-1.pdf, 1993.

[48] ——, *Recommended elliptic curves for federal government use*. http://csrc.nist.gov/encryption, 1999.

[49] ——, *FIPS Publication 186-2*. http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf, 2001.

[50] ——, *Secure hash standard (SHS)*. http://csrc.nist.gov/publications/drafts/fips180-3/draft_fips_180-3_June-08-2007.pdf, 2007.

[51] ——, *SP 800-52 Rev. 1*, NIST Special publication, (2013).

[52] ——, *Draft SHA-3 standard (FIPS 202)*. http://csrc.nist.gov/publications/drafts/fips180-3/draft_fips_180-3_June-08-2007.pdf, 2014.

[53] K. OKEYA, K. SCHMIDT-SAMOA, AND T. T. C. SPAHN, *Signed binary representations revisited*, in CRYPTO, vol. 3152 of Lecture Notes in Computer Science, Springer, 2004, pp. 123–139.

[54] V. Y. PAN, *Simple multivariate polynomial multiplication*, Journal of Symbolic Computation, 18 (1994), pp. 183–186.

[55] S. POHLIG AND M. HELLMAN, *An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance*, IEEE Transactions on Information Theory, 24 (1978), pp. 106–110.

[56] J. M. POLLARD, *A Monte Carlo method for factorization*, BIT Numerical Mathematics, 15 (1975), pp. 331–334.

[57] M. O. RABIN, *Digitalized signatures and public-key functions as intractable as factorization*, tech. rep., Cambridge, MA, USA, 1979.

[58] C. REBEIRO, S. S. ROY, AND D. MUKHOPADHYAY, *Pushing the limits of high-speed GF(2^m) elliptic curve scalar multiplication on FPGAs*, in CHES, vol. 7428 of Lecture Notes in Computer Science, Springer, 2012, pp. 494–511.

[59] G. REITWIESNER, *Binary arithmetic*, Advances in Computers, 1 (1962), pp. 231–308.

[60] R. RIVEST, A. SHAMIR, AND L. ADLEMAN, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM, 21 (1978), pp. 120–126.

[61] T. SATOH AND K. ARAKI, *Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves*, Commentarii Mathematici Universitatis Sancti Pauli, 47 (1998), pp. 81–92.

[62] C. P. SCHNORR, *Efficient identification and signatures for smart cards*, in CRYPTO, vol. 435 of Lecture Notes in Computer Science, Springer, 1989, pp. 239–252.

[63] ——, *Efficient identification and signatures for smart cards (Abstract)*, in EUROCRYPT, vol. 434 of Lecture Notes in Computer Science, Springer, 1989, pp. 688–689.

[64] ——, *Message recovery for signature schemes based on the discrete logarithm problem*, in EUROCRYPT, vol. 950 of Lecture Notes in Computer Science, Springer, 1995, pp. 182–193.

[65] I. A. SEMAEV, *Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p*, Mathematics of Computation, 67 (1998), pp. 353–356.

[66] ——, *The discrete logarithm problem on elliptic curves of trace one*, Journal of Cryptology, 12 (1999), pp. 193–196.

[67] ——, *Summation polynomials and the discrete logarithm problem on elliptic curves*. http://eprint.iacr.org/2004/031, 2004.

[68] D. SHANKS, *Class number, a theory of factorization and genera*, in Symposia in Pure Mathematics, vol. 20, 1971, pp. 415–440.

[69] ——, *Five number theoretic algorithms*, in Proceedings of the Second Manitoba Conference on Numerical Mathematics, 1973, pp. 51–70.

[70] J. A. SOLINAS, *Improved algorithms for arithmetic on anomalous binary curves*, tech. rep., Originally presented in CRYPTO, 1997.

[71] M. STAM, *On Montgomery-like representations for elliptic curves over GF*(2^k), in PKC, vol. 2567 of Lecture Notes in Computer Science, Springer, 2003, pp. 240–253.

[72] ——, *Speeding up subgroup cryptosystems*, PhD thesis, Technische Universiteit Eindhoven, 2003.

[73] P. SZCZECHOWIAK, *Security in Wireless Sensor Networks: Elliptic Curve Cryptography and Pairing-Based Cryptography Solutions*, LAP LAMBERT Academic Publishing, 2011.