May 20, 2011

Smart card implementation of a digital signature scheme for Twisted Edwards curves

Author: Niels Duif*

Supervisors: Prof. Dr. Tanja Lange[1] and Dr. Ir. Cees Jansen[2]

Department of Mathematics and Computer Science

Technische Universiteit Eindhoven

Master thesis
Student number: 0554878

[1] Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
[2] Compumatica secure networks BV, P.O. Box 201, 5400 AE Uden, Netherlands
* Sponsored by Compumatica secure networks BV

Abstract

This report presents a new digital signature scheme for Twisted Edwards curves. The scheme was implemented on a smart card, using the Java Card language. The signature scheme is efficient; both signing and verification are faster than ECDSA. The scheme is inversion-free and suitable for batch verification.

The Java Card implementation of the scheme is protected against side-channel attacks. The implementation contains many useful techniques that reduce the computation time. Java Card proves to be a worthless platform for high-speed cryptography. Despite the speedups, generating a signature takes more than 28 minutes for a private key of 254 bits.

Contents

1 Introduction

In digital communication there are many applications of digital signatures. A digital signature is mostly used to verify that the sender of a digital message is as claimed, and that the message is unaltered. Just one example is a digital signature on software produced by a trusted party. Banking software and internet browsers are often downloaded through http, which is not secured. Therefore, the software is digitally signed. A user can verify the digital signature to ensure he has downloaded the right software, and not a malware-infected variant.

This thesis takes digital signatures to the world of smart cards. A typical smart card is a thin plastic card of 54x85 mm with a chip. Figure 1.1 shows an example of a smart card. The chip is clearly visible on the left. The card can exchange signals through the contact pads on the chip. There are also contactless smart cards, which communicate through an antenna integrated in the card. Some smart cards have both a contact and a contactless interface.

A smart card can perform computations. Especially, it can compute digital signatures. However, a smart card typically has very limited memory, and its computations are not very fast. Therefore, efficient signature schemes are needed for smart cards.

This thesis describes such an efficient signature scheme in Chapter 3, and its smart card implementation in Chapter 5. The signature scheme is based on elliptic curves, which are introduced in Chapter 2. For efficiency, Twisted Edwards curves are used. Chapter 2 introduces these curves, and gives a theoretical background for signature schemes, elliptic curves, and secure smart card implementation.

The smart card used for the implementation is the Cosmo ID One Lite v 5.4. Chapter 4 describes this smart card, and the programming language used for development on this card: Java Card. Chapter 6 presents the performance of the implementation, and Chapter 7 contains the conclusions of this thesis.

Figure 1.1: This bank card is a smart card with a contact interface. By the end of 2011, bank cards in the Netherlands will no longer use a magnetic stripe; only the smart card will be used [4]. This helps to prevent skimming.

2 Theoretical background

This chapter provides background information on signature schemes, elliptic curves, and secure smart card implementation. It describes two widely used signature schemes: RSA and ECDSA. The security of these schemes is also discussed.

2.1 Attacks

An *attack* on a cryptographic protocol is a procedure that recovers secret information, or enables one to use the protocol in an unintended way. Examples of attacks are generating a digital signature on a message m without possessing the secret key, or reconstructing the secret key from a collection of signatures. The set of entities attacking the protocol is called the *adversary*. If an attack requires the adversary to interact with the protocol, the attack is called *active*. If the attack only involves publicly known primitives, and intercepted messages, the attack is called *passive*.

An obvious attack that works for almost any protocol, is guessing the secret key. Plain guessing is known as a *brute force attack*. Secure protocols are constructed in such a way that a brute force attack cannot be completed within any reasonable time. From here, only attacks that are more efficient than a brute force attack will be considered an attack. With this definition, an attack still need not be practical.

2.1.1 Passive attacks

For a passive attack all the public parameters of the cryptographic protocol are assumed to be known to the adversary. In particular, the adversary knows the public key. Also, it is assumed that the adversary knows all the algorithms that are used in the protocol. If no further information is required for an attack, the attack is called a *key only attack* (KOA). For signature schemes, the message and the corresponding signatures are usually public. An attack that uses a set of known message-signature pairs is called a *ciphertext only attack* (COA).

2.1.2 Active attacks

In an active attack, public parameters and exchanged messages are assumed to be known, just as in a passive attack. But now the adversary is also allowed to interact with the protocol. A possible scenario is that the adversary observes the communication between A and B. The adversary then later establishes communication with A, while pretending to be B. This type of attack is called a *replay attack* (RA). Digital signature schemes cannot prevent replay attacks; a signature will remain valid. In a communication protocol, replay attacks can usually be detected by including an increasing counter in each message.

Sometimes an active adversary can send messages, intercept messages, block messages, and change messages while communication is in progress. This type of attack is called a *man in the middle attack* (MITM). A use of a digital signatures is to detect such interference.

It may be possible for an adversary to obtain signatures on a set of chosen messages. If this leads to an attack, the attack is called a *chosen plaintext attack* (CPA).

It may also be possible to obtain a decryption for a set of chosen ciphertexts. An attack that exploits this is called a *chosen ciphertext attack* (CCA1). If each queried ciphertext may be constructed using the results of the previous decryptions, the attack is called an *adaptive chosen ciphertext attack* (CCA2). For signature schemes, CCA1 or CCA2 assumes the attacker can get signatures on any bit string, while a CPA only allows messages as an input.

2.1.3 Application to smart cards

For some of the attacks described above, there is a realistic scenario in which they would be possible for a smart card based system. A risk for these systems is that a smart card is easy to steal. If a smart card is compromised by the adversary then it may be sent as many queries as desired. For this reason, bank cards usually allow only three trials for a personal identification number (PIN).

An RA or MITM is possible by physically modifying the smart card, if the contact pads can be connected with wires to a device other than the regular card accepting device (CAD). From this device, communication to the regular CAD is possible by applying a new layer with contact pads on top of the old one. Of course, the wiring also allows a COA. If the smart card has a wireless communication interface, it is sufficient for the adversary to have an antenna close to the smart card, and one close to the CAD.

A smart card may be used as a signing device in a bigger system. The smart card may simply act as a signing oracle. In that case, a CPA, CCA1, or CCA2 may be possible, depending on the implementation.

2.2 Digital signature schemes

A digital signature scheme generates a signature S, given a message m and a private key d. The aim of a digital signature is usually to prove that a message m was indeed transmitted by the claimed sender, and to verify that the message has not been modified. This section lists some requirements that are often imposed on signature schemes. In Section 2.5 and Section 2.8 these requirements will be used to assess the security of two digital signature schemes.

2.2.1 Breakability

A digital signature scheme is called *breakable* (BK) if it is possible to obtain the secret key with an attack. For example, if it is possible to construct the private key from the public key, then the system is called BK-KOA, or breakable with a key only attack.

2.2.2 Forgeability

A digital signature scheme is called *universally forgeable* (UF) if an adversary can sign any message m on behalf of another user, say user A. If a scheme is prone to UF-CPA then the forged signature should be on a message different from the ones that were queried.

If an adversary can sign some message m on behalf of user A, then the digital signature scheme is called *existentially forgeable* (EF). In this case the adversary is free to choose m as long as it was not signed by user A. As an example of an existential forgery, suppose S is a valid signature on m, generated by A. If an adversary finds a message $m' \neq m$ that also has S as a valid signature, then m' is an existential forgery.

2.2.3 Malleability

A digital signature scheme is called *malleable* (ML) if a message-signature pair (m, S) can be changed to a different pair (m', S') that is valid. It is not required that m and S are both changed in this process. Note that UF implies ML, but the converse is not true.

2.2.4 Bindingness

A digital signature scheme is called *binding* if it is infeasible for a signer to find two different messages m_1, m_2 with the same signature S. Note that a binding scheme resists EF-CPA but the converse is not true.

2.2.5 Hidingness

A digital signature scheme is called *hiding* if it is infeasible to recover the message m from the signature S.

2.3 Hash functions

A *hash function* H is a function that maps any bit string to a fixed length output. This is denoted as H: $\{0,1\}^* \rightarrow \{0,1\}^{L_h}$, where L_h is the length of the hash function. A hash function should be infeasible to invert, and it should be infeasible to find two inputs that map to the same output. What these requirements mean exactly is discussed in the Sections 2.3.1, 2.3.2, 2.3.3, and 2.3.4. Section 2.3.5 discusses the security of the SHA-2 hash functions, based upon these requirements.

2.3.1 First-preimage resistance

Given a hash value $H(m) = e \in \{0,1\}^{L_h}$ it should be infeasible to find the message m. Since the message can have any length, there is usually an infinite number of messages with hash value e. If finding any m' with $H(m') = e$ is infeasible, the hash function is called *first-preimage resistant* or *one-way*. Infeasible for first-preimage resistance means that there is no faster method than a brute force search. On average, a brute force approach takes 2^{L_h} queries to the hash function to find a preimage of e. The first-preimage-resistance security is then L_h bits.

In practice, if the message m is shorter than L_h then a preimage may be found in significantly fewer queries than the expected 2^{L_h}. This is done by querying all possible messages in increasing length. This kind of attack can be prevented by using a padding function based on random numbers.

2.3.2 Second-preimage resistance

Given a message m, it should be infeasible to find a message $m' \neq m$ such that $H(m') = H(m)$. A hash function that satisfies this property is called *second-preimage resistant*. The difference with first-preimage resistance is that in this case the message m is given, and that it is required that $m' \neq m$. Usually second-preimage resistance implies first-preimage reistance, although one can construct artificial examples where this is not the case. The intuition is that if first-preimage resistance does not hold, then one can always find a message m' such that $H(m') = H(m)$. Since there are usually infinitely many possibilities for m', it is unlikely that $m' = m$, so in general this also breaks second-preimage resistance.

Infeasibility for second-preimage resistance also means that the fastest way of finding a second-preimage is a brute force search. On average this should take 2^{L_h} queries on H. The second-preimage-resistance is then L_h bits.

2.3.3 Collision resistance

For a hash function H it should be infeasible to find two messages $m_1 \neq m_2$ such that $H(m_1) = H(m_2)$. A hash function with this property is called *collision resistant*. Infeasibility in this case means that the fastest way of finding a collision is a brute force search. For a collision this takes an expected number of $\sqrt{\frac{\pi}{2}} 2^{\frac{L_h}{2}}$ queries on H, using a birthday attack. This complexity is motivated in [28]. The complexity is $O(2^{\frac{L_h}{2}})$ hash queries, which means that the collision resistance is $\frac{L_h}{2}$ bits.

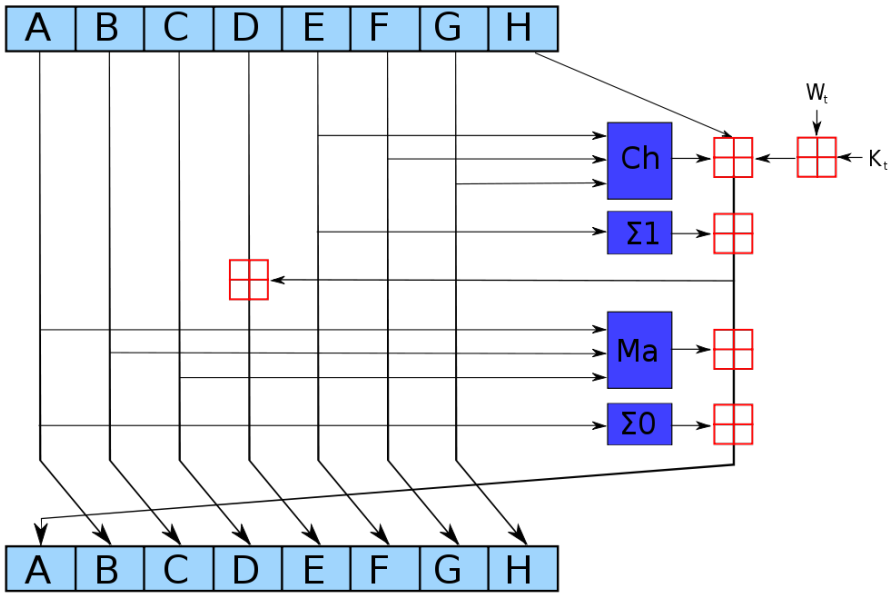


Figure 2.1: One iteration in a SHA-2 family compression function. The blue components perform the following operations: $\mathrm{Ch}(E,F,G) = (E \wedge F) \oplus (\neg E \wedge G)$, $\mathrm{Ma}(A,B,C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$, $\Sigma_0(A) = (A \gg 2) \oplus (A \gg 13) \oplus (A \gg 22)$, and $\Sigma_1(E) = (E \gg 6) \oplus (E \gg 11) \oplus (E \gg 25)$. The bitwise rotation uses different constants for SHA-512. The given numbers are for SHA-256. This figure was copied from Wikipedia.

If it is easy to find second-preimages for a hash function, then it is certainly not collision resistant. However, as long as the second-preimage-resistance of H is at least $\frac{L_h}{2}$ bits, H may still be collision resistant.

2.3.4 Zero resistance

If it is infeasible to find a message m such that $H(m) = 0$ then the hash function H is called *zero resistant*. Zero resistance is not implied by preimage resistance, since preimage resistance only means that it is generally infeasible to invert H, not for any specific value. Infeasibility means that finding a zero of H takes on average 2^{L_h} hash queries.

2.3.5 Security of the SHA-2 hash functions

The SHA-2 family of hash functions consists of the hash functions SHA-224, SHA-256, SHA-384, and SHA-512. SHA-224 is a truncated version of SHA-256, and SHA-384 is a truncated version of SHA-512. Also these hash functions use different initial vectors.

The SHA-2 hash functions all have the same basic structure. SHA-224 and SHA-256 operate on 8 blocks of 32-bit words. SHA-384 and SHA-512 operate on 8 blocks of 64-bit words. The internal operations are bitwise AND, OR, and XOR, bit rotations of words, and word additions modulo 2^{32} or 2^{64} respectively. All operations operate on 1-word blocks.

In iteration j, the blocks A_j, B_j, C_j, E_j, F_j, and G_j are simply copied into another position. This is good for performance, and if enough iterations are performed this will not affect security. Among the copied blocks are E_j, F_j, and G_j, which are the input to Ch and Σ_1. So D_j can be computed from E_{j+1}, F_{j+1}, G_{j+1}, and H_{j+1}. This partial inversion makes it possible to see whether some intermediate p_j-value at step j could produce another intermediate value p_{j+7}, 7 steps ahead. Only partial knowledge of p_j is needed for this. This method and other useful meth-

ods for partially breaking SHA-2 are discussed in [58].

A good hash function has no obvious structure, and therefore hash functions are hard to analyse. The following list states some criteria that a hash function must fulfill. If a hash function has any significant bias in one of these aspects, it can be exploited to find a collision or preimage.

- The output of H should be pseudo-random.
- A relation between two input messages m_1 and m_2 should not produce a relation between the outputs $H(m_1)$ and $H(m_2)$. For example, if m_1 and m_2 differ in one bit, $H(m_1)$ and $H(m_2)$ should, on average, differ in about half their bits.

The pseudo-randomness implies that the output should on average have as many zeros as ones, and should not have observable patterns.

Empirically these conditions are met quite well by the SHA-2 hash functions, although there are no proofs that guarantee any of these criteria. Currently there are no known attacks on SHA-2 that are more efficient than a brute force search. Guo and Matusiewicz have found a preimage attack for 42 out of 64 rounds of SHA-256 with a complexity equivalent to $2^{251.7}$ hash queries [13]. Sasaki et al have found a preimage attack for 46 out of 80 rounds of SHA-512 with a complexity equivalent to $2^{511.5}$ hash queries [58]. Both attacks are far too complex to be practical, and they do not apply to the full 64-round SHA-256, nor to the full 80-round SHA-512. Collisions were found by Sanadhya and Sarkar for 24 rounds of SHA-256 with a complexity of $2^{28.5}$ hash queries, and for 24 rounds of SHA-512 with a complexity of $2^{32.5}$ hash queries.

A hash function is considered broken as soon as any attack on the full-round hash function has a complexity lower than a brute force attack. Usually this does not imply that such an attack is practical. The SHA-2 hash functions are nowhere near broken, but the breaks for 42 out of 64 rounds and 46 out of 80 rounds have made some people feel uneasy. New attacks often improve an old attack by combining known methods with new ones. In this way, attacks for reduced-round versions of MD5, SHA-0, and SHA-1 have been extended to attacks for the full round versions, see [67], [68], [47]. At the moment this report is written, NIST is running a competition for SHA-3, the successor of SHA-2.

2.3.6 Random oracle model

In security proofs hash functions are often modeled as a *random oracle*.

Definition 2.3.1 (Random oracle)**.** *A* random oracle *is a hash function RO:* $\{0,1\}^* \to \{0,1\}^{L_h}$ *with the following properties.*

- *If* m *was never queried before,* $RO(m)$ *is chosen at random in* $\{0,1\}^{L_h}$*.*
- *If* m *was queried before,* $RO(m)$ *is equal to the output when* m *was last queried.*

The random oracle RO meets all the requirements in Section 2.3, and satisfies all properties of a hash function, except that it is not deterministic. It is obvious that a deterministic hash function cannot actually behave as a random oracle. A model that assumes only preimage resistance, collision resistance, and zero resistance of a hash function is called the *standard model*.

2.4 RSA

RSA is a system for public key encryption and digital signing, invented in 1977 by Rivest, Shamir, and Adleman [55]. It is defined as follows:

Definition 2.4.1 (RSA)**.** RSA *is a family of trapdoor permutations defined by*

- *The bit length k, which determines the security.*
- *The modulus N, which is a k-bit number that is the product of two randomly generated $\frac{k}{2}$-bit prime numbers p and q.*
- *The public exponent $e \leq \phi(N)$, which is randomly generated until* $\gcd(e, \phi(N)) = 1$.
- *The private exponent d defined by* $de \equiv 1 \pmod{\phi(N)}$.
- *The encryption function* f*:* $x \to x^e \pmod{N}$.
- *The decryption function* f^{-1}*:* $y \to y^d \pmod{N}$.

In Defnition 2.4.1 ϕ is Euler's totient function, so $\phi(N) = (p-1)(q-1)$. Note that f^{-1} is indeed the inverse of f since

$$f^{-1}(f(x)) = (x^e)^d \pmod{N} \equiv x^{(ed \pmod{\phi(N)})} \pmod{N} \equiv x^1 \pmod{N} = x, \tag{2.1}$$

and

$$f(f^{-1}(x)) = (x^d)^e \pmod{N} = x, \tag{2.2}$$

where the last equality in Equation 2.2 follows from Equation 2.1. For encryption f is used together with an invertible padding scheme μ. Encryption is explained in Section 2.4.1. Decryption is explained in Section 2.4.2. Signing and signature verification are explained in Sections 2.4.3 and 2.4.4.

The requirement that e should be randomly generated is usually relaxed to make encryption and signature verification less computationally intensive. Common choices are $e = 3$, $e = 17$, and $e = 2^{16} + 1$. Security restrictions on the parameters are discussed in the security analysis of RSA in Section 2.5.

The values (k, N, e) are made public, while d must be kept secret. The values $(p, q, d \pmod{q}, d \pmod{p})$ may be stored and must be kept secret. $\phi(N)$ and other unused parameters, if any, must be deleted to minimise the risk of losing sensitive information.

2.4.1 RSA encryption

To encrypt a message m the sender computes $f(\mu(m))$. The padding scheme μ should be such that $\mu(m)$ does not exhibit any algebraic structure the message m might have. The message m has length at most $k - k_{\text{pad}}$, while $\mu(m)$ always has length k. k_{pad} is the extra length required for the padding scheme. A secure padding scheme μ is often probabilistic in nature, and is always invertible.

2.4.2 RSA decryption

To decrypt a ciphertext $c = f(\mu(m))$ the receiver computes $\mu(m) = f^{-1}(c)$. The receiver then computes $m = \mu^{-1}(\mu(m))$ to recover the message m.

2.4.3 RSA signing

To sign a message m the signer computes $s = f^{-1}(\mu(H(m)))$, where H is a hash function that is publicly known before signing. The hash function is used as a function that is not invertible and collision-resistant. This ensures that one cannot find two different messages m_1 and m_2 with the same signature, or use a signature on m to compute a signature on $m' \neq m$. The padding function μ may be different from the one used in encryption. The signer then sends the pair (m, s) to the verifier.

2.4.4 RSA signature verification

To verify a signature (m, s) the verifier computes $H(m)$ from m, and then checks whether $H(m) \stackrel{?}{=} \mu^{-1}(f(s))$. If so, he accepts the signature. Otherwise he rejects the signature.

2.5 RSA security analysis

The RSA cryptosystem and its parameters are described in Section 2.4. In this section the security of RSA is analysed both when used for encryption, and for signing.

Some weak parameters for RSA are known, and there are some insecure ways in which it can be used. Classic examples include the following:

- Use no message padding, and send the same message m to at least 3 different recipients who all use $e = 3$.
- Encrypt and sign messages with the same key, where the encryption and signing are implemented as inverse operations. For example when no padding is used.
- Act as an RSA oracle and sign any bit string with the secret key.

To prevent these and other insecure settings, the following is assumed about the RSA implementation:

- $e > 3$. Usually $e = 2^{16} + 1$ is used, and for very secure implementations e is randomly generated.
- A sufficiently secure padding scheme is used. For example, OAEP+ is secure against adaptive chosen ciphertext attacks in the random oracle model (Section 2.3.6) [61].

For OAEP+ to be secure a secure random number generator is needed. Using a pseudo-random number generator with the system time as a random seed is usually not sufficiently secure. Also note that $e = 3$ may be perfectly secure if the padding scheme is secure [61], but this exponent is usually avoided nonetheless [3]. When p and q are close, an efficient factoring method for N exists, so [3] requires $\log|p - q| \geq \frac{\log N}{2} - 100$.

2.5.1 RSA for encryption

The parameters N and e are publicly known. To break the system, one of the following is sufficient:

1. Given any ciphertext $c = m^e$, compute m.
2. Compute d.
3. Compute $\phi(N)$.
4. Compute p and q.

1 is known as the RSA problem, and is equivalent to breaking RSA. Note that 2, 3, and 4 are equivalent to each other and imply 1 [55]. Currently, factoring N into p and q is the most feasible way of solving the RSA problem. It is not known whether factoring an integer can be done in polynomial time, and the problem is not known to be NP-complete either.

The fastest known algorithm for factoring a number that is not of a special form is the General Number Field Sieve (GNFS). This algorithm runs in sub-exponential time. Its complexity is

$$L_n[\frac{1}{3}, c] = \exp\left((c + o(1))\,(\log N)^{\frac{1}{3}}\,(\log\log N)^{\frac{2}{3}}\right), \tag{2.3}$$

where $c = \left(\frac{64}{9}\right)^{\frac{1}{3}}$. The GNFS consists of a sieving step, a matrix step, and some pre- and post-processing. The sieving step is the most computationally intensive. It finds relations $a^2 \equiv b \pmod{N}$, where b is a number that only has small prime factors. For $N \approx 2^{768}$ "small" means that all prime factors are smaller than 10^8, except for four, which must be smaller than 2^{40}.

The sieving step can easily be distributed over multiple computers. Efficient open source GNFS software exists. An example is kmGNFS, which is available both as a single machine version, and as a distributed version [8]. Another CPU intensive step of the GNFS is the matrix step, which consists of solving a large sparse linear system of equations. This step can be parallelised, for example by the Block Wiedemann algorithm, but this is more involved than parallellising the sieving step. In the kmGNFS project the matrix step does not run in parallel, and therefore takes a lot of time. When Kleinjung et al completed the factorisation of a 768-bit RSA modulus in December 2009 the time ratio between the sieving step and the matrix step was approximately 10 to 1 [42], while a large cluster was used for the matrix step. In total, the computation took about 2^{67} operations, requiring over 2 years. Note that this is only $\frac{1}{730}$ of the value L_n from Equation 2.3.

In practice, the cheapest way of factoring an RSA modulus would be buying a large botnet for the computation. Getting GNFS software to run smoothly with maximum parallellisation on the botnet is a challenging but feasible task. Assuming a budget of 1 million euro, an attacker could buy up to 4 million hijacked PCs [1]. If the attacker is prepared to spend at most one year on the computation, this yields about 2^{77} operations, assuming the PCs have a 1 GHz processor and sufficient memory. With an estimate of 2^{67} operations for RSA-768 and Equation 2.3, this means that it should be possible to factor a 1010-bit RSA modulus with the best algorithms available. Of course, these estimates will not be entirely accurate, but it is strongly advised not to use RSA-1024 for any application that is worth more than a million euros. Note that botnets are illegal, and that a botnet of 4 million PCs is likely to be detected. If someone figures out your botnet is factoring an RSA modulus then the modulus may be published, which means that the users of that modulus will abandon it.

In special cases N can be factored quite easily. Therefore, there are some restrictions on p and q. Let p_1 and q_1 be the largest prime factors of $p-1$ and $q-1$ respectively. Then p_1 and q_1 must both be large. Otherwise one of them can be found with Pollard's $p-1$-algorithm [54]. This algorithm takes

$$O(p_1 \log p_1 (\log N)^2) \tag{2.4}$$

time, so it is faster than the GNFS approximately when $\log p_1 < (\frac{64}{9})^{\frac{1}{3}} (\log N)^{\frac{1}{3}} (\log \log N)^{\frac{2}{3}}$. So for $\log_2 N = 2048$, p_1 and q_1 should at least have $\log_2 p = 117$ bits. To be safe, some extra bits are recommended. NIST requires a minimum length of 140 bits for $\log_2 N = 2048$ [3], which seems quite reasonable. It also specifies upper bounds for p_1 and q_1, while there is no apparent reason for this. Maybe these are specified to limit storage.

Factoring N is also feasible when $p+1$ or $q+1$ has only small factors. This is done by Williams' $p+1$ algorithm [71]. It is somewhat slower than Pollard's $p-1$ factorisation algorithm, so for $p+1$ and $q+1$ the same requirements should hold as for $p-1$ and $q-1$.

The $p-1$ and $p+1$ idea can be generalised to any cyclotomic polynomial value $\Phi_k(p)$. However, $\Phi_1(p) = p-1$ and $\Phi_2(p) = p+1$ are the only cyclotomic polynomials in p of degree 1, which means that the probability of finding a value $\Phi_k(p)$ with only small prime factors for $k > 2$ is very small if p is randomly generated.

Also, p and q must both be sufficiently large. The complexity of some factoring algorithms is measured in terms of the smallest factor of N, so if either p or q is too small, factoring N may be feasible. An example is the Lenstra elliptic curve factorisation method (ECM) [45]. Its time complexity is

$$L_n[\frac{1}{2}, 1]M(N) = \exp\left((\sqrt{2} + o(1))\,(\log p \log\log p)^{\frac{1}{2}}\right) M(N), \tag{2.5}$$

where p is the smallest prime factor of N, and $M(N)$ is the complexity of performing a multiplication $\pmod N$. $M(N)$ is at most $(\log N)^2$, and for large N it converges to $O(\log N \log\log N \log\log\log N)$ or even less [31]. However, for a realistic 2048-bit RSA modulus the fastest multiplication algorithm is Karatsuba's method [72], which takes approximately $O((\log N)^{1.585})$ time [39]. The complexity of reducing modulo n is ignored, since it is not more than the complexity of multiplication. This can be seen from the Montgomery multiplication algorithm [51]. It follows from Equations 2.3 and 2.5 that the ECM is faster than any general purpose factorisation method approximately when $\sqrt{2\log p \log\log p}(\log N)^{1.465} < (\frac{64}{9})^{\frac{1}{3}}(\log N)^{\frac{1}{3}}(\log\log N)^{\frac{2}{3}}$. For $\log_2 N = 2048$ bits this means that p should at least have $\log_2 p = 594$ bits. To be safe, it is recommended to use a few bits more, since it is not wise to have two different attacks that are equally feasible. NIST is very conservative in this, and requires p and q to have equal bit length [3]. From a practical viewpoint it is sensible to take p and q of equal bit length, since this is easiest to implement, requires the least time in prime number generation and verification, and is an optimal choice for the performance of decryption by the Chinese Remainder Theorem.

When d is small, or when $d \pmod p$ and $d \pmod q$ are small, then it is feasible to factor N [38].

The probability that any of these attacks is possible, is very small if p and q, are randomly generated, and d is not chosen to be small.

2.5.2 RSA for signing

To break the RSA signature scheme, it is sufficient to factor the RSA modulus. However, there are other methods of breaking it. A collision of the hash function H makes it possible to generate two different messages that belong to the same signature. A second preimage attack allows to generate for a given message m, a different message $m^* \neq m$ that has the same signature (r, s). So the hash function H needs to resist these techniques for the signature scheme to be binding.

Usually the hash function has a shorter length than the RSA message length. In some versions of RSA this can be exploited, for example in the original versions of ISO/IEC 9796-1 and ISO/IEC 9796-2 [27]. The essence of these attacks is that a large part of the to be signed bit string is known to an attacker who launches a chosen message attack. In a scheme like RSA-OAEP the signer uses a (true) random number generator and a hash function to ensure that every bit of the to be signed bit string is pseudo-randomly affected.

A different way of breaking the RSA signing algorithm is finding d from m and s, where $s = m^d \pmod N$. This is known as the discrete logarithm problem, and it is in general harder than factoring a number of the same length. In this case the discrete logarithm problem is equivalent to factoring the modulus, since 2 and 4 from Section 2.5.1 are equivalent.

In the context of signatures, the RSA problem ((1 from Section 2.5.1)) is to be understood as: given $\mu(H(m))$, compute s satisfying $s^e = \mu(H(m))$. So breaking the RSA problem is equivalent to finding a signature s for a message m.

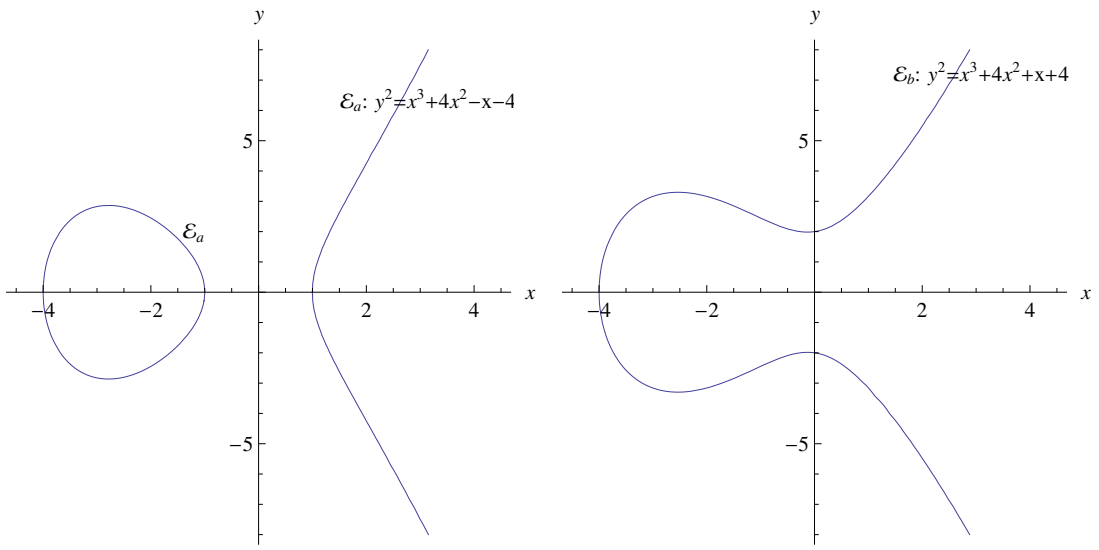
In conclusion, breaking the RSA signing algorithm requires at least one of the following:

- Breaking the hash function H by finding a collision or worse.
- Breaking the RSA problem.

2.6 ECC

Elliptic curve cryptography (ECC) is a cryptographic primitive that is used for public key cryptography. It can be used in digital signature schemes, and in key agreement protocols. A common digital signature scheme based upon ECC is ECDSA, which is discussed in Section 2.7.

ECC is based upon an *elliptic curve* and an *addition law*. An elliptic curve can be defined by its Weierstrass form:



(a) $\Delta(\mathcal{E}_a) > 0$ (b) $\Delta(\mathcal{E}_b) < 0$

Figure 2.2: Two elliptic curves in $\mathbb{R}^2$. For $\Delta(\mathcal{E}_a) > 0$, the curve has two affine parts, and for $\Delta(\mathcal{E}_b) < 0$, $\mathcal{E}_b$ has one affine part.

Definition 2.6.1 (Weierstrass curve)**.** *Let $\mathbb{F}$ be a field, and $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}$. A* Weierstrass curve *$\mathcal{W}$ is the set of points $P = (x, y) \in \mathbb{F}^2$ defined by*

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{2.6}$$

together with a point at infinity *P_∞.*

A pair $P = (x, y)$ that satisfies Equation 2.6 is called a *point* on the curve $\mathcal{W}$. Figure 2.2 shows an example of two Weierstrass curves in $\mathbb{R}^2$. Both curves have no cusps or self-intersections, which means they are non-singular. This implies that both these Weierstrass curves are elliptic curves. Algebraically, a Weierstrass curve $\mathcal{W}$ is non-singular if the discriminant $\Delta(\mathcal{W})$ is non-zero. $\Delta(\mathcal{W})$ is defined by

$$\begin{aligned} b_2 &= a_1^2 + 4a_2 \\ b_4 &= a_1a_3 + 2a_4 \\ b_6 &= a_3^2 + 4a_6 \\ b_8 &= a_1^2a_6 - a_1a_3a_4 + a_2a_3^2 + 4a_2a_6 - a_4^2 \\ \Delta &= -b_2^2b_8 + 9b_2b_4b_6 - 8b_4^3 - 27b_6^2. \end{aligned} \tag{2.7}$$

Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be curves. If there exists a rational map ϕ: $\mathcal{C}_1 \to \mathcal{C}_2$ with a rational inverse $\psi = \phi^{-1}$, then $\mathcal{C}_1$ and $\mathcal{C}_2$ are called *birationally equivalent*. An additional requirement is that this equivalence preserves the group structure of an elliptic curve. A group structure on elliptic curves is defined later in this section. Now an elliptic curve is defined as follows.

Definition 2.6.2 (Elliptic curve)**.** *Let $\mathbb{F}$ be a field, and $\mathcal{E}$ an algebraic curve over $\mathbb{F}$. Then $\mathcal{E}$ is an* elliptic curve *if and only if there exists a non-singular Weierstrass curve $\mathcal{W}$ that is birationally equivalent to $\mathcal{E}$.*

Elliptic curves are described more naturally in the projective plane. Let $\mathcal{E}_1$ be defined by Equation 2.6. Then there is a birational equivalence to the curve $\mathcal{E}_2$ in $\mathbb{P}^2(\mathbb{F})$ defined by $x = \frac{X}{Z}$, and $y = \frac{Y}{Z}$. The point P_∞ is mapped to $(0 : 1 : 0)$. The curve equation of $\mathcal{E}_2$ then becomes:

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3. \tag{2.8}$$

If the field $\mathbb{F}$ over which the curve $\mathcal{E}$ is defined does not have characteristic 2 (i.e. $2 \not\equiv 0$), then Equation 2.6 can be simplified with the substitution $x' = x, y' = y + (a_1x + a_3)/2$ to give the equation

$$y'^2 = x'^3 + \frac{b_2}{4}x'^2 + \frac{b_4}{2}x' + \frac{b_6}{4}, \tag{2.9}$$

where b_2, b_4, and b_6 are as in Equation 2.7.

To keep the definitions and constructions compact, it is assumed that all elliptic curves are defined over a field $\mathbb{F}$ with $\text{char}(\mathbb{F}) \neq 2$. For these definitions and constructions there are often binary equivalents that hold for $\text{char}(\mathbb{F}) = 2$.

If also $\text{char}(\mathbb{F}) \neq 3$ then substituting $x = x' - a_2/3, y = y'$ into Equation 2.9 gives

$$y^2 = x^3 + a_4''x + a_6'' \tag{2.10}$$

which is called a *short Weierstrass* equation. If $\text{char}(\mathbb{F}) \neq 2, 3$ and $a_2^2 - 3a_4$ is a square, then Equation 2.9 may instead be rewritten as

$$y^2 = x^3 + a_2'''x^2 + a_6''', \tag{2.11}$$

using the substitution $x = x' - (a_2 + \sqrt{a_2^2 - 3a_4})/3, y = y'$. Equation 2.11 is called a *Montgomery* form.

To find points on an elliptic curve, suppose that P and Q are given points on the elliptic curve $\mathcal{E}$. Then a third point on $\mathcal{E}$ may be found by intersecting the line through P and Q with $\mathcal{E}$. This operation will be denoted by "$*$", and is depicted in Figure 2.3.

Definition 2.6.3. *Let $\mathcal{E}$ be an elliptic curve in Weierstrass form, and P and Q points on $\mathcal{E}$. Then "$*$" is defined by*

$$P * Q = R \tag{2.12}$$

where R is the third point of intersection of $\mathcal{E}$ and the line l through P and Q.

For this definition to make sense $\mathcal{E}$ should be viewed projectively, and multiplicities should be taken into account. For example, when the line l through P and Q is tangential to $\mathcal{E}$ at Q, then Q has multiplicity 2 on l, so $R = Q$. On the other hand, when $P = Q$ then the line through P and Q is the tangent to $\mathcal{E}$ at P. When l intersects $\mathcal{E}$ in only two affine points (counting multiplicities), then the third point of intersection is the point at infinity P_∞. For a curve in Weierstrass form this occurs when the line l is vertical.

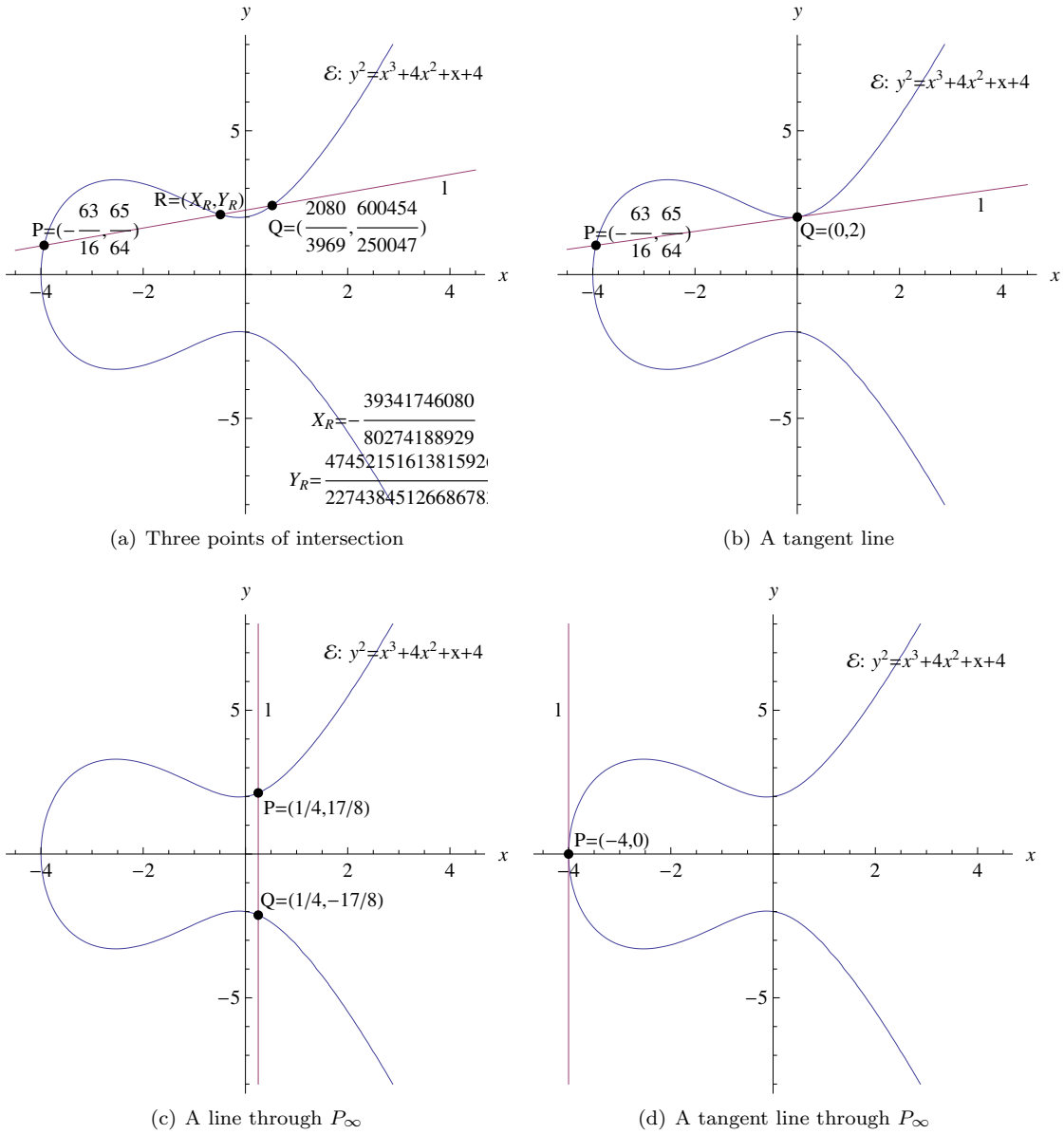
That a line through two points on an an elliptic curve always intersects it in exactly one more point, will be shown in the proof of Theorem 2.6.5.
On an elliptic curve a *point addition* "$+$" can be defined. It allows *adding* two points P and Q on an elliptic curve $\mathcal{E}$. It is proved in Theorem 2.6.5 that this defines an Abelian group $(\mathcal{E}, +)$.

Definition 2.6.4 (Point addition)**.** *Let $\mathcal{E}$ be an elliptic curve over $\mathbb{F}$, and take $\mathcal{O}$ to be a point on $\mathcal{E}$ that satisfies $\mathcal{O} * \mathcal{O} = \mathcal{O}$. Also let P and Q be points on $\mathcal{E}$. Then* point addition *"$+$" on $\mathcal{E}$ is defined by*

$$P + Q = (P * Q) * \mathcal{O},$$

where "$*$" is as in Definition 2.6.3. Also define
$-P = P * \mathcal{O}$.

A common choice is $\mathcal{O} = P_\infty$. The elliptic curve is then represented by Equation 2.9, and the lines that intersect P_∞ are vertical lines. This means that for a point $P = (x, y)$, $-P$ is given by $(x, -y)$. It is now shown that the operation "$+$" from Definition 2.6.4 turns $\mathcal{E}$ into an Abelian group.



(a) Three points of intersection

(b) A tangent line

(c) A line through P_∞

(d) A tangent line through P_∞

Figure 2.3: A line l through two points on an elliptic curve $\mathcal{E}$ has a third point of intersection with $\mathcal{E}$. Here $\mathcal{E}$ is defined over $\mathbb{R}$. In (a) $P * Q = R$. In (b) $P * Q = Q$, since l is a tangent at Q. In (c) l is vertical so $P * Q = P_\infty$. In (d) $P * P = P_\infty$, since the tangent line l at P is vertical.

Theorem 2.6.5 (Group law)**.** *Let $\mathcal{E}$ be an elliptic curve in Weierstrass form over $\mathbb{F}$,* $\text{char}(\mathbb{F}) \neq 2$*, and P, Q, T points on $\mathcal{E}$. Furthermore, let "+" be the* point addition *defined in Definition 2.6.4. Then $(\mathcal{E}, +)$ forms an additive Abelian group with $\mathcal{O} = P_\infty$ as the neutral element. In other words:*

1. *If $P + Q = S$ then S is on $\mathcal{E}$*
2. $P + Q = Q + P$
3. $(P + Q) + T = P + (Q + T)$
4. $P + \mathcal{O} = P$
5. $P + (-P) = \mathcal{O}$

Proof. $\text{char}(\mathbb{F}) \neq 2$ so $\mathcal{E}$ is given by

$$y^2 = x^3 + a_2x^2 + a_4x + a_6. \tag{2.13}$$

It is first shown that Definition 2.6.3 is well-defined. The line l through $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ has projective equation $aX + bY + cZ = 0$ for some $a, b, c \in \mathbb{F}$. If $b = 0$ then this is a vertical line, and the third point of intersection is $R = (0 : 1 : 0) = P_\infty$. Otherwise, l can be dehomogenised to give $y = dx + e$ for some $d, e \in \mathbb{F}$. Substituting $dx + e$ for y in Equation 2.13 gives a cubic equation in x, which has 3 roots, counting multiplicities. Since x_P and x_Q are in $\mathbb{F}$, and are roots of this equation, the third root of this equation must also be in $\mathbb{F}$. So we find a solution $x_S \in \mathbb{F}$ that must be the x-coordinate of the third point of intersection of l and $\mathcal{E}$.

1. Follows from the construction above, and applying Definition 2.6.4.
2. This property follows directly from Definition 2.6.4 and Definition 2.6.3.
3. The associativity of the addition law is by far the hardest to prove. For an algebraic proof, the reader is referred to [30]. A proof that requires some knowledge of algebraic geometry can be found in [25].
4. Let $S = P + \mathcal{O} = (P * \mathcal{O}) * \mathcal{O}$, then S is found by first intersecting the line l through P and $\mathcal{O}$ with $\mathcal{E}$, and taking the third point of intersection $P * \mathcal{O} = R$. Next, the line through R and $\mathcal{O}$ is taken, and S is the third point of intersection with $\mathcal{E}$. The line through R and $\mathcal{O}$ must be l, so $S = P$.
5. Let $S = P + (-P) = (P * (P * \mathcal{O})) * \mathcal{O}$, then S is found by first intersecting the line l through P and $\mathcal{O}$ with $\mathcal{E}$, and taking the third point of intersection $P * \mathcal{O} = R$. Then the line through R and P is again l, so $P * R = \mathcal{O}$. Finally, S is found as $\mathcal{O} * \mathcal{O} = \mathcal{O}$. We now show that this property holds for $\mathcal{O} = P_\infty$ if $\mathcal{E}$ satisfies Equation 2.9. $\mathcal{O} * \mathcal{O}$ is defined as the third point of intersection of $\mathcal{E}$ and the tangent l at $\mathcal{O}$. For $\mathcal{O} = P_\infty = (0 : 1 : 0)$ the tangent l satisfies $aX + bY + cZ = 0$, where $a = \frac{\partial F}{\partial x}(P_\infty)$, $b = \frac{\partial F}{\partial y}(P_\infty)$, and $c = \frac{\partial F}{\partial z}(P_\infty)$. For F the homogeneous equivalent of Equation 2.9 is used:

$$F(X, Y, Z) = -Y^2Z + X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{2.14}$$

to obtain $a = 0$, $b = 0$, $c = 1$. Intersecting l: $Z = 0$ with $\mathcal{E}$: $F = 0$ gives $Z = 0$. This implies $X^3 = 0$, so $X = 0$. This means that $(0 : 1 : 0)$ is the only solution to this equation, with multiplicity 3 as required. So indeed $P_\infty * P_\infty = P_\infty$. □

Now that adding points defines an Abelian group structure, it makes sense to define a *point-scalar multiplication*.

Definition 2.6.6 (Point-scalar multiplication)**.** *Let $\mathcal{E}$ be an elliptic curve, and let "+" be the point addition from Definition 2.6.4. Also, let P be a point on $\mathcal{E}$. Then for $k \in \mathbb{Z}$ define*

$$kP = \begin{cases} \underbrace{P + P + \cdots + P + P}_{k \text{ times}} & \text{if } k \geq 0 \\ |k|(-P) & \text{if } k < 0 \end{cases} \tag{2.15}$$

As usual, the empty addition $0P$ is the neutral element $\mathcal{O}$.

Efficient algorithms exist for point-scalar multiplication. Section 2.9 discusses some possible algorithms for Edwards curves. For ECC the elliptic curve is defined over a finite field $\mathbb{F}_q$. This implies that the number of points on $\mathcal{E}$ is finite. For a point P on $\mathcal{E}$ the *order* of that point is the smallest integer n such that $nP = \mathcal{O}$. Since the number of points on an elliptic curve $\mathcal{E}$ over a finite field $\mathbb{F}_q$ is finite, the number n is also finite. ECC requires a point G on $\mathcal{E}$ that generates a subgroup of large prime order. The order n of this point divides the number of points on $\mathcal{E}$. By Hasse's theorem [64] the number of points on an elliptic curve N_p over a finite field $\mathbb{F}_p$ is bounded by

$$|N_p - (p+1)| \leq 2\sqrt{p}. \tag{2.16}$$

So by Equation 2.16, n cannot be much larger than $|\mathbb{F}_q|$. In practice, elliptic curves are chosen in such a way that n has (almost) the same bit length as q.

Let G be a point of order n, and $k \in \{1, \ldots, n-1\}$. While it is easy to find kG from k, the opposite is not true. ECC relies on the assumption that this operation is infeasible for large n.

Assumption 2.6.1 (Elliptic curve discrete logarithm assumption (ECDLA))**.** *Let $\mathcal{E}$ be an elliptic curve, and let G be a publicly given point on $\mathcal{E}$ of large prime order n. Let $Q = kG$ be given for a randomly selected $k \in \{1, \ldots, n-1\}$. Then it is computationally infeasible to find k from Q.*

It is generally believed that the complexity of this problem is proportional to $\sqrt{n}$, which means it is exponential in $\log_2(n)$, the bit length of n. ECDLA is discussed in Section 2.8.

ECC can be used for a Diffie-Hellman or MQV key agreement, and for a digital signature system. The signature scheme ECDSA is described in Section 2.7. The other schemes are not discussed in this report.

2.7 ECDSA

The elliptic curve digital signature algorithm (ECDSA) is a digital signature algorithm based on ECC. ECC is described in Section 2.6. ECDSA is defined as follows [9], [3].

Definition 2.7.1 (ECDSA)**.** *Let $\mathbb{F}_q$ be a finite field, and $\mathcal{E}$ an elliptic curve defined over $\mathbb{F}_q$. Let G be a point on $\mathcal{E}$ of large prime order n, and let the private key d be in $\{1, \ldots, n-1\}$. The public key is $Q = dG$. Let H be a hash function with output length $L_h \geq L_n = \lfloor \log_2 n \rfloor + 1$, the bit length of n. Then a signature (r, s) on the message m is computed as follows.*

1. *Take z as the L_n most significant bits of $H(m)$.*
2. *Select k at random in the interval $\{1, \ldots, n-1\}$.*
3. *Compute $(x_1, y_1) = kG$, and $r = x_1 \pmod{n}$. If $r = 0$, go to Step 2.*
4. *Compute $s = k^{-1}(z + rd) \pmod{n}$. If $s = 0$ go to Step 2.*
5. *The signature is (r, s).*

A signature (r, s) is verified as follows:

1. *Verify Q is on $\mathcal{E}$, $Q \neq \mathcal{O}$, $nQ = \mathcal{O}$, and r, s are in $\{1, \ldots, n-1\}$.*
2. *Compute z as the L_n leftmost bits of $H(m)$.*
3. *Compute $w = s^{-1} \pmod{n}$.*
4. *Compute $u_1 = zw \pmod{n}$, and $u_2 = rw \pmod{n}$.*
5. *Compute $(x_2, y_2) = u_1 G + u_2 Q$.*
6. *Verify that $r = x_2 \pmod{n}$.*

In short, the signing is described by:

$$r = (kG)_x \pmod{n} \tag{2.17}$$
$$s = k^{-1}(H(m) + rd) \pmod{n}, \tag{2.18}$$

where $(.)_x$ takes the x-coordinate of a point, and $H(m)$ is the truncated hash of m. The verification is described in short by:

$$r \stackrel{?}{=} (H(m)s^{-1}G + rs^{-1}Q)_x \pmod{n}, \tag{2.19}$$

where $(.)_x$ takes the x-coordinate of a point, and $H(m)$ is the truncated hash of m.

A correctly generated signature is always accepted. Suppose that (r, s) is a signature on m that was generated according to Definition 2.7.1. Then

$$u_1G + u_2Q = kz(z + rd)^{-1}G + kr(z + rd)^{-1}Q = k(z + rd)^{-1}(z + rd)G = kG, \tag{2.20}$$

so indeed $r = x_2$, and the verification succeeds. Further properties of this signature scheme are discussed in Section 2.8.

2.8 ECDSA security analysis

ECDSA is described in Section 2.7. This section analyses the security of ECDSA. Weak choices of parameters values are discussed, as well as the best known way of breaking the general case. First the properties of the algorithm are discussed. It is assumed that the hash function is at least collision resistant, and that the elliptic curve discrete logarithm problem is intractable. Whenever stronger assumptions are needed they are explicitly stated.

2.8.1 Pollard's methods

Any discrete logarithm based system can be attacked by Pollard's rho or Pollard's kangaroo method. These methods are described for multiplicative groups modulo a prime number in [53], but work for any cyclic group $\mathbf{G}$. When G is a generator of $\mathbf{G}$, and $Q_A \in \mathbf{G}$ then either method finds $k_A \in \mathbb{Z}_n$ such that $Q_A = k_AG$. Both algorithms are probabilistic. For a group of order n, the expected running time of the Pollard-rho algorithm is

$$\sqrt{\frac{\pi n}{2}}. \tag{2.21}$$

So the method is exponential in $\log_2 n$. Here an elementary operation is an addition when $\mathbf{G}$ is viewed additively. So for an elliptic curve this is a point addition.

Section 2.9 gives an algorithm that computes a point addition in 8 modular multiplications. Comparing Equation 2.21 to Equation 2.3 reveals that finding a 256-bit ECDSA private key has approximately the same complexity as factoring a 3072-bit RSA modulus. This approximation is also found in [10]. Asymptotically, ECC-based schemes are clearly more efficient. A 512-bit ECC-based scheme is approximately as secure as 14848-bit RSA.

2.8.2 Unforgeability

Unforgeability is explained in Section 2.2.2. In short, it means that it is computationally infeasible to generate a valid signature (r, s) for a given message m without knowing the secret key d. Unforgeability implies unbreakability, which is discussed in Section 2.2.1.

ECDSA can be proven unforgeable, but an additional assumption is needed:

Assumption 2.8.1 (Elliptic curve discrete semi-logarithm assumption (ECSLA))**.** *Let $\mathcal{E}$ be an elliptic curve, and let G be a publicly given point on $\mathcal{E}$ of large order n. Let f be the map $f(R) = x_R \pmod n$ for $R = (x_R, y_R)$ on $\mathcal{E}$. Let P on $\mathcal{E}$ satisfy*

$$t = f(u^{-1}(G + tP)), \tag{2.22}$$

for some $t, u \in \{1, \ldots, n-1\}$. Then it is computationally infeasible to find (t, u) from P.

Note that this is a stronger assumption than the ECDLA, Assumption 2.6.1. Suppose we can take discrete logarithms on the elliptic curve $\mathcal{E}$. Then taking semi-logarithms can be done by taking some point T on $\mathcal{E}$, which defines $f(T) = t$. Compute $S = G + tP$, and use the discrete logarithm to obtain u such that $S = uT$. Then $f(u^{-1}S) = t$ as required.

If Assumption 2.8.1 does not hold, then ECDSA can be selectively forged by computing z from m, and setting $P = z^{-1}Q$. Breaking ECSLA gives (t, u) that satisfy $t = f(u^{-1}(G + tP))$, which means that (t, uz) is a valid signature on m.

If Assumption 2.8.1 does hold, then ECDSA is selectively unforgeable in the standard model. To see this, let G and P be given. Select a message m, and set $Q = zP$. Assume we can generate a signature (r, s) on m, then $(t, u) = (r, sz^{-1})$ breaks ECSLA. So unbreakability of ECSLA is equivalent to selective unforgeability of ECDSA.

The only algorithms known to break ECSLA can also break ECDLA. However, ECSLA seems theoretically easier to break than ECDLA, as is shown in [52].

Brown has shown in [24] that ECDSA is existentially unforgeable against adaptive chosen ciphertext attacks under the ECDLA in the *generic group model*. In short, this model assumes that for the elliptic curve group no structure is known, except for adding, subtracting, and negating points. Brown also proves selective unforgeability against adaptive chosen plaintext attacks in the same model.

2.8.3 Malleability

ECDSA signatures are malleable [63]. Given a message-signature pair (m, S), one can find a different pair that is also valid. Here the signature $S = (r, s)$ is as in Definition 2.7.1. Now the pair $S' = (r, -s)$ also is a valid signature for m:

$$f(u_1'G + u_2'Q) = f(-kG) = r, \tag{2.23}$$

since $-kG = -(kG) = (x_k, -y_k)$ has the same x-coordinate as $kG = (x_k, y_k)$. So we obtain a (almost always) different valid pair (m, S'). Hence, the signature scheme is malleable.

The malleability relies on a curve isomorphism that sends the point P to $-P$. No forgeries can be made in this way, since the procedure requires a valid signature, and produces another valid signature for the same message. However, malleability may be an undesirable property in some situations, because it makes it possible to transmit information without being detected. Binary information is encoded in $s \pmod n$, as $s < \frac{n}{2}$ for a 0-bit, and $s \geq \frac{n}{2}$ for a 1-bit. The transmitter of the information does not need to sign messages, he just needs to change signatures. This abuse of malleability is called *oblivious transfer* of information. A practical application of oblivious transfer is leaking company secrets from an otherwise highly secured location.

The malleability can be eliminated by a slight adaptation of the protocol. In this variant s is always chosen in the interval $\{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$. Signatures that do not satisfy this property are then rejected.

2.8.4 Bindingness

Some signature schemes are binding. This property is explained in Section 2.2.4. Bindingness is useful when a sender wants to commit to a message m, but does not want to reveal the message until a later time. Another advantage of bindingness is that a sender cannot send a message m_1, and later claim to have sent message $m_2 \neq m_1$ instead.

ECDSA is not binding, as pointed out in [63], but the last property is not completely lost. Suppose a sender wants to produce a signature (r, s) that is valid for two different messages m_1 and m_2, with $m_1 \neq m_2$. He computes z_1 as the n leftmost bits of $H(m_1)$, and z_2 in a similar way. He selects k at random in $\{1, \ldots, n-1\}$, and computes $r = f(kG)$. He then sets $d = -\frac{z_1+z_2}{2r}$, and computes $s = k^{-1}(z_1 + rd)$ in the usual way. Note that $k^{-1}(z_2 + rd) = k^{-1}(-2rd - z_1 + rd) = k^{-1}(z_1 + rd) = s$, so indeed r, s is a valid signature for both m_1 and m_2.

This renders ECDSA useless as a commitment scheme, but a sender who sends m_1, and later claims to have transmitted $m_2 \neq m_1$ is very probably lying. Given m_1 and m_2 the private key d is recovered as $d = -\frac{z_1+z_2}{2r}$, where z_1 and z_2 are the n leftmost bits of $H(m_1)$ and $H(m_2)$ respectively. If the sender claims that m_1 was constructed by an adversary, then he actually claims that an adversary recovered the private key d from m_2 and the signature (r, s). By the ECSLA this is computationally infeasible.

2.8.5 Hidingness

Hidingness is explained in Section 2.2.5. ECDSA is hiding even if the hash function H is invertible. To see this, consider the expression $s = k^{-1}(z + rd)$. The values r, s, and z are publicly known, but it is still infeasible to recover d, since k is unknown. Similarly, z cannot be recovered from r and s. So the scheme is hiding even if m can be reconstructed from z.

2.8.6 Malicious parameter generation

A malicious signer may generate elliptic curve parameters such that two chosen messages m_1 and m_2 have the same signature. The signer has no complete freedom on the messages, but two random messages m_1 and m_2 have a non-negligible probability of producing valid parameters. Let m_1 and m_2 be given and take $z_1 = H(m_1)$ and $z_2 = H(m_2)$. Note that the hash values are not truncated, so z_1 and z_2 have length L_h. Consider z_1 and z_2 as L_h-bit integers. W.l.o.g. assume that $z_1 > z_2$. If $n = z_1 - z_2$ is not an L_h-bit prime number, then the procedure fails. Otherwise, the signer proceeds as follows.

Construct an elliptic curve with n points with the method of *complex multiplication* [44]. This method finds such a curve for a given finite field $\mathbb{F}_q$ with reasonable probability if n satisfies Equation 2.16. By varying q such a curve can be found efficiently [23].

The probability of finding a valid curve from z_1 and z_2 is quite large. Assume that z_1 and z_2 are randomly distributed in $\{0, \ldots, 2^{L_h} - 1\}$. If H is a secure hash funcion then z_1 and z_2 will be quasi-random, so this is a plausible assumption. Now $n = z_1 - z_2$ is an L_h-bit number with probability $\frac{1}{4}$. The probability that n is prime, is estimated with the prime number theorem:

Theorem 2.8.1 (Prime number theorem)**.** *Let $\pi(x)$ be the number of prime numbers less than or equal to x. Then*

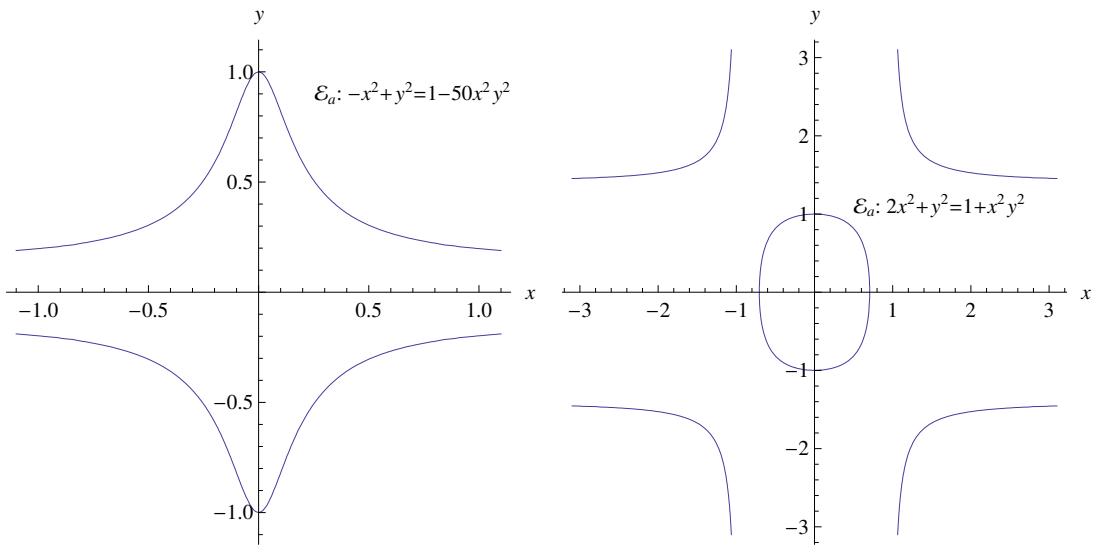
$$\lim_{x \to \infty} \frac{\pi(x)}{\frac{x}{\ln x}} = 1. \tag{2.24}$$

This means that $\pi(x)$ behaves like $\frac{x}{\ln x}$ for large values of x. This estimate is not entirely accurate for small x, but accurate enough for numbers with hundreds of bits. The marginal prime density is obtained by differentiating $\frac{x}{\ln x}$, and is equal to

$$\frac{1}{\ln x} - \frac{1}{(\ln x)^2} \approx \frac{1}{\ln x}. \tag{2.25}$$

By Equation 2.25 n is prime with probability approximately $\frac{1}{\ln n}$. So for $\log_2 n = 256$ bits the probability of finding valid parameters is

$$\frac{1}{4 \ln(2^{256})} \approx \frac{1}{710}. \tag{2.26}$$



(a) A curve with two affine parts

(b) A curve with five affine parts

Figure 2.4: Two twisted Edwards curves over $\mathbb{R}$.

The malicious signer now has valid parameters such that $H(m_1) \equiv H(m_2) \pmod{n}$, which means that any signature for m_1 is also valid for m_2.

Probabilities for given m_1 and m_2 can further be increased by trying different hash functions. If n turns out to be 1 or 2 bits too short then valid parameters may be generated by finding a curve with $2n$ or $4n$ points respectively.

2.9 Twisted Edwards curves

Twisted Edwards curves are a family of elliptic curves introduced by Bernstein et al in [19]. This paper gives very efficient formulas for point addition and point doubling on these curves, which makes them useful for efficient arithmetic. The curve equation of a twisted Edwards curve is

$$ax^2 + y^2 = 1 + dx^2y^2, \tag{2.27}$$

where $a \neq d$, and $a, d \neq 0$.

Two examples of a twisted Edwards curve over $\mathbb{R}$ are shown in Figure 2.4. A Twisted Edwards curve can be written in Montgomery form by substituting $x = \frac{u}{v}$ and $y = \frac{u-1}{u+1}$ [19]. After simplification this gives:

$$4v^2 = (a-d)u^3 + 2(a+d)u^2 + (a-d)u, \tag{2.28}$$

and dividing by $(a-d)$ gives the Montgomery form

$$\frac{4}{a-d}v^2 = u^3 + 2\frac{a+d}{a-d}u^2 + u. \tag{2.29}$$

To avoid inversions in the point addition law, projective coordinates are used. Setting, $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$, Equation 2.27 becomes:

$$Z^2(aX^2 + Y^2) = Z^4 + dX^2Y^2. \tag{2.30}$$

Adding and doubling points on a Twisted Edwards curve can be done very efficiently. To study the efficiency, a common measure for elliptic curve computations is introduced. When computing

a point addition or a point doubling, most of the computation time is spent on multiplications and squarings. The cost of a multiplication is denoted by **M**, and the cost of a squaring is denoted by **S**. Inversions are avoided because they are very slow. Where needed, an inversion is denoted by **I**.

On a twisted Edwards curve, addition in projective coordinates costs $10\mathbf{M} + 1\mathbf{S}$. Doubling costs $3\mathbf{M} + 4\mathbf{S}$. This neglects additions, subtractions, and multiplications by curve parameters. Both algorithms are given in [19].

[36] introduces a fourth coordinate T, defined by $xy = \frac{T}{Z}$. This makes point addition even faster when $a = -1$: point addition only costs $8\mathbf{M}$. But it makes doubling slower: $4\mathbf{M} + 4\mathbf{S}$. Both algorithms are given in [36]. They are presented here as Algorithms 1 and 2.

Algorithm 1 Point doubling on a twisted Edwards curve with $a = -1$:

Input: A point $P = (X1 : Y1 : Z1)$ on the curve
Output: The point $2P = (X3 : Y3 : Z3 : T3)$

$A \leftarrow X1^2$
$B \leftarrow Y1^2$
$C \leftarrow 2 * Z1^2$
$D \leftarrow a * A$
$E \leftarrow (X1 + Y1)^2 - A - B$
$G \leftarrow D + B$
$F \leftarrow G - C$
$H \leftarrow D - B$
$X3 \leftarrow E * F$
$Y3 \leftarrow G * H$
$T3 \leftarrow E * H$
$Z3 \leftarrow F * G$

Algorithm 2 Point addition on a twisted Edwards curve with $a = -1$, where $k = 2d$:

Input: Two points $P1 = (X1 : Y1 : Z1 : T1)$, and $P2 = (X2 : Y2 : Z2 : T2)$ on the curve
Output: The point $P1 + P2 = P3 = (X3 : Y3 : Z3 : T3)$

$A \leftarrow (Y1 - X1) * (Y2 - X2)$
$B \leftarrow (Y1 + X1) * (Y2 + X2)$
$C \leftarrow T1 * k * T2$
$D \leftarrow Z1 * 2 * Z2$
$E \leftarrow B - A$
$F \leftarrow D - C$
$G \leftarrow D + C$
$H \leftarrow B + A$
$X3 \leftarrow E * F$
$Y3 \leftarrow G * H$
$T3 \leftarrow E * H$
$Z3 \leftarrow F * G$

Algorithm 1 does not require the T-coordinate as an input. Therefore, if a doubling is followed by a doubling, the first doubling does not need to compute T. This saves $1\mathbf{M}$. Similarly, when an addition is followed by a doubling, the addition does not need to compute T. Again, this saves $1\mathbf{M}$.

A nice bonus of Algorithms 1 and 2 is that both algorithms are *complete* over a field $\mathbb{F}$ if a is a square, and d is a non-square in $\mathbb{F}$. Also, the addition is *strongly unified*. Strongly unified means that the addition can also be used to double a point. Complete means that the addition and doubling formulas work for all inputs; there are no exceptional cases. Both properties are valuable for making a side-channel-resistant implementation. The handling of exceptional cases is

difficult to hide from side channel analysis. Side channel attacks are discussed in the next section.

2.10 Side channel attacks

A *side channel attack* gains information about a cryptosystem from the physical implementation of the algorithm, rather than through regular cryptanalysis. Typical attacks on smart cards include the monitoring of power consumption, electromagnetic radiation, and computation time. These are non-invasive side channel attacks and are therefore undetectable by the smart card. An example of an invasive attack is *differential fault analysis*, where faults are inserted into the computation. These may be single or multiple bit errors in any type of memory, or faults in the processor that cause it to skip instructions, execute instructions multiple times, or even perform entirely different instructions.

If the implementation of a cryptographic algorithm is not properly protected, a side channel attack may reveal secret information, such as the private key. A classic example is discussed in Section 2.10.2. On a smart card, side channel attacks are a serious concern. Emanation of electromagnetic radiation cannot be effectively shielded on a smart card, nor can the smart card filter its own power trace with complex equipment. In addition, radiation can be measured very close to the source. Java Cards are especially slow, and therefore individual instructions can be discerned without the need for advanced equipment. Moreover, a smart card relies on an external power source, and therefore its computations may be disturbed by short power interrupts.

On the other hand, a smart card uses little power, so the magnitude of all signals is low. Moreover, special hardware is used in secure smart cards to uniformise power consumption, radiation, and/or processing time. Finally, secure smart cards may contain noise generators of different types to decrease the signal-to-noise ratio.

This section discusses the most common side channel attacks on smart cards. The last subsection presents a number of solutions that may prevent these attacks.

2.10.1 Timing analysis

Timing analysis studies the execution time of an algorithm. The total running time of an algorithm may reveal some secret information, but to break a cryptographic scheme more detailed information is usually required. Therefore timing analysis is often combined with simple power analysis which is discussed in Section 2.10.2, or with electromagnetic radiation analysis which is discussed in Section 2.10.4. These techniques yield more useful information than timing only, but the example discussed in Section 2.10.2 relies on timing analysis only.

2.10.2 Simple power analysis

Simple power analysis (SPA) studies the power consumption of the device performing a cryptographic computation. This is practical for a smart card with a contact interface. The power consumption may be measured with a resistor and an oscilloscope. Figure 2.5 shows part of a power trace of a typical unprotected RSA implementation. No power measurements were taken; the power trace contains simulated data.

The power trace in Figure 2.5 corresponds to a small part of an RSA computation. Here $x^d \pmod N$ is computed with the *square and multiply* method. This method is described in Algorithm 3.

A squaring can be done faster than a multiplication. In the implementation of Figure 2.5 this was used to speed up the computation. The first peak in the power consumption is wider than the second peak. Therefore, the first peak corresponds to a multiplication, while the second peak corresponds to a squaring. Since a multiplication is performed only when a 1-bit occurs in d, this analysis reveals the binary representation of the secret exponent d.

This attack is easily extended to attack a *square and multiply* method that evaluates the secret exponent in the opposite direction, or to attack the *double and add* method, which is the equivalent

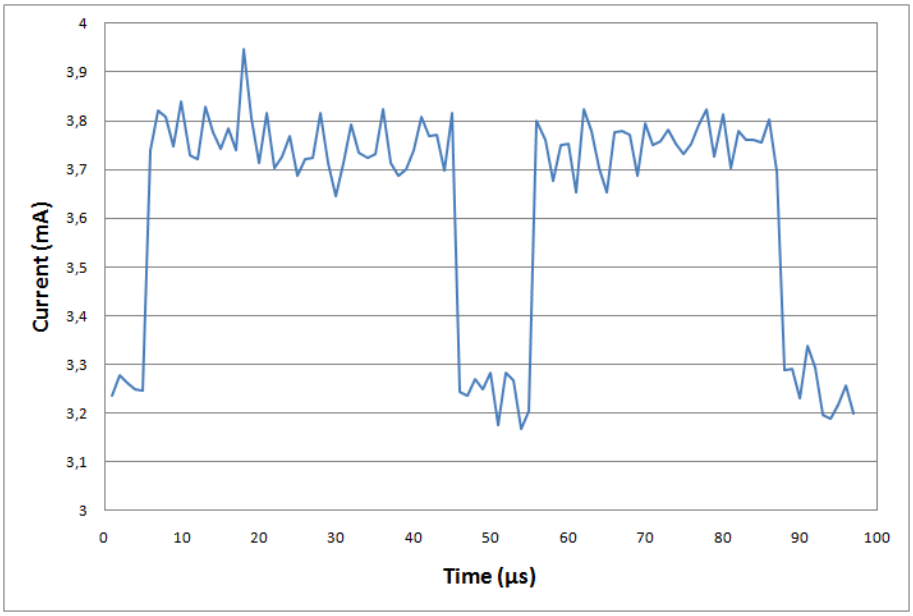


Figure 2.5: Typical power trace of a square and multiply implementation of RSA.

Algorithm 3 Square and multiply algorithm for computing $x^d \pmod N$:

Input: x, d, where d is represented as an array of bits with the most significant bit at index 0
Output: $y = x^d \pmod N$

```
y ← 1
for i = 0 to length(d) do
    y ← y^2 (mod N)
    if d[i] == 1 then
        y ← y * x (mod N)
    end if
end for
```

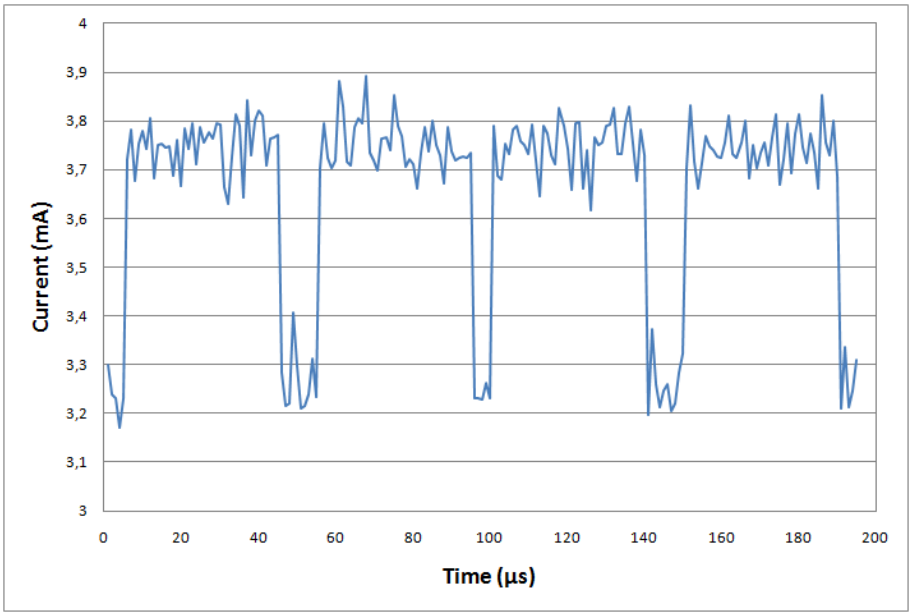


Figure 2.6: Typical power trace of an algorithm with branching points.

method for ECC computations.

SPA also reveals *branching points* in the computation. These are points where the computation makes a conditional jump. A typical example is an IF-THEN-ELSE construction. Consider Algorithm 4. This algorithm is almost the same as Algorithm 3, but now the squaring has been replaced by a multiplication. Therefore, the power analysis described above no longer applies. Of course, the Hamming weight of the exponent is leaked, but [16] shows that this only provides 6.06 bits of information for a 1024-bit RSA key, or at most 3.6 bits for strong primes p and q [17]. However, it may be possible to exploit the branching point in the IF-construction of Algorithm 4.

Algorithm 4 Multiply and multiply algorithm for computing $x^d \pmod N$:

Input: x, d, where d is represented as an array of bits with the most significant bit at index 0
Output: $y = x^d \pmod N$

```
y ← 1
for i = 0 to length(d) do
    y ← y * y (mod N) // NOTE: do not use a squaring here!
    if d[i] == 1 then
        y ← y * x (mod N)
    end if
end for
```

A typical power trace of Algorithm 4 is shown in Figure 2.6. All the peaks now have the same width, since a squaring is implemented by a multiplication. However, the intervals between the peaks are not the same. The interval between the second and the third peak is shorter than the other intervals. The reason is the IF statement in Algorithm 4. If it evaluates to TRUE then the JVM proceeds to the next line. If it evaluates to false, then it skips to the line marked by ENDIF, and proceeds from there. Jumping usually costs more time than not jumping, so in Figure 2.6 no jump is made between the second and the third peak. So the third peak corresponds to a multiplication. Again, the secret exponent d is completely revealed.

SPA reveals more information than the timing of the operations only. It can also be used to distinguish an operation with a low power consumption from an operation with a high power

consumption, even when these operations take the same time. For branching points that always use a jump, such as some SWITCH statements, the characteristic shape of the power trace may reveal which jump is made. Sometimes a single power trace is not sufficient because the measured effect is smaller than the noise in the measurement. The next section explains how to exploit minor effects by using repeated measurements on varying data.

2.10.3 Differential power analysis

Differential power analysis (DPA) is a technique similar to SPA. It analyses the power consumption of a device that performs a cryptographic computation. DPA uses repeated measurements and statistical analysis to obtain information about secret data. It requires knowledge of either the input or output of the algorithm. In most cryptographic protocols at least one of these is public.

Consider Algorithm 5 that computes the modular exponentiation in RSA. It is an improvement to Algorithm 4 because it avoids branching points.

Algorithm 5 Branch-free algorithm for computing $x^d \pmod N$:

Input: x, d, where d is represented as an array of bits with the most significant bit at index 0
Output: $y[0] = x^d \pmod N$

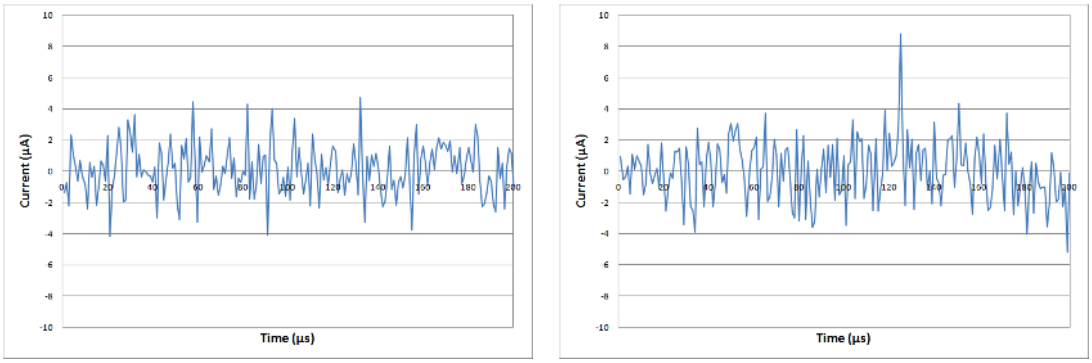
```
y[0] ← 1
for i = 0 to length(d) do
    y[0] ← y[0]^2 (mod N)
    y[1] ← y[0] * x (mod N)
    y[0] ← y[d[i]]
end for
```

Note that in Algorithm 5 the squaring from Algorithm 3 is reintroduced. The algorithm now behaves regularly, performing a fixed number of cycles that each consist of a squaring, a multiplication, and a copying operation. However, the algorithm is computationally more expensive than the previous algorithms in this section. And, unfortunately, it may still be susceptible to DPA.

In many hardware components the power trace depends on the actual bit values that are processed. A common phenomenon is that power consumption varies with the Hamming weight of the word that is processed. For example, the power consumption of an addition operation may increase with the number of 1-bits in the result. Such a dependency is not very strong, and usually much weaker than random noise in the power consumption. But combining many measurements may expose this effect. Consider Algorithm 5, and assume that $d[0] = 1$ (if this is not the case than DPA may also reveal this). Then after the first iteration $y_1[0] = y_1[1] = x$. Consider the internal state of the algorithm after the second iteration. If $d[1] = 0$ then $y_2[0] = x^2 \pmod N$, and if $d[0] = 1$ then $y_2[0] = x^3 \pmod N$. In the third iteration $y_2[0]$ will be squared, yielding either $y_2[0]^2 = x^4 \pmod N$ or $y_2[0]^2 = x^6 \pmod N$. Note that only one of these values will be computed during the execution of the algorithm, ignoring some extremely rare exceptions. It is assumed that the power trace of this squaring operation depends on the first bit of $y_2[0]^2$. A similar analysis is possible for different bits of $y[0]^2$, or for blocks of bits. It is also assumed that the attacker observes the decryption of k known plaintexts $K = \{x_1, \ldots, x_k\}$. DPA is also feasible if the attacker knows k ciphertexts instead. The attacker splits the plaintexts into two sets S_0 and S_1. S_0 contains the plaintexts x_i for which x_i^4 starts with a zero bit, i.e. $S_0 = \{x \in K | (x^4 \pmod N)_0 = 0\}$. Similarly $S_1 = \{x \in K | (x^4 \pmod N)_0 = 1\}$. Let $C_i(t)$ be the power trace that corresponds to plaintext x_i. Then the correlation function $g(t)$ is defined by

$$g(t) = < C_i(t) >_{x_i \in S_1} - < C_i(t) >_{x_i \in S_0}, \tag{2.31}$$

where $< . >$ denotes the mean. If $x^4 \pmod N$ is computed then there is an expected correlation in the third round of the exponentiation. This will result in a peak in $g(t)$. If $x^6 \pmod N$ is computed instead, then no significant correlation is expected. Note that there is some correlation between the bit representation of $x^4 \pmod N$ and $x^6 \pmod N$. Nevertheless, there will be a clear



(a) Incorrect guess (b) Correct guess

Figure 2.7: Two correlation functions in a differential power analysis on a DES implementation. For the correct guess the correlation function shows a peak at $t = 125\ \mu s$. For the RSA implementation of Algorithm 5 a similar peak in the DPA plot will occur for the correct guess $y_2[0]^2 = x^4 \pmod N$, while no peak will occur for the incorrect guess $y_2[0]^2 = x^6 \pmod N$.

difference between the correlation plots corresponding to $x^4 \pmod N$ and $x^6 \pmod N$. Figure 2.7 shows typical correlation functions for an incorrect and a correct guess. Note that the scale of the y-axis is much smaller than the scale of the SPA plots in Section 2.10.2. Therefore typically 100 to 1000 measurements are needed to reveal a peak. More sophisticated variants need as few as 20 power traces to reveal a complete RSA key [41]. Once the first bit of the secret exponent d is known, a similar attack is performed on the second bit. The power traces may be reused for this purpose.

For DPA the attacker needs to know something about the algorithm and the internal representation of the values. Mamiya et al prevent this by using a blinding technique [46]. For elliptic curves, this technique uses a random point R. dP is computed as follows:

Algorithm 6 Binary expansion with a random initial point (BRIP):

Input: d, P
Output: $T = dP$

```
R ← randompoint()
T ← R, T_0 ← −R, T_1 ← P − R
for i = n − 1 to 0 do
    T ← 2T
    T ← T + T_{d_i}
end for
T ← T + T_0
```

An equivalent technique for RSA computes x^d by setting $T = r$, $T_0 = r^{-1}$, and $T_1 = xr^{-1}$. This technique is quite expensive, since it uses an inversion. Worse, Kim et al have published a *second order differential power attack* on the RSA variant of BRIP [41]. They use the second order power trace, defined by

$$g_2(t) =< C_i(t+\delta) - C_i(t) >_{x_i \in S_1} - < C_i(t+\delta) - C_i(t) >_{x_i \in S_0} . \tag{2.32}$$

In Equation 2.32 the parameter δ is the time delay between two points of interest in the computation. Kim et al compare the processing of the most significant bit of d to the processing of every other bit of d. When random processor clocking is used, 70 power traces are sufficient to reveal the secret key. By changing the order of multiplication, this increases to 88000 power traces.

This attack works because loading T_0 gives a different power trace than loading T_1. A DPA of order higher than 2 is also possible, but the signal-to-noise ratio quickly decreases as the order increases.

2.10.4 Electromagnetic radiation analysis

Electromagnetic radiation analysis (ERA) measures radiation emitted by the smart card. The energy of the radiation produces a trace related to a power trace. Getting a good ERA trace requires some effort to eliminate background radiation from other devices. ERA covers an entire spectrum of measurable frequencies, which makes it possible to distinguish different hardware components. By measuring radiation at different locations with respect to the smart card, location-dependent radiation patterns can be constructed.

2.10.5 Differential fault analysis

Differential fault analysis (DFA) uses faults that occur during a cryptographic computation. These faults may occur naturally due to hardware glitching, or they may occur due to interference of an attacker. For some smart cards, interference is possible with power supply glitching [12], laser pulses [62], or external electric fields [15], but there are also other methods. This type of interference is known as *fault injection*. [15] gives a detailed overview of fault attacks.

[29] explains how to attack a Montgomery ladder point-scalar multiplication on an elliptic curve, using one or two fault insertions. The attack only works if points are represented by one coordinate, and the quadratic twist of the curve has a smooth order. A fixed base point G on the curve $\mathcal{E}$ is injected with a fault at the beginning of the computation. With probability $\frac{1}{2}$ the faulty point G' is on the quadratic twist $\tilde{\mathcal{E}}$. If not, the attack fails, and the attacker tries again. If the attack succeeds, the output dG' can be used to retrieve d. This requires guessing G' from G, and solving the discrete logarithm problem on $\tilde{\mathcal{E}}$. Solving the discrete logarithm problem on $\tilde{\mathcal{E}}$ is easy since $\tilde{\mathcal{E}}$ has a smooth order, and guessing G' takes at most 256 trials if the fault is inserted into an 8-bit register. This attack only works for Montgomery curves, where only the x-coordinate is used in the computation.

2.10.6 Preventing side channel attacks

This section discusses common methods for preventing side channel attacks. The list is certainly not extensive. Some of the discussed methods were used in the reference implementation.

Dummy operations

A common countermeasure against side channel attacks are dummy operations. Looking back at Algorithm 3, the main problem there is that an attacker can learn from the power trace when a multiplication is performed. A solution is to always do a multiplication. When the multiplication is not needed for the computation, a dummy multiplication is done instead. This is the same as a normal multiplication, but the results is discarded, and the original value is kept.

Dummy operations can be discovered by inserting faults into the computation. For an example, see Section 2.10.5. Suppose an attacker succeeds in disturbing the cryptographic computation. If the disturbance occurs during a dummy operation, the computation will be valid, and its output will be normal. If the disturbance occurs at any other time, then the result of the computation will probably be invalid. By repeating this procedure, an attacker can find which operations are dummy operations, and, in the situation described above, recover the secret key.

A disadvantage of dummy operations, is that they introduce a lot of overhead. In the RSA example above, dummy multiplications would increase the computational time with more than 33%. Other ways of preventing side channel attacks usually introduce less overhead.

Randomisation

Randomisation is a technique for hiding the internal representation of secret data. This technique is very effective for preventing DPA. For points on a twisted Edwards curve represented as $P = (X : Y : Z)$ or $P = (X : Y : Z : T)$, an easy randomisation technique exists: pick a random number $r \in \mathbb{F}_q$. Multiply all coordinates with r. This yields the same point P, since the fractions $x = \frac{X}{Z}$, $y = \frac{Y}{Z}$, and $xy = \frac{T}{Z}$ all remain the same. An attacker will now have a very hard time determining even a single bit of the representation of P.

This technique only works if all coordinates of P are non-zero. For Twisted Edwards curves (see Section 2.9) the points with zero coordinates have low order. The only point of low order in the subgroup generated by the base point G, is the neutral element $\mathcal{O}$. $\mathcal{O} = nG$ will not occur in the computation of rG (see Section 3.1), because $r \in \{1, \ldots, n-1\}$. However, in the reference implementation r may be larger than n. It is explained in Section 5.3.1 how $\mathcal{O}$ is avoided in this case.

In the reference implementation, G is randomised every time before a signature is generated. This means that all elliptic curve computations start with a freshly randomised version of G. Therefore, the correlation between the power traces of two different computations, is not expected to reveal any secret information.

Physical countermeasures

Smart cards usually offer physical protection against side channel attacks. This eliminates the need for some countermeasures, or makes other countermeasures more effective. Some common physical countermeasures are discussed here.

Random processor clocking By using a clock with a large variance, timing-related attacks become less effective. This includes SPA, ERA, and especially DPA. The number of DPA traces needed increases by one or two orders of magnitude [41]. The costs of this countermeasure are a more expensive smart card, and possibly a reduced average clock rate.

Monitoring A smart card can monitor physical quantities like temperature, electromagnetic radiation, and its power supply. Whenever any of these quantities is outside the permissible range, the smart card resets. If the monitoring equipment fails, the smart card also resets. When this countermeasure is implemented well, it is an effective protection against DFA. But one should keep in mind that an attacker can always try to insert a fault through a mechanism that is not monitored by the smart card. The costs of this countermeasure are a more expensive smart card, and some extra power consumption.

Shielding Some types of interference can be shielded, even on a smart card. The most common shielding technique for smart cards is encapsulating the chip in a hard, opaque epoxy that is resistant to common solvents [5]. The epoxy shields the chip from laser interference, a type of DFA. This countermeasure is very cheap.

Noise generation A smart card can generate random noise in its power trace and radiation. This may be done with dedicated hardware or by letting the processor occasionally execute random instructions. This noise increases the number of measurements needed for SPA, ERA, and DPA. The costs of this countermeasure are a more expensive smart card, and some extra power consumption.

Power trace flattening A smart card can flatten its power trace by always using the maximum amount of power available. Power that is not used by the processor is dissipated, for example with an adjustable resistor. This makes SPA and DPA impossible, but ERA can reveal what power is used by the processor, and what power is dissipated by flattening. The costs of this countermeasure are a more expensive smart card, and maximum power consumption.

2.11 Random number generation

ECDSA uses a random number k for each signature, see Section 2.7. If one can reconstruct this random number, the secret key d is recovered as $d = \frac{ks-z}{r} \pmod{n}$. The scheme that is introduced in Chapter 3 allows a similar reconstruction of the secret key from a secret random number. Therefore, it is important to use cryptographically secure random numbers.

Secure pseudo-random number generators exists, but they are usually not very fast. [48] discusses two secure pseudo-random number generators. The most efficient of these is the Blum-Blum-Shub pseudo-random bit generator. This generator relies on the intractability of integer factorisation. Therefore, it needs additional protection against fault attacks, because changing the modulus N can produce a modulus N' that is easy to factor.

In the reference implementation, true random numbers were used. These are generated by the smart card. For true random number generator, many sources are possible. Some examples of true random number generators are mentioned in [48]. These include the frequency of a free running oscillator, and the capacitor with the greatest charge among two identical capacitors. A true random number generator can also be disturbed by fault attacks. Extreme temperatures, electric fields, or radiation may determine or bias the generated random numbers.

3 A digital signature algorithm for Edwards curves

This section describes a digital signature algorithm for Edwards curves. The algorithm has similarities with both ECDSA from Section 2.7, and EC-KCDSA which is described in [40]. Among the common signature schemes, the Schnorr signature [49] is most similar to it.

This section describes setup, message signing, and signature verification. It also discusses the security, efficiency, and some additional properties of the signature scheme. A smart card implementation of the signing and verification algorithms is described in Chapter 5.

3.1 The signature scheme

The signature scheme relies on a security parameter s_p, so that an attack takes an expected number of 2^{s_p} operations. The bit length of the scheme is computed as $N = 2s_p + 2$. The term "+2" is a correction, because the used curves will have cofactor $h = 4$.

Let $\mathbb{F}_q$ be a finite field, so $q = p^e$, with p a prime number. $2^{N-1} - 2^{\frac{N}{2}} < q < 2^N + 2^{\frac{N}{2}}$. A more common choice is $2^{N-1} < q < 2^N$, but increasing the range, allows an implementer to use convenient fields, like $\mathbb{F}_p$ with $p = 2^{255} - 19$ for $N = 256$. The range for q is increased with a factor of $1 + 2^{2-\frac{N}{2}}$, so this has a negligible impact on the security. The number of points N_p on $\mathcal{E}$ will be at least $2^{N-1} - (1 + \sqrt{2})2^{\frac{N}{2}}$ by the Hasse bound (Equation 2.16) and some approximations.

Let $\mathcal{E}$ be a twisted Edwards curve over $\mathbb{F}_q$, where $\mathrm{char}(q) \neq 2$. A variant for binary fields is not specified here, but a similar scheme can be used in that case. $\mathcal{E}$ is defined projectively by

$$\mathcal{E} = \{(X : Y : Z) \mid Z^2(aX^2 + Y^2) = Z^4 + dX^2Y^2 \pmod{p}\}. \tag{3.1}$$

The parameters a and d are chosen in $\mathbb{F}_q$, with the restriction that a is a square in $\mathbb{F}_q$, and d is a non-square in $\mathbb{F}_q$. This ensures that the point addition on $\mathcal{E}$ is complete [19].

$\mathcal{E}$ is chosen such that it has $\#|\mathcal{E}| = 4n$ points, with n a prime number. The base point G is chosen verifiably at random on $\mathcal{E}$ as follows.

1. Pick a random bit string s_r of length N.
2. Compute $h_r = \text{SHA-2}(q|a|d|s_r)$.
3. Set x as the unsigned integer defined by the bits of h_r.
4. If $x > p$ go to 1.
5. If $\frac{1-ax^2}{1-dx^2}$ is not a square in $\mathbb{F}_q$, go to 1.
6. Compute y as the solution to $y^2 = \frac{1-ax^2}{1-dx^2}$ in $\left\{-\left\lfloor \frac{p}{2} \right\rfloor, \ldots, \left\lfloor \frac{p}{2} \right\rfloor\right\}$, with sign equal to the first bit of s_r. Here 0 means positive and 1 means negative. For non-prime fields this defines the sign of the constant term in some fixed polynomial representation of y.

7. Define the point $P = (P_X : P_Y : P_Z)$ by $P_X = x$, $P_Y = y$, and $P_Z = 1$. Set the base point G as $G = 4P$.
8. If $G = \mathcal{O}$, go to 1.

Setting $G = 4P$ ensures that the base point has order n, and not order $2n$ or $4n$. A point of order $2n$ or $4n$ would be less secure, for the following reason: suppose G has order $2n$, and the signing operation computes the public value $R = rG$ from the secret value r. If r is even, then R has order n. If r is odd, then R has order $2n$. The order of R is easily checked, revealing one bit of r.

The sign of y may be chosen before x and y are computed. The definition makes sure that an honest setup picks G at random on $\mathcal{E}$, where all points in the subgroup of order n have equal probability.

The secret key k_A is chosen at random in $\{2, \ldots, n-1\}$. The public key is computed as the point-scalar multiplication $Q_A = k_A G$.

Signing a message is done as follows:

1. Take at random $r \in_R \{1, \ldots, n-1\}$.
2. Compute on $\mathcal{E}$ the point-scalar multiplication $R = rG$.
3. Take $R_{xyz} = (R_x|R_y|R_z)$.
4. Set $h = \text{SHA-2}(R_{xyz}|m)$.
5. If $h = 0$ go to 1.
6. Compute $s \equiv k_A - rh \pmod{n}$.
7. The signature is $(0|R_{xyz}|s)$.

In short:

$$s \equiv k_A - rH(0|rG|m) \pmod{n}. \tag{3.2}$$

R is always represented in projective coordinates $(X : Y : Z)$. The T-coordinate (see Section 2.9) is optional. If T is transmitted, this is indicated by setting the first bit of the signature equal to 1 instead of 0.

Verification of a signature is done as follows:

1. Verify the validity of Q_A.
2. Verify that $0 \leq s < n$.
3. Verify that R is on $\mathcal{E}$.
4. Set $h = \text{SHA-2}(R_{xyz}|m)$.
5. Verify that $sG + hR = Q_A$.

In short:

$$sG + H(0|R|m)R \stackrel{?}{=} Q_A. \tag{3.3}$$

It is not specified how Q_A should be verified. Two possibilities are that Q_A is signed by a certification authority, or that Q_A is in a list of trusted parties. For interoperability, s is represented as an unsigned integer in $\{0, \ldots, n-1\}$. R is represented as a 3-tuple of unsigned long integers.

In ECDSA, the case $s = 0$ is excluded, see Section 2.7. In this scheme, $s = 0$ is perfectly acceptable. For an accepted signature with $s = 0$, we find that $H(R|m)R = Q_A$. It is clear that an adversary cannot find a point R such that this equation holds. In addition, such a signature does not help an adversary in finding k_A. The equation $rh \equiv k_A \pmod{n}$ does not help, nor does the equation $H(R|m)rG = k_A G$.

3.1.1 Correctness

A valid signature is always accepted:

$$sG + hR = (k_A - rh)G + h(rG) = k_A G = Q_A. \tag{3.4}$$

In this signature scheme, s is reduced modulo n, but this does not change the correctness. G has order n, so $(s + kn)G = sG$ for any integer k.

3.1.2 Batch verification

Sometimes, a large number of signatures should be verified at the same time. Verification can then be done more efficiently if the curve $\mathcal{E}$ is the same for all signatures. When all signatures are from the same user, batch verification is even more efficient. This section studies batch verification for multiple signatures from the same user. For multiple valid signatures $(R_1, s_1), (R_2, s_2), \ldots, (R_k, s_k)$ Equation 3.3 gives

$$\begin{aligned} s_1 G + h_1 R_1 &= Q_A \\ s_2 G + h_2 R_2 &= Q_A \\ &\cdots \\ s_k G + h_k R_k &= Q_A. \end{aligned} \tag{3.5}$$

Using random constants $c_1, c_2, \ldots, c_k \in \mathbb{Z}_n^*$, and adding, the following Equation is found:

$$\begin{aligned} c_1(s_1 G + h_1 R_1) &= c_1 Q_A \\ c_2(s_2 G + h_2 R_2) &= c_2 Q_A \\ &\cdots \\ c_k(s_k G + h_k R_k) &= c_k Q_A \\ \hline (c_1 s_1 + \cdots + c_k s_k)G + c_1 h_1 R_1 + \cdots + c_k h_k R_k &= (c_1 + \cdots + c_k) Q_A \end{aligned} \tag{3.6}$$

If exactly one signature is invalid, Equation 3.6 will not hold. If more than one signature is invalid, Equation 3.6 will hold with probability $\frac{1}{n}$. For accepting multiple signatures at once, a higher confidence level may be required. The procedure may be repeated with different random constants to decrease the probability of accepting a set of invalid signatures by a factor $\frac{1}{n}$.

Equation 3.6 saves some point-scalar multiplications. $k - 2$ point-scalar multiplications are saved, at the cost of $2k$ multiplications and additions. With a point-scalar multiplication costing about 2100**M**, the cost of extra additions and multiplications is negligible. The double point-scalar multiplications are replaced by a k-tuple point-scalar multiplication, which can be done even more efficiently. Section 6.4 shows this for quadruple point-scalar multiplication. For many points, the algorithm credited to Bos and Coster in [56] is very efficient. [22] describes batch verfication on 100 signatures using this algorithm, where the numbers $c_1, \ldots, c_{100}$ are 128-bit. It takes approximately 6500 point additions, while 100 separate verifications would require the equivalent of more than 29000 point additions.

Batch verfication is more efficient when the constants are chosen $c_1 = c_2 = \cdots = c_k = 1$. The verification then becomes

$$(s_1 + \cdots + s_k)G + h_1 R_1 + \cdots + h_k R_k \stackrel{?}{=} kQ_A. \tag{3.7}$$

While this choice does not directly exhibit a possible attack, the scheme can no longer be proven secure in this setting. And the scheme becomes malleable, because the batch verification does not depend on the individual values $s_1, \ldots, s_k$. The batch verification always succeeds as long as the sum $s_1 + \cdots + s_k$ remains correct. It is essential that each hash value h_i has the point R_i as an input. This ensures the R_i cannot be chosen after the h_i are fixed. Otherwise an attacker can

set $s_1 = s_2 = 0$, $R_1 = Q_A$, and $R_2 = \frac{2-h_1}{h_2} Q_A$. Then $h_1 R_1 + h_2 R_2 = h_1 Q_A + (2 - h_1) Q_A = 2Q_A$, and the batch verification succeeds.

An even more efficient scheme can be constructed by setting $c_i = h_i^{-1}$, for $i = 1, 2, \ldots, k$. The batch verification then becomes:

$$(h_1^{-1} s_1 + \cdots + h_k^{-1} s_k) G + R_1 + \cdots + R_k \stackrel{?}{=} (h_1^{-1} + \cdots + h_k^{-1}) Q_A. \tag{3.8}$$

While using k inversions, the scheme only requires a double point-scalar multiplication, $k - 1$ ADDs, and some minor computations. Inversions are not extremely expensive to compute, as will be shown in Section 6.1. The scheme of Equation 3.8 is not secure if one of the $s_i, i \in \{1, \ldots, k\}$ is allowed to be 0. But if these are all required to be non-zero, the scheme is only known to be malleable, as in the previous case. To see that it is malleable, let (R_1, s_1) and (R_2, s_2) be valid signatures, and $h_1 = H(0|R_1|m_1)$, $h_2 = H(0|R_2|m_2)$. From these signatures, two different signature pairs (R_1, s_1'), and (R_2, s_2') are derived. Pick $s_1' \neq s_1$ at will. According to Equation 3.8: $(h_1^{-1} s_1 + h_2^{-1} s_2) G + h_1 R_1 + h_2 R_2 = (h_1^{-1} + h_2^{-1}) Q_A$. Set $s_2' = s_2 + h_2 h_1^{-1} (s_1 - s_1')$. Then $h_1^{-1} s_1' + h_2^{-1} s_2' = h_1^{-1} s_1' + h_2^{-1} (s_2 + h_2 h_1^{-1} (s_1 - s_1')) = h_1^{-1} s_1 + h_2^{-1} s_2$, as required, and the rest of Equation 3.8 has not changed. So the batch verification will be successful.

While the two last batch verification schemes are quite efficient, it is recommended to take the constants $c_1, \ldots, c_k$ at random. Then a batch verification is surely as secure as a regular verification.

In ECDSA, the verification computes a double point-scalar multiplication, and then compares the x-coordinates, see Equation 2.19. A procedure for batch verification like Equation 3.6 is only possible when the y-coordinate of this point is known. Only the sign of the y-coordinate is unknown. If ECDSA is slightly adapted to include the sign of the y-coordinate, batch verification is possible. In addition, this makes ECDSA unmalleable.

In EC-KCDSA [40], the verficiation uses the result of a double point-scalar multiplication as the input of a hash function. This ruins the possibility of batch verification, since a proper hash function does not behave linearly.

3.2 Efficiency

This section analyses the efficiency of the scheme introduced in this chapter. The scheme is compared to ECDSA and EC-KCDSA. ECDSA is discussed in Section 2.7, and EC-KCDSA is described in [40].

3.2.1 Computations

The computational complexity of the scheme from this chapter is low compared to ECDSA. For signing, the most expensive operations are a point-scalar multiplication, and a multiplication modulo n. ECDSA also requires a point-scalar multiplication for signing. However, it needs two multiplications and one inversion modulo n, see Section 2.7. EC-KCDSA signing has a point-scalar multiplication and a multiplication modulo n as its most expensive operations. For signing, EC-KCDSA is slightly slower than the scheme introduced in this chapter, because points are represented in affine coordinates.

For verification, the scheme from this chapter requires a double point-scalar multiplication, and a verification of the point R as its most expensive operations. ECDSA also needs these operations, and uses an additional inversion, and two multiplications modulo n. EC-KCDSA verification is slightly faster than the scheme introduced in this chapter, because it does not need to verify that a given point is on the elliptic curve $\mathcal{E}$. Therefore, 1 signing operation plus 1 verification in EC-KCDSA costs $1\mathbf{M} + 2\mathbf{S}$ less than in the scheme from this chapter.

The key generation in EC-KCDSA requires an inversion, which is not needed in the scheme from this chapter. Key generation is a one time only effort, but the need for a side-channel-resistant inversion does increase the code complexity.

3.2.2 Memory

The scheme from this chapter produces relatively long signatures. An ECDSA signature consists of two numbers of $\log_2 n$ bits each, so it has $2\log_2 n$ bits. EC-KCDSA produces signatures of the same length. The scheme from this chapter has signatures with length $4\log_2 n + 1$ bits, or even $5\log_2 n + 1$ bits when the T-coordinate is transmitted. This uses the estimate $\log n \approx \log p$.

The scheme from this chapter can easily be changed to produce shorter signatures. R can be represented as $(R_x|\text{sgn}(y))$, which produces signatures of $2\log_2 n + 1$ bits. However, this requires representing R in affine coordinates, costing a double inversion. In the reference implementation this costs $254\mathbf{S} + 12\mathbf{M}$ (see Section 5.1.7). Also, the reconstruction of R_y requires solving $y^2 = \frac{1-ax^2}{1-dx^2}$. This requires a division, and a square root modulo p. The division increases the code complexity, and the square root computation costs an exponentiation modulo p. The equation $y^2 = \frac{1-ax^2}{1-dx^2}$ needs to be solved anyway when generating G, but this is a one-time effort that need not be done on the smart card.

4 Smart card specifications

The signature scheme of Chapter 3 was implemented on a smart card: the Cosmo ID One Lite v 5.4 (Cosmo 5.4). This card is equipped with a Java Card Virtual Machine (JCVM). Therefore, the algorithms presented in this report were implemented in Sun Microsystem's Java Card 2.1.1. Java Card is a programming language, and is discussed in Section 4.3.

4.1 Processor

The Cosmo 5.4 is a smart card with a contact interface. It has a JCVM that runs applets on the main processor. The main processor is a microcontroller from the Temic C51 family. This is an 8-bit microcontroller with a clock rate of 60 MHz. Addition takes 12 clock cycles, while multiplication takes 48 clock cycles. Moving data takes 12 or 24 clock cycles, depending on the location of the data.

The Cosmo 5.4 also has a coprocessor. The coprocessor is optimised for cryptographic operations such as DES, RSA, and SHA-1. It supports elliptic curves of 161 to 192 bits [2].

4.2 Memory

The Cosmo 5.4 has three types of memory: RAM, EEPROM, and ROM. Read only memory (ROM) is not discussed in this section, since it is not relevant for programming smart card applets.

4.2.1 RAM

The Cosmo 5.4 has 1364 bytes of RAM. Of this memory, 254 bytes is stack memory [2]. The stack is used by the JCVM for executing applets, but it is not addressable from an applet. Another 274 bytes is the APDU buffer. This is the buffer used for communication to and from the smart card. The APDU buffer is directly addressable. The remaining 836 bytes is heap memory. The heap is used by the JCVM for storing objects. Objects declared at runtime are allocated on the heap, unless they are arrays. Arrays may be allocated on the heap or in EEPROM, see Section 4.2.2.

Java Card partitions the RAM into CLEAR_ON_DESELECT and CLEAR_ON_RESET memory. Both types of RAM are cleared whenever a reset command is issued to the card or when power loss occurs. CLEAR_ON_DESELECT memory is also cleared whenever an applet is deselected, that is when another applet is selected. Therefore, this type of memory is the most suitable for storing sensitive data. If desired, sensitive data may also be stored in CLEAR_ON_RESET memory. When an applet is deselected, its "deselect" method will be called. This method will then be responsible for clearing all sensitive data in CLEAR_ON_RESET memory. The APDU buffer is cleared whenever an applet is selected [6]. On the Cosmo 5.4 the APDU buffer memory is cleared at the same time as CLEAR_ON_DESELECT memory. On smart cards that allow selection of multiple applets simultaneously this need not be the case.

4.2.2 EEPROM

Electrically erasable programmable read only memory (EEPROM) is a type of non-volatile memory that is fast to read but slow to write. Usually data are read and written in multiple byte blocks. The measurements in Section 4.5.2 show that writing to EEPROM is roughly 30 times as slow as writing to RAM.

Applets on the smart card are stored in EEPROM. Objects declared by applets may be stored in EEPROM too. The Cosmo 5.4 has 64 kilobytes of EEPROM, of which 512 bytes are the Java Card transaction buffer. The transaction mechanism is explained in Section 4.3.5. Data stored in EEPROM persists even if the smart card receives a reset command or when power loss occurs.

4.3 Java Card

The Cosmo 5.4 supports Java Card. Java Card is a programming language derived from Java, and is specifically designed for smart cards and other small embedded devices with very limited memory and processing power.

A Java Card applet is written as a Java applet, using a subset of the Java language, and some specific instructions from Java Card APIs. The Java applet is converted to a Java Card applet, after which it is uploaded to a smart card. The applet runs completely inside the smart card, and is processed by the JCVM.

The Cosmo 5.4 has restricted functionality for security reasons. For example, it is not possible to program an application in a language different from Java Card. Java Card may of course be used to program insecure applets, but it cannot be used to decrease the security of any secure applets already present on the card. This section explains how this is achieved.

Another advantage of Java Card is that Java is a platform independent language. This means that an application can be run on any smart card that has a JCVM without the need for recompiling. Of course, there may be reasons to redesign the application for a different type of smart card, for example if the smart card offers more memory, or if the processor architecture has longer word lengths.

4.3.1 Assembly optimisation

Any Java based language is compiled to Java bytecode. This is not machine code, and it is not executable by most processors, although Java processors do exist. On non-Java processors the bytecode is executed by a Java Virtual Machine (JVM). The JVM is a piece of software that translates the bytecode into machine instructions, which are then executed by the processor.

Therefore, it is impossible to perform assembly optimisation on the compiled code. However, it is possible to optimise the Java bytecode. Java Card automatically compresses the bytecode to a smaller size, so it more easily fits on a smart card. Performance optimisation must be done by hand.

Bytecode optimisation is not a full equivalent of assembly optimisation. For example, removing dependencies works both in assembly and bytecode optimisation, but using self-modifying code is not possible with Java bytecode. Another difference is that assembly optimisation can focus on a specific type of processor. A speed critical application may be produced in several versions, each one optimised for a different type of processor. With Java Card one should hope that the JCVM makes the right choice when converting bytecode to machine instructions. The JCVM is not optimised for cryptography; it has an API that enables a programmer to leave the cryptography to the cryptographic coprocessor. So the JCVM does not expect to do the cryptography by itself.

4.3.2 Garbage collection

An issue that may affect the performance of Java applications is garbage collection. Java has an automatic garbage collector that deletes objects from memory that are no longer used. A garbage collector does not delete all such objects, nor does it always delete them immediately. Garbage

collection may slow down an application if there are many memory allocations. This issue does not occur in Java Card, since it has no garbage collector. In the implementations in this report the need for garbage collecting was avoided completely by reusing memory slots.

4.3.3 Firewalling

Java Card automatically prevents applets from accessing other applets or data outside their own scope. The JCVM performs checks on array indices just as a normal JVM does. Obviously, this reduces the performance, but for security reasons these checks cannot be omitted. Otherwise an applet could try to access data it should not have access to. These checks impose a huge performance penalty on memory reads and writes. Stack variables are not affected by this, but a typical stack size is only 256 to 512 bytes. Therefore, not all values can be kept on the stack.

If desired, data may be explicitly shared by an applet. Firewalling eliminates the need for encrypting data that are stored in EEPROM. Applet firewalls are mandatory in all implementations of the Java Card Virtual Machine [6].

4.3.4 Sandboxing

The JCVM acts as a sandbox for the applet it is running. The JCVM handles all exceptions that are not handled by the applet itself. The exceptions raised by the JCVM are general, so that an exception does not directly reveal sensitive information. An exception consists of a single error code with no additional information. Common exceptions raised by the JCVM are '69 85': conditions of use not satisfied, '6A 84': not enough memory space, and '6F 00': error, no specific diagnosis. Of course, general exceptions still reveal information, so a developer must ensure that exceptions are handled in a secure way.

The main advantage of sandboxing is that programming errors will always raise an exception and not result in unexpected behaviour. Defining new exceptions costs time, but throwing exceptions is handled quickly and hardly reduces performance [59].

4.3.5 Transactions

Java Card supports a transaction mechanism. This mechanism ensures that a set of instructions is either executed completely, or is not executed at all. This makes the set of instructions behave like an atomic operation. The transaction mechanism saves the machine state at the beginning of the transaction to the transaction buffer in EEPROM. If the end of the transaction is not reached, for example because of power loss, then the machine state is rolled back to the latest saved state as soon as the smart card is powered on. The transaction mechanism may be used to prevent accidental or deliberate occurrence of errors by power loss.

4.3.6 Pointers

Like Java, Java Card does not support pointers. Fortunately, arrays are passed by reference. In the smart card implementation, function calls mostly pass arrays, so there is hardly any performance restriction there. However, it is impossible in Java to manually manipulate memory addresses, so all addresses need to be stored in memory.

For ADD, this means that 16 addresses have to be kept in memory. When the point $(X_1 : Y_1 : Z_1 : T_1)$ is added to $(X_2 : Y_2 : Z_2 : T_2)$, 8 addresses are needed for the X, Y, Z, and T coordinates, and 4 addresses for the result. Additionally, ADD needs 4 auxiliary arrays. Multiplication by the curve parameters a and d are built in functions, so no addresses are needed for these parameters.

Memory is allocated once, when the applet is initialised. The addresses are stored in EEPROM, so no RAM is wasted on this. EEPROM is slow to write, but as fast as RAM to read, so this does not slow down the application. Passing the references does slow down the application, so the performance impact of inlining functions was tested.

4.3.7 Data types and operations

Java Card supports a very limited set of data types and operations. This makes sense, since a smart card has very limited memory and processing capacity. Also, there is no need for features like a graphical user interface or file input and output. This section discusses the most important data types and operations that are supported in Java Card.

Data types Java Card supports 8-bit numerical values called BYTE, and 16-bit values called SHORT. Both represent signed numbers, so a BYTE value ranges from -128 to 127, and a SHORT value ranges from -32768 to 32767. These data types are identical to their Java equivalents. One-dimensional BYTE and SHORT arrays are also supported. The 32-bit Java type INT is supported in some smart cards, but not in the Cosmo 5.4. Java Card always supports booleans, classes, methods, and supertypes such as PRIVATE and STATIC.

Operations Java Card supports all common arithmetic operations such as addition, multiplication, and bitwise operations. These are the same operations as in Java. More complex operations such as exponentiation are not supported. In Java those operations are found in the class "Math". Java Card also supports basic control and flow statements such as IF, SWITCH, FOR, and method calls.

4.4 Java Card APIs

Java Card has several APIs that provide access to some internal functions on the smart card. One of these functions is copying data. The API provides a function to copy (a part of) an array of BYTE or SHORT values. The source and destination may be in any type of memory. It is explained in Section 4.5.2 why this operation is faster than copying data with a loop.

A different Java Card API provides access to the cryptographic functions supported by the smart card. These functions are executed by the cryptographic coprocessor. The Cosmo 5.4 supports RSA, DSA, ECC, ECDSA, 3DES, AES, and SHA-1 through this API.

4.5 Timing of basic operations

To implement an efficient digital signature algorithm on a smart card, it is important to know how efficient the available arithmetic operations are. Most of the operations described in Section 4.4 were timed. This section describes the method of timing and gives the timing results for these operations.

4.5.1 Method of timing

All operations timed are basic Java Card instructions. These instructions were repeatedly performed using a loop. Also, the time to execute an empty loop was recorded. The empty loop takes 121 μs per iteration. This time was subtracted from all other measurements.

Unfortunately, the interface between my PC and the smart card did not have a timing option. Therefore, the timings were done using a stopwatch. This method is further described in Appendix A. The method of timing introduces Gaussian noise in the measurements with zero mean, and a standard deviation of 0.035 seconds.

4.5.2 Timing results

Table 4.1 contains the timing results for most operations supported by the Cosmo 5.4. The intervals are 95 % confidence intervals. The operation "empty loop" increments a counter, and performs a conditional jump.

Operation	Time (μs)
Empty loop	121 ± 1
Increment/decrement	19 ± 1
Add/subtract	74 ± 1
Multiply	65 ± 1
Bitwise AND	62 ± 1
Bitshift	68 ± 1
Cast SHORT to BYTE	15 ± 1
Read/write CLEAR_ON_DESELECT RAM	720 ± 1
Read/write CLEAR_ON_RESET RAM	510 ± 1
Read EEPROM	196 ± 1
Write EEPROM	16285 ± 15
Copy an array from EEPROM to RAM	32 ± 1 per byte
Copy an array from RAM to EEPROM	513 ± 1 per byte
Copy an array from RAM to RAM	23 ± 1 per byte

Table 4.1: Time taken by the Cosmo 5.4 to perform basic instructions. For copying, arrays of 128 bytes were used. The time this operation takes, is the same for shorter arrays, which means the time per byte will increase for shorter arrays.

CLEAR_ON_RESET RAM is significantly faster than CLEAR_ON_DESELECT RAM. It is not apparent why this is the case. A possible explanation is that accessing CLEAR_ON_DESELECT RAM requires more checks by the JCVM, since it may only be accessed if the active applet owns the object that is accessed. CLEAR_ON_RESET RAM is available to any active applet, so the ownership of the object does not need to be checked. For the array copying operations, the faster CLEAR_ON_RESET RAM was used. In Table 4.1 this memory is simply referred to as "RAM".

Reading data from RAM has unexpected behaviour. When data are read from an array, the time of the reading operation depends on the array index. More details are provided in Appendix B. There it is also explained why this behaviour does not reveal side channel information.

As expected, writing to EEPROM is very slow compared to other memory operations. Copying a 128 byte array from RAM to EEPROM takes 65692 μs, which is 513 μs per byte. Writing a single byte value to EEPROM takes 16285 μs, which is just under one fourth of the time it takes to write a 128 byte array. Therefore, it is assumed that on the Cosmo 5.4 EEPROM is written in 32 byte blocks.

Surprisingly, reading from EEPROM is faster than reading from RAM. Probably this is due to caching. Declaring multiple objects in EEPROM does not change the results, which may indicate that multiple objects may be cached simultaneously.

For copying arrays the Java Card API for copying data is used. This is clearly faster than using a loop for copying data. For example, copying a 128 byte array from RAM to RAM costs 2.9 ms using the API, while it costs 146.0 ms using a loop.

4.6 Debugging

There are two ways of debugging a Java Card applet. The first is using a smart card simulator, which is described in Section 4.6.2. The second is testing it on the smart card, which is described in Section 4.6.1. Both methods have similar problems, which are described in Section 4.6.3.

4.6.1 The smart card simulator 'cref'

The Java Card Development Kit version 2.2.1 or newer comes with a smart card simulator, which is the C-language Java Card Runtime Environment (cref). Together with the Java Card platform Workstation Development Environment (jcwde) this allows simulating the installation and running

of smart card applets. 'cref' and 'jcwde' are PC applications. 'jcwde' is used to send APDU commands to 'cref'. 'cref' accepts these APDU commands, and returns responses as a normal smart card does. Since it runs on a regular PC, it is much faster than a smart card. The simulator allows saving and loading of the current smart card state. Also, it provides statistics of how much memory is used, and how many times each operation is performed. However, the simulator has some limitations.

As a first limitation, built-in smart card functions like RSA and AES have only limited support in the simulator. For example, RSA goes up to 512 bits, and for AES, only AES-128 is supported. For a function like RSA, the user may need to use an external RSA implementation, which is inconvenient. Even worse, smart card input and output is restricted to 256 bits, so the input and output must be done in multiple chunks. When RSA operations are used for long integer multiplication, as described in Section 5.2.2, the number of user inputs becomes too large to be practical.

As a second limitation, the simulator 'cref' does not have the same memory layout as the Cosmo 5.4. The simulator is conservative in memory use, and allows only 256 bytes of CLEAR_ON_RESET RAM and 128 bytes of CLEAR_ON_DESELECT RAM [7]. While this is great for developing applets that should work on all smart cards, it is not convenient for an applet that tries to get the best performance on a given card. For cryptography, more memory means better performance, so most of the memory available on the smart card will be needed. This may be solved in the simulator by declaring memory in EEPROM instead of RAM. This does not make the simulator significantly slower. However, quite some lines of code need to be changed; all array declarations need to be programmed in a different way.

4.6.2 The smart card

Testing an applet on the smart card gives the developer all functionality of the smart card. This includes all functions like RSA and AES, which may have limited support in a simulator. Also the developer can use the exact amount of memory available on the card.

However, a smart card is quite slow when used for cryptography. Uploading and installing a 4 kilobyte applet on the smart card takes approximately 10 seconds on the Cosmo 5.4. Elliptic curve cryptography may take anything from seconds to minutes, depending on the bit length, and on whether the desired curve is supported by the cryptographic coprocessor. Compiling and converting an applet takes some additional seconds, as does deleting the previous applet. Therefore, debugging on the smart card is quite time consuming.

4.6.3 Issues

Debugging Java Card applets is especially inconvenient because the errors are usually very general. Most illegal instructions that are not caught by the compiler, result in a '6F 00' status, which means 'error: no specific diagnosis'. Among these are 'array out of bounds' exceptions, and invalid calls to Java Card native methods such as 'ArrayCopy' and 'RSA'. Except for the status word, such as '6F 00', no information is given. Most notably, there is no reference to the line number where the error occurred.

Java Card does not facilitate memory dumps, nor the tracking of values. Once output is generated by the smart card, the applet halts, awaiting further instructions from the host. So the best way of gathering debug information, is saving it in an array, and dumping it all at once upon termination.

During my project I found the following debugging strategy to be the best practice:

- If the smart card returns an error, perform a bisection procedure on the code that may be responsible for the error. Start by raising a different error halfway the code to be debugged. If this different error occurs, repeat the procedure for the remainder of the code. If the current error persists, examine the first half of the code in the same manner.

- Copy intermediate values of interest to an array that is not used in the remainder of the code. Upon termination, output these values along with the normal response of the applet. Alternatively, a separate debug array may be used. However, this is not always feasible because memory is scarce.

5 Implementation

This chapter describes the smart card implementation of the signature scheme from Chapter 3. The smart card on which the implementation was tested, is introduced in Chapter 4.

5.1 Finite field arithmetic

This section describes how the finite field arithmetic was implemented in Java Card. The implementation attempts to minimize the computation time, while keeping the applet size reasonable. Wherever tradeoffs were made between speed and code size, this is underpinned.

The main focus for making a fast implementation, is minimizing the number of memory operations. Table 4.1 in Section 4.5.2 shows that loading and storing values in RAM is by far the slowest basic operation needed in finite field arithmetic. Fortunately, this only holds for values that are stored in an array. Local variables, which are also stored in RAM, are much faster to read and write. For example, the addition and multiplication as timed in Table 4.1, both include loading two local variables, and storing one. To make the implementation faster, local variables are used to store values that are needed more than once.

In this section, memory used for local variables will be referred to as *registers*. While these are not actually registers in the usual sense, this naming models the Java Card environment as a normal processor architecture. Alternatively, the local variables may instead be thought of as cache memory.

5.1.1 Representation

Finite field elements are represented in Java Card as an array of `byte` values. One element of this array is called a *word*. So in this case the word size is 1 `byte`. The numeric types in Java Card are `byte`, which holds 8 bits, and `short` which holds 16 bits. The reason for using a word size of 1 `byte`, is that the product of two `byte` values fits in a `short` value. This means that two words may be multiplied by a simple multiplication.

Representation of a single word

The numeric types `byte` and `short` in Java Card are always signed. This means that one bit is used for the sign, and the other bits represent the numeric value. The notation for this is $sgn|val$, where sgn is the sign, and val is the numeric value. For example, the number 17 is represented as follows:

$$17 = 0|0010001, \tag{5.1}$$

where $sgn = 0$ indicates the number is positive, and $val = 010001$ is the common binary representation of the number 17. This representation has the most significant bit on the left. So, using "$_2$" for binary and "$_{10}$" for decimal notation, we find $010001_2 = 0{\cdot}2^5{+}1{\cdot}2^4{+}0{\cdot}2^3{+}0{\cdot}2^2{+}0{\cdot}2^1{+}1{\cdot}2^0 = 17_{10}$.

Negative numbers are represented in *two's complement notation*. For a number $-x$ with $-2^{n-1} \leq -x < 0$, the n-bit two's complement notation of $-x$ is defined as follows:

Definition 5.1.1 (Two's complement notation)**.** *Let $n \in \mathbb{N}$ be the bit length, and $0 < x < 2^{n-1}$. Let $y_2 = 2^{n-1} - x$ be the $(n-1)$-bit unsigned binary representation of $2^{n-1} - x$. Then the two's complement notation of $-x$ is $1|y_2$.*

As an example, consider the 8-bit representation of the number -12. $-x = -12$ so $x = 12$. Then $y_2 = 2^7 - 12 = 116 = 1110100_2$. Finally, prepend the sign bit, which is 1 for negative numbers, to obtain

$$-12 = 1|1110100. \tag{5.2}$$

There is a one-to-one correspondence between n-bit words and integers in the range $\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$.

A signed `byte` value can be used as an unsigned 8-bit number by interpreting the sign bit as the most significant bit. In Java Card this is done by casting it to a `signed short`, and then computing the bitwise AND with 0000000011111111.

Representation of finite field elements

Finite field elements are represented as an array of words. Each word is a `signed byte` value, which is between -128 and 127. An element of $\mathbb{F}_p$ with $p = 2^{255} - 31$ is represented as an array of 32 `bytes`. Let $a \in \mathbb{F}_p$ and $p = 2^{255} - 31$, then a representation of a is

$$\hat{a} = [a_{31}|a_{30}|\ldots|a_1|a_0], \tag{5.3}$$

such that

$$a \equiv a_{31} \cdot 256^{31} + a_{30} \cdot 256^{30} + \cdots + a_1 \cdot 256 + a_0 \pmod{p}, a_i \in \{-128, \ldots, 127\}. \tag{5.4}$$

This is called a *signed-byte representation*. Since $p = 2^{255} - 31 = -31 + 128 \cdot 256^{31}$, every element of $\mathbb{F}_p$ can be represented in more than one way. For example, 7 can be represented in two ways: $7 \equiv 38 - 128 \cdot 256^{31} \pmod{p}$. Most finite field elements can be represented in two ways, some in three ways.

Here are some examples of numbers in signed-byte representation, using two words:
$\hat{7} = [0|7]$
$\widehat{-7} = [0|-7]$
$\widehat{144} = [1|-112]$
$\widehat{-144} = [-1|112]$
$\widehat{260} = [1|4]$
$\widehat{-260} = [-1|-4]$.

5.1.2 Modular reduction

Whenever finite field elements are added, subtracted, multiplied, or squared, the result may no longer be represented in the form of Equation 5.4. Therefore a reduction modulo p is done after each such operation. For addition and subtraction, a reasonable alternative is to use one extra byte in the representation. Then a modular reduction only needs to be done after a multiplication or a squaring. However, modular reduction after an addition is very efficient for $p = 2^{255} - 31$, and therefore we rather save a byte in the representation. Note that using an additional byte would make multiplication and squaring slower.

Roughly, finite field elements are reduced to the interval $\{-\lfloor \frac{p}{2} \rfloor, \ldots, \lfloor \frac{p}{2} \rfloor\}$. This keeps their absolute value as small as possible, which again produces small results upon addition or multiplication. These results are then easier to reduce modulo p. "Roughly" means that the numbers may fall outside this range by no more than 1%. A more precise explanation follows.

As an example, let $a, b \in \mathbb{F}_p$, with $-\left\lfloor \frac{p}{2} \right\rfloor \leq a, b \leq \left\lfloor \frac{p}{2} \right\rfloor$. Then $-p \leq a+b \leq p$. The sum $s = a+b$ is represented in the same way as a and b. How s is computed will be explained in Section 5.1.3. Modular reduction of s is done in the following way:

$$s_{reduced} := \begin{cases} s - p & \text{if } s_{31} \geq 64 \\ s + p & \text{if } s_{31} < -64 \\ s & \text{otherwise} \end{cases} \tag{5.5}$$

This method resembles the ideal reduction, which would be

$$s_{reduced} := \begin{cases} s - p & \text{if } s > \frac{p}{2} \\ s + p & \text{if } s < -\frac{p}{2} \\ s & \text{otherwise} \end{cases} \tag{5.6}$$

The method of Equation 5.6 always outputs $s_{reduced}$ in the range $\{-\left\lfloor \frac{p}{2} \right\rfloor, \ldots, \left\lfloor \frac{p}{2} \right\rfloor\}$. For the method of Equation 5.5 the range is $\{-64 \cdot 256^{31} + 31 - \left\lfloor \frac{256}{255} \cdot 128 \cdot 256^{30} \right\rfloor, \ldots, 64 \cdot 256^{31} - 1\}$. Then $|a + b| < p + \left\lfloor \frac{p}{2} \right\rfloor$, so the reduction may deviate from Equation 5.6 only if $s_{31} = \pm 64$. The smallest result then occurs when $\hat{s} = [\,64\,|\,-128\,|\,-128\,|\ldots|\,-128\,]$, resulting in $s_{reduced} = -64 \cdot 256^{31} + 31 - \left\lfloor \frac{256}{255} \cdot 128 \cdot 256^{30} \right\rfloor$. The largest possible result occurs when $s = 64 \cdot 256^{31} - 1$, resulting in $s_{reduced} = 64 \cdot 256^{31} - 1$.

The reduction method in Equation 5.5 is more efficient, since it checks the most significant word of $\hat{s}$, instead of checking $\hat{s}$ completely. One might suggest that checking s_{31} is usually sufficient, and s_{30} only needs to be checked if $s_{31} = \pm 64$. However, such a method would reveal side channel information, since the checking takes longer if $s_{31} = \pm 64$.

The modular reduction method used in addition is slightly different than explained above, because it is combined with carrying. The method is completely explained in Section 5.1.3.

Avoiding a conditional jump

The naive way of programming Equation 5.5 would be with conditional jumps. However, this would reveal side channel information. To be precise, it would reveal whether $s_{31} < -64$, $-64 \leq s_{31} < 64$, or $64 \leq s_{31}$. In the reference implementation I could not find a vulnerability that exploits this information. But, a closely related implementation may be attacked with this information. Consider point addition on the elliptic curve. This addition loads a precomputed point $P_i = (X_i : Y_i : Z_i : T_i)$ that is added to the current point P_{cur}. Algorithm 2, among other things, computes $Y_i + X_i$. Each time P_i is used in a point addition, this same computation is performed. So a distinction between precomputed points can be made. For $P_i = (X_i : Y_i : Z_i : T_i)$, define $S_i = Y_i + X_i$. Then there are three categories:
$\mathcal{P}_- = \{P_i | S_i[31] < -64\}$
$\mathcal{P}_0 = \{P_i | -64 \leq S_i[31] < 64\}$
$\mathcal{P}_+ = \{P_i | S_i[31] \leq 64\}$.
This distinction helps an attacker to decide whether two points P_j and P_k used in different point additions are the same point. If $P_j \in \mathcal{P}_-$ and $P_k \in \mathcal{P}_+$ then these points cannot be the same.

On average $\frac{1}{8}$ of the points belongs to $\mathcal{P}_-$, $\frac{1}{8}$ belongs to $\mathcal{P}_+$, and $\frac{3}{4}$ belongs to $\mathcal{P}_0$. So for 16 precomputed points there are on average 2 points in $\mathcal{P}_-$, 2 points in $\mathcal{P}_+$, and 12 points in $\mathcal{P}_0$. This means that the side channel information is $-\frac{1}{8}\log_2\frac{1}{8} - \frac{1}{8}\log_2\frac{1}{8} - \frac{3}{4}\log_2\frac{3}{4} \approx 1.06$ bits per point addition. This is 68 bits of information in total. However, this assumes that the attacker knows which points correspond to which category, which is not the case. Therefore $-\log_2\left(\frac{2}{16}\frac{1}{15}\frac{2}{14}\frac{1}{13}\right) \approx 13$ bits of information should be subtracted, resulting in 55 bits of information obtained by side channel analysis.

In the reference implementation this attack does not work, since the values $Y_i + X_i$ are precomputed. Nevertheless, related attacks cannot be excluded as long as conditional jumps are used in the modular reduction.

Therefore, conditional jumps are avoided. This is not very costly, and therefore considered best practice. Equation 5.5 is implemented by using the bitwise representation of s. Let

$-\frac{255}{128}p < s < \frac{255}{128}p$, and let $\hat{s}$ be the signed-byte representation of s, introduced in Section 5.1.1, with the exception that s_{31} is not restricted to $\{-128, \ldots, 127\}$. s_{31} is represented as a `signed short` value. This is equivalent to a `signed byte` value, except that it has 16 bits instead of 8. $-\frac{255}{128}p < s_{31} < \frac{255}{128}p$, so $-255 \leq s_{31} \leq 255$. The bitwise representation of s_{31} is
$s_{31} = sgn \,|\, b_{15}\, b_{14}\, b_{13}\, b_{12}\, b_{11}\, b_{10}\, b_9\, b_8\, b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0$

Lemma 5.1.1. *Let*

$$\alpha = (2b_8 - b_7 - b_6), \tag{5.7}$$

then $s + \alpha p = s_{reduced}$ *as in Equation 5.5 for* $-192 \leq s_{31} < 192$. *For* $-256 \leq s_{31} \leq -193$ *we have* $\alpha = 2$, *and for* $192 \leq s_{31} \leq 255$ *we have* $\alpha = -2$.

Proof. Eight cases are distinguished:

1. $192 \leq s_{31} \leq 255$: then $s_{31} = 0|000000\mathbf{011}\ldots\ldots$, so $(2b_8 - b_7 - b_6) = -2$. This case does not occur in Equation 5.5, since $|s_{31}| < 192$ there. However, the modular reduction still works; the result is approximately in the desired interval.

2. $128 \leq s_{31} \leq 191$: then $s_{31} = 0|000000\mathbf{010}\ldots\ldots$, so $(2b_8 - b_7 - b_6) = -1$. So p is subtracted from s once, which is as in Equation 5.5.

3. $64 \leq s_{31} \leq 127$: then $s_{31} = 0|000000\mathbf{001}\ldots\ldots$, so $(2b_8 - b_7 - b_6) = -1$. p is subtracted from s once, which is as in Equation 5.5.

4. $0 \leq s_{31} \leq 63$: then $s_{31} = 0|000000\mathbf{000}\ldots\ldots$, so $(2b_8 - b_7 - b_6) = 0$. 0 is added to s, which is as in Equation 5.5.

5. $-64 \leq s_{31} \leq -1$: then $s_{31} = 1|111111\mathbf{111}\ldots\ldots$, using two's complement notation. (See Definition 5.1.1 for an explanation.) $(2b_8 - b_7 - b_6) = 0$. 0 is added to s, which is as in Equation 5.5.

6. $-128 \leq s_{31} \leq -65$: then $s_{31} = 1|111111\mathbf{110}\ldots\ldots$, so $(2b_8 - b_7 - b_6) = 1$. p is added to s once, which is as in Equation 5.5.

7. $-192 \leq s_{31} \leq -129$: then $s_{31} = 1|111111\mathbf{101}\ldots\ldots$, so $(2b_8 - b_7 - b_6) = 1$. p is added to s once, which is as in Equation 5.5.

8. $-256 \leq s_{31} \leq -193$: then $s_{31} = 1|111111\mathbf{100}\ldots\ldots$, so $(2b_8 - b_7 - b_6) = 2$. This case does not occur in Equation 5.5, since $|s_{31}| < 192$ there. However, the modular reduction still works; the result is approximately in the desired interval.

□

The costs of this method are reasonable; on average 2.02 conditional jumps are replaced by three bitwise ANDs, three bit shifts, two multiplications, and two subtractions. In the reference implementation this increases the computation time by approximately $450\mu s$ per modular reduction. The results in Chapter 6 show that this cost is negligible.

5.1.3 Addition

Adding large numbers cannot be done in a single instruction. Instead the numbers should be added one `byte` at a time. As explained in Section 5.1.1, large numbers are represented as an array of `signed bytes`. Let a and b be large numbers, and let $\hat{a} = [a_{31}|a_{30}|\ldots|a_0]$ and $\hat{b} = [b_{31}|b_{30}|\ldots|b_0]$ be their signed-byte representations. Their sum $\tilde{s}$ is computed as

$$\tilde{s} = [a_{31} + b_{31}|a_{30} + b_{30}|\ldots|a_0 + b_0]. \tag{5.8}$$

The partial sums $a_i + b_i$ need not fit in a `byte` value, but they do fit in a `short` value. To get s in a signed-byte representation, excesses are carried. The carrying is combined with modular

reduction, therefore first $s_{31} = a_{31} + b_{31}$ is computed. From s_{31} we obtain α through Equation 5.7. For the modular reduction αp is added to $\tilde{s}$. $p = 2^{255} - 31 = 64 \cdot 256^{31} - 31$, so first 64α is added to s_{31}. Then -31α is added to s_0. Next, the excess of s_0 is carried to s_1, the excess of s_1 is then carried to s_2, and so forth. The carry from s_{30} to s_{31} is -1, 0, or 1, so the result has $|s_{31} \leq 65|$. Measurements on the Cosmo 5.4 show that adding 0 takes the same amount of time as adding any other value.

Carrying

A carry for an unsigned representation, does not work for a signed-byte representation. A general carry routine is introduced for numbers in signed-byte representation. This routine will also work for multiplication, when carries may be much larger than 1. As an example, a `signed short` ss is converted to two `signed bytes` sb_0 and sb_1. The bit representation of ss is
$\widehat{ss} = sgn \,|\, b_{15}\, b_{14}\, b_{13}\, b_{12}\, b_{11}\, b_{10}\, b_9\, b_8 \,|\, b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0$,
where sgn is the sign bit.

The first case shows an example where an unsigned carry does work. Let $ss = 2925$, then $\widehat{ss} = 0|0001011|01101101$. Take $\widehat{sb_0}$ as the 8 least significant bits of $\widehat{ss}$, and $\widehat{sb_1}$ as the 8 most significant bits of $\widehat{ss}$. sb_0 and sb_1 are both interpreted as `signed byte` values. Then $sb_0 = 109$, $sb_1 = 11$, and $256 \cdot sb_1 + sb_0 = 2925$. So indeed the conversion is successful.

Unsigned carrying fails if and only if $b_7 = 1$. In that case sb_0 will be negative, since b_7 is its sign bit. Let y be the value of $\widehat{sb_0}$ interpreted as an unsigned integer, and x the value of the last 7 bits of $\widehat{sb_0}$ interpreted as an unsigned integer. Then $y = 128 + x$. By Definition 5.1.1 $sb_0 = x - 128$. It follows that $y = sb_0 + 256$. More generally,

$$y = sb_0 + 256 \cdot b_7 \tag{5.9}$$

relates a signed number sb_0 to the unsigned interpretation of its bits, where b_7 is the sign bit of sb_0. For positive numbers this suggests the following carrying procedure: take $\widehat{sb_0}$ as the 8 least significant bits of $\widehat{ss}$, and $\widehat{sb_1}$ as the 8 most significant bits of $\widehat{ss}$. Then add b_7 to sb_1. An example illustrates this procedure when $b_7 = 1$.

Let $ss = 3053$, then $\widehat{ss} = 0|0001011|11101101$. Then $sb_0 = -19$ and $sb_1 = 11 + b_7 = 12$. And indeed $256 \cdot sb_1 + sb_0 = 3053$.

Interestingly, this carry method also works when $ss < 0$. As usual $\widehat{sb_1}$ are the 8 most significant bits of $\widehat{ss}$, and $\widehat{sb_0}$ are the 8 least significant bits of $\widehat{ss}$. sb_0 and sb_1 are both interpreted as `signed byte` values. Define y as $\widehat{sb_0}$, interpreted as an unsigned integer, and let z be the last 7 bits of $\widehat{sb_1}$, interpreted as an unsigned integer. Finally, define u as the value obtained by interpreting the 15 least significant bits of $\widehat{ss}$ as an unsigned integer. As a graphic:

$$\widehat{ss} = \underbrace{1\overbrace{0001011}^{z}}_{sb_1}\underbrace{1\overbrace{1101101}^{x}}_{sb_0}. \qquad \overbrace{\quad\quad\quad\quad\quad\quad\quad}^{u},\ \overbrace{\quad\quad\quad\quad\quad\quad}^{y} \tag{5.10}$$

The values above the graphic are `unsigned` numbers, the values below are `signed` numbers.

By Equation 5.9 and Definition 5.1.1, $ss = u - 2^{15} \cdot b_{15}$, $sb_1 = z - 128 \cdot b_{15}$, and $y = sb_0 + 256 \cdot b_7$. Also, $u = y + 256 \cdot z$. Substitution gives $ss = y + 256 \cdot z - 2^{15} \cdot b_{15} = sb_0 + 256 \cdot b_7 + 256 \cdot (sb_1 + 128 \cdot b_{15}) - 2^{15} \cdot b_{15} = sb_0 + 256 \cdot (sb_1 + b_7)$.

5.1.4 Subtraction

Subtraction is identical to addition, except that the addition of two words is replaced by the subtraction. Due to the signed-byte representation of the numbers, the output range of a subtraction is almost the same as that of an addition. Carrying and modular reduction is done in exactly the same way for subtraction as for addition.

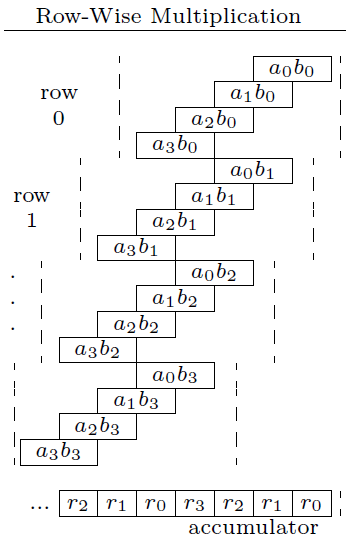


Figure 5.1: The row-wise multiplication algorithm for two long integers a and b of 4 words each. The accumulator uses 4 registers. This figure was copied from [34] with the permission of Hans Eberle, one of the authors.

5.1.5 Multiplication

Like adding large numbers, multiplying large numbers cannot be done in a single instruction. A naive approach for multiplying large numbers is given in Algorithm 7. This method is essentially the multiplication method taught in primary school. It will be called *row-wise multiplication* here.

Algorithm 7 Row-wise multiplication:

Input: a, b
Output: $result = a * b$

```
result ← 0
l ← length(a)
m ← length(b)
for j = 0 to m − 1 do
  for i = 0 to l − 1 do
    result[i + j] ← result[i + j] + a[i] ∗ b[j]
    carry from result[i + j] to result[i + j + 1]
  end for
end for
```

Figure 5.1 gives an example for $l = m = 4$, adopted from [34]. In the rest of this thesis, it is assumed that $l = m$ unless stated otherwise.

[34] analyses the row-wise multiplication algorithm, and also the column-wise variant. For row-wise multiplication, the index of one of the multiplicands is constant throughout each inner loop. In Algorithm 7 this is the value $b[j]$. So in an implementation, the value $b[j]$ is usually stored locally throughout the execution of one inner loop.

Column-wise multiplication is similar to row-wise multiplication, described in Algorithm 7. But for column-wise multiplication the index of $result[]$ is kept constant in the inner loop, instead of the index of $b[]$. Row-wise and column-wise multiplication are the leftmost algorithms in Figure 5.2. This figure is adopted from [34], and has $l = m = 4$.

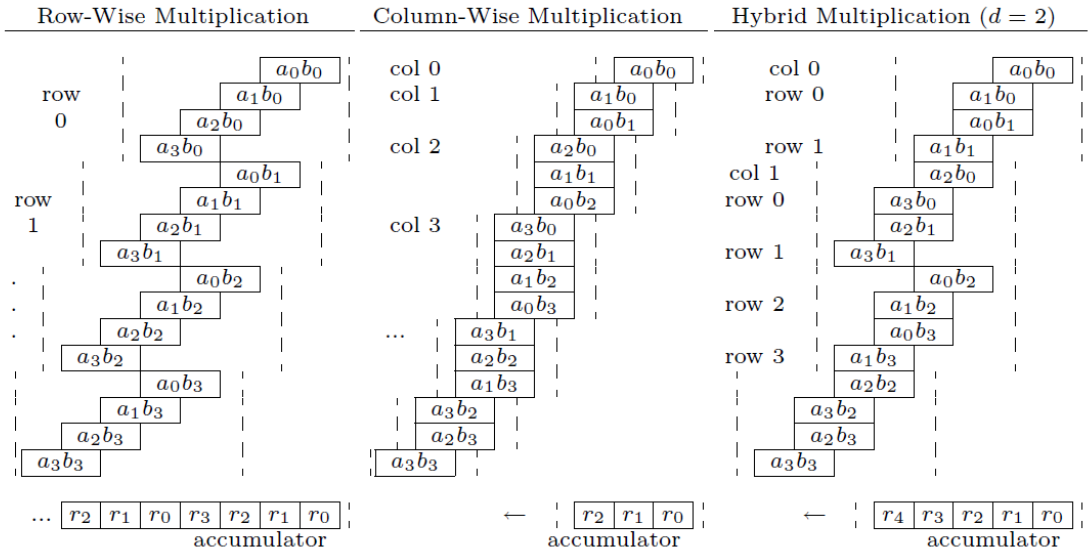


Figure 5.2: Row-wise, column-wise, and hybrid multiplication methods. In all cases the long integers a and b have 4 words each. For the hybrid multiplication $d = 2$. The accumulator uses 4, 3, or 5 registers respectively. This figure was copied from [34] with the permission of Hans Eberle, one of the authors.

Section 4.5.2 shows that the bottleneck for most algorithms will be the number of memory operations. For long multiplication this number is typically $O(n^2)$. [34] introduces a new multiplication algorithm, called *hybrid multiplication*. This algorithm reduces the number of memory operations. It is suitable for restricted platforms, and is introduced below.

Hybrid multiplication

Hybrid multiplication is a family of multiplication methods, ranging from row-wise to column-wise multiplication. The variant is specified by a parameter $d \in \{1, \ldots, l\}$, where $l = m$ is the number of words in the operands $a[]$ and $b[]$. d specifies the column width used in the algorithm. In Figure 5.2 the column-wise multiplication methods uses a column width of $d = 1$. The index of the result is kept constant throughout a column. For example, in "col 3" $i + j = 3$ for all partial products $a_i b_j$. For this method the accumulator always uses three registers. The row-wise method uses a column width of $l = m = 4$. Throughout a row, the index j of b_j is kept constant. The accumulator uses $l = m = 4$ registers.

In Figure 5.2 the hybrid multiplication method uses a column width of $d = 2$. For each column c, the sum of the indices $i + j$ is within the range $\{dc, \ldots, d(c + 1)\}$. For example, for column 1, $c = 1$, so $i + j$ is in the range $\{2, 3, 4\}$. Within each column, rows are distinguished. Within a row, j is kept constant. Row 0 of column 1 has $b[j] = b_0$, so $j = 0$.

The accumulator uses $2d + 1$ registers, and additional registers are required for the operand values. In column 1 of the hybrid multiplication in Figure 5.2 the values b_0, b_1, b_2, and b_3 are loaded in registers. These operands are all used d times, saving memory loads. The values a_i are loaded each time. By reusing registers, this requires $d + 1$ additional registers for the operand values.

Table 5.1 shows that hybrid multiplication can significantly reduce the number of memory operations. In the reference implementation $d = 4$ was used, reducing the time required for a multiplication by 40%. More computation time may be saved at the cost of a larger code size. On

Method	Row-wise	Column-wise	Hybrid
Accumulator registers	m	3	$2d+1$
Operand registers	2	2	$d+1$
Memory loads	m^2+m	$2m^2$	$2\lceil\frac{m^2}{d}\rceil$
Memory stores	$2m$	$2m$	$2m$
Total registers	$m+2$	5	$3d+2$
Total memory operations	m^2+3m	$2m^2+2m$	$2\lceil\frac{m^2}{d}\rceil+2m$

Table 5.1: A comparison of three multiplication methods. The table shows the number of registers and memory operations required for the multiplication of two long integers of m words each.

the Cosmo 5.4, up to 128 objects may be declared. 48 are already in use, so 80 are available for improving multiplication. However, local variables in Java Card are not indexable, which means that the code size grows quadratically in d. To illustrate this, Algorithm 8 shows pseudo-code of the inner loop of a hybrid multiplication method with $d = 4$, implemented in Java Card.

For $d = 4$ it costs 638 bytes to change the multiplication method, and an additional 1093 bytes to change the squaring method in a similar way.

Modular reduction

When two large numbers a and b of 32 words each are multiplied, the result $c = a \cdot b$ has at most 64 words. The 32 most significant words of c are called c_{hi}. The 32 least significant words are called c_{lo}. This means that $c = c_{lo} + 2^{256}\, c_{hi}$. Reducing c modulo $p = 2^{255} - 31$ is very easy. $2^{256} \equiv 2 \cdot 2^{255} \equiv 2 \cdot 31 \equiv 62 \pmod{p}$. Therefore, $c \equiv c_{lo} + 62\, c_{hi} \pmod{p}$. Schematically the modular reduction looks like this:

$$\begin{array}{cccccccc} 62\,c_{63} & 62\,c_{62} & 62\,c_{61} & \ldots & 62\,c_{34} & 62\,c_{33} & 62\,c_{32} & \\ c_{31} & c_{30} & c_{29} & \ldots & c_2 & c_1 & c_0 & + \\ \hline r_{31} & r_{30} & r_{29} & \ldots & r_2 & r_1 & r_0 & \end{array} \tag{5.11}$$

Of course, the sums $62c_{i+32}+c_i$ will not always fit in a `signed byte`. The excess values are carried from r_i to r_{i+1}. The first carry is from r_{30} to r_{31}. This carry is done first to prevent a large carry from r_{30} to r_{31} later. Such a large carry could cause r_{31} to overflow, which means another cycle of carries would be needed. The second carry is from r_{31} to r_0. If this leaves $r_{31} < -64$ then p is added to the result. If $r_{31} \geq 64$ then p is subtracted from the result. This works in the same way as the modular reduction in Section 5.1.2; 128 is added to or subtracted from r_{31}, and -31 is added to or subtracted from r_0 accordingly. When carrying from r_{31} to r_0 the carry is reduced modulo $2^{255} - 31$ by multiplying it with 62. Next, carrying continues from r_0 to r_1, from r_1 to r_2, etcetera, and finally from r_{30} to r_{31}. The last carry from r_{30} to r_{31} is -1, 0, or 1. This means that the result has $-65 \leq r_{31} \leq 64$.

Karatsuba multiplication

The first multiplication algorithm faster than $O(n^2)$ was introduced by Karatsuba and Ofman in 1963 [39]. Asymptotically, its complexity is $O(n^{\log_2 3}) \approx O(n^{1.585})$. The algorithm was credited to Karatsuba, and is often referred to as *Karatsuba multiplication*. Karatsuba multiplication is explained in this section. Toom generalised Karatsuba's algorithm, achieving a better asymptotic complexity [66]. Today, multiplication can be done in almost linear time [60], [31], but this complexity is not achieved for numbers below 100,000 bits [32].

When multiplying two large numbers x and y, a speedup may be achieved by splitting these numbers into their most and least significant halves. This yields $x = (x_{hi}|x_{lo})$ and $y = (y_{hi}|y_{lo})$. Taking the radix $R = b^{\lceil \frac{n}{2} \rceil}$ gives $x = x_{hi}R + x_{lo}$ and $y = y_{hi}R + y_{lo}$, where n is the bit length of x and y. The product is $xy = x_{hi}y_{hi}R^2 + (x_{hi}y_{lo} + x_{lo}y_{hi})R + x_{lo}y_{lo}$. This replaces a length n multiplication with 4 length $\lceil \frac{n}{2} \rceil$ multiplications. For an $O(n^2)$ algorithm this yields no improvement.

Algorithm 8 Inner loop of hybrid multiplication:

$b0 \leftarrow b[4 * j]$
$b1 \leftarrow b[4 * j + 1]$
$b2 \leftarrow b[4 * j + 2]$
$b3 \leftarrow b[4 * j + 3]$

$a0 \leftarrow a[4 * (i - j)]$

$r0 \leftarrow r0 + a0 * b0$
carry from $r0$ to $r1$
$r1 \leftarrow r1 + a0 * b1$
carry from $r1$ to $r2$
$r2 \leftarrow r2 + a0 * b2$
carry from $r2$ to $r3$
$r3 \leftarrow r3 + a0 * b3$
carry from $r3$ to $r4$

$a0 \leftarrow a[4 * (i - j) + 1]$

$r1 \leftarrow r1 + a0 * b0$
carry from $r1$ to $r2$
$r2 \leftarrow r2 + a0 * b1$
carry from $r2$ to $r3$
$r3 \leftarrow r3 + a0 * b2$
carry from $r3$ to $r4$
$r4 \leftarrow r4 + a0 * b3$
carry from $r4$ to $r5$

$a0 \leftarrow a[4 * (i - j) + 2]$

$r2 \leftarrow r2 + a0 * b0$
carry from $r2$ to $r3$
$r3 \leftarrow r3 + a0 * b1$
carry from $r3$ to $r4$
$r4 \leftarrow r4 + a0 * b2$
carry from $r4$ to $r5$
$r5 \leftarrow r5 + a0 * b3$
carry from $r5$ to $r6$

$a0 \leftarrow a[4 * i - j + 3]$

$r3 \leftarrow r3 + a0 * b0$
carry from $r3$ to $r4$
$r4 \leftarrow r4 + a0 * b1$
carry from $r4$ to $r5$
$r5 \leftarrow r5 + a0 * b2$
carry from $r5$ to $r6$
$r6 \leftarrow r6 + a0 * b3$
carry from $r6$ to $r7$

Karatsuba found that the mixed term $x_{hi}y_{lo} + x_{lo}y_{hi}$ may be computed as $(x_{lo} + x_{hi})(y_{lo} + y_{hi}) - x_{hi}y_{hi} - x_{lo}y_{lo}$. In total, this requires only three length $\lceil \frac{n}{2} \rceil$ multiplications, some additions, some subtractions, and some bit shifts. Since the additions, subtractions, and bit shifts can be done in linear time, this reduces the complexity of multiplication. The algorithm may be used recursively until single-word multiplication is reached.

For 256-bit multiplications in Java Card, Karatsuba multiplication does not provide a significant speedup. The reason is that the execution time is dominated by the time it takes to read and write from and to arrays. Also, an elementary addition of two `bytes` is no faster than a multiplication of two `bytes`. Table 4.1 shows that `byte` multiplication is slightly slower than addition, and that a read or write in CLEAR_ON_RESET RAM is about 8 times as slow as a multiplication. While Karatsuba multiplication reduces the number of elementary multiplications, it actually increases the number of memory accesses.

As an example, consider the multiplication of two 256-bit numbers x and y. Both are represented as arrays of 32 `bytes` each. Basic column-wise multiplication takes $32^2 = 1024$ memory reads and $2 \cdot 32 = 64$ memory writes. This method requires $32^2 = 1024$ `byte` multiplications. In total, 1088 memory operations are needed. When Karatsuba multiplication is used once, the multiplications $x_{hi}y_{hi}$ and $x_{lo}y_{lo}$ operate on arrays of 16 bytes. The multiplication $(x_{lo}+x_{hi})(y_{lo}+y_{hi})$ in general operates on arrays of 17 bytes, since the sums $x_{lo} + x_{hi}$ and $y_{lo} + y_{hi}$ may not fit in 16 bytes. To prevent the revelation of side channel information, 17 bytes should always be used. Hence the three multiplications require $2 \cdot 16^2 + 17^2 = 801$ memory reads, $2 \cdot 32 + 33 = 97$ memory writes, and $2 \cdot 16^2 + 17^2 = 801$ multiplications. The additions $x_{lo} + x_{hi}$ and $y_{lo} + y_{hi}$ require $2 \cdot 2 \cdot 16 = 64$ memory reads and $2 \cdot 17 = 34$ memory writes. The double subtraction $(x_{lo}+x_{hi})(y_{lo}+y_{hi}) - x_{hi}y_{hi} - x_{lo}y_{lo}$ takes $33 + 2 \cdot 32 = 97$ memory reads and 33 memory writes. Since the parts $x_{hi}y_{hi}R^2$, $(x_{hi}y_{lo} + x_{lo}y_{hi})R$, and $x_{lo}y_{lo}$ overlap, one more addition is required, using $2 \cdot 33 = 66$ memory reads and 33 memory writes. By combining operations, up to 67 memory writes may be saved. The total number of memory operations then is 1094, which is more than the 1088 needed for basic column-wise multiplication. Since the total number of additions and multiplications is lower for Karatsuba multiplication, there is an overall speedup expected of about 2 ms for 256-bit multiplication. This is less than 0.3%, and therefore Karatsuba multiplication was not considered worth the extra code complexity.

5.1.6 Squaring

The squaring algorithm is similar to the multiplication algorithm. Products of words are added in the accumulator. A hybrid multiplication method with $d = 4$ is used. Modular reduction is exactly the same as after multiplication. But, squaring can be done more efficiently than multiplication. Notice in the following multiplication example that some products of words occur more than once.

$$\begin{array}{cccccc} & & a_2 & a_1 & a_0 & \\ & & a_2 & a_1 & a_0 & \times \\ \hline & & a_0a_2 & a_0a_1 & a_0a_0 & \\ & a_1a_2 & a_1a_1 & a_1a_0 & & \\ a_2a_2 & a_2a_1 & a_2a_0 & & & \end{array} \tag{5.12}$$

For example, in the middle column the product a_0a_2 occurs twice. Every product a_ia_j with $i \neq j$ will occur exactly twice in the same column. Products of the form a_ia_i occur only once. Computations may be saved by computing the products of the form a_ia_j with $i \neq j$ only once. In the reference implementation only those products are computed for which $i \geq j$. The sum of column k is then multiplied by 2. If k is even then the column has a product of the form $a_{k/2}a_{k/2}$. This product is then subtracted, since it was counted twice instead of once.

When a consists of m words, a normal multiplication method computes m^2 products of words. The squaring method only computes each unique product once. These are $\binom{m}{2} = \frac{m(m-1)}{2}$ products.

[18] mentions a trick to make the squaring slightly faster: instead of multiplying the column sums by 2, the values $2a_0, 2a_1, \ldots, 2a_m$ are precomputed. The products $a_i \cdot a_j$ are then replaced by

the products $2a_i \cdot a_j$. So the column sums need not be multiplied by 2 anymore. This reduces the number of doublings from $2m$ to m. I was made aware of this trick after finishing measurements with the reference implementation.

5.1.7 Inversion

Inversion is the most time-consuming finite field operation in elliptic-curve-based protocols. In the reference implementation, no inversions are needed. However, in many related protocols, such as ECDSA and EC-KCDSA, inversions are needed. When inversions can be done very efficiently, elliptic curve computations can sometimes be done in a faster way [20]. Therefore, I tested inversions to see how fast they can be done. This allows me to make a fair comparison between common signature schemes, and the scheme presented in Chapter 3. Two inversion methods were tested.

Euclid's algorithm

The first method inversion uses the Binary Extended Euclidean Algorithm with blinding. A description of the Binary Extended Euclidean Algorithm can be found in [43]. This algorithm computes an inverse modulo p in $O(m^2 \log m)$ time, where m is the bit length of p, but it heavily relies on branching in the code. Therefore, it is possible to determine the number being inverted by simple power analysis. This is prevented with a blinding technique. The number x is inverted modulo p as follows: generate a random number $r \in \mathbb{F}_p^*$. Compute $rx \pmod p$, and invert the result. This gives $x^{-1}r^{-1} \pmod p$. Multiplying with r gives $x^{-1} \pmod p$. Since rx is uniformly random in $\mathbb{F}_p^*$ the power trace gives no information about x. The costs of blinding are $2\mathbf{M}$ plus the generation of a random number in $\mathbb{F}_p^*$.

The blinded version of this algorithm is patented by US patent 2008/0201398, and therefore not free to use. For reference, the algorithm was tested.

Inversion by exponentiation

US patent 2008/0201398 covers all inversion methods that use blinding combined with an efficient inversion algorithm. Fermat's little theorem gives an inversion method that is not covered by this patent.

Theorem 5.1.2 (Fermat's little theorem)**.** *Let p be a prime number and $x \in \mathbb{Z}$. Then*

$$x^p \equiv x \pmod p. \tag{5.13}$$

Proof. There are many easy proofs, for example, see [69]. □

Since $\mathbb{F}_p$ is a field, it follows that

$$x^{p-2} \equiv x^{-1} \pmod p. \tag{5.14}$$

So $x^{-1} \pmod p$ may be computed as $x^{p-2} \pmod p$. Unfortunately, this computation is not as efficient as Euclid's algorithm. When multiplication is estimated as $O(m^2)$, the complexity is $O(m^3)$, where m is the bit length of p. Even with faster multiplication algorithms, the complexity is not as low as the $O(m^2)$ of the Euclidean Algorithm. For $p = 2^{255} - 31$ the straightforward exponentiation costs $254\mathbf{S} + 10\mathbf{M}$. This exponentiation first computes x^{31} and then uses repeated squarings, alternated with occasional multiplications. One can find the shortest multiplication chain with some effort, but with $\mathbf{S} \approx 0.63\mathbf{M}$ the straightforward exponentiation is probably the fastest way of computing $x^{2^{255}-33}$.

5.2 Using the RSA function

Java Card applets are executed by the JCVM, which usually runs on the main processor of the smart card. Java Card applets have no control over the cryptographic coprocessor. However, Java Card has an interface that allows the execution of standard built-in cryptographic functions, such as RSA, DSA, ECC, ECDSA, 3DES, AES, and SHA-1. When used in the right way, these function can be used to compute finite field squarings and multiplications.

5.2.1 Squaring with the RSA function

According to the specifications the Cosmo 5.4 has a built-in RSA function with modulus sizes ranging from 256 to 2048 bits, by steps of 32 bits [2]. Contrary to these specifications, it turned out that 256- and 512-bit RSA are not supported by the smart card, so 1024-bit RSA was used instead.

Let $x < p$ be a 256-bit unsigned integer to be squared modulo the prime number p. Given is a 1024-bit RSA function. Select two prime numbers $q_1, q_2 > p^2$ of 512 bits, and set $N = q_1 q_2$. Next, set the public exponent $e = 2$, if allowed. In the Cosmo 5.4 this is not allowed, so $e = \phi(N) + 2$ is used instead, where ϕ is the Euler-phi function. If $q_1 \neq q_2$ then $\phi(N) = (q_1 - 1)(q_2 - 1)$. Note that there is no d satisfying $de \equiv 1 \pmod{\phi(N)}$, since $\phi(N)$ and e are both even. If the RSA function tries to compute d, it will raise an error since $\gcd(e, \phi(N)) \neq 1$. Luckily, the RSA function only computes d if instructed to do so. Also, the RSA function does not check whether e is odd. With $e = \phi(N) + 2$ the public key operation computes

$$x^{\phi(N)+2} \pmod{N} \equiv x^2 \pmod{N}, \tag{5.15}$$

since $\gcd(x, N) = 1$. To obtain $x^2 \pmod{p}$, the result of Equation 5.15 is reduced modulo p. A 256-bit RSA function where N is allowed to be prime would be convenient. With $N = p$ the RSA function would then automatically reduce the results modulo p.

5.2.2 Multiplying with the RSA function

From a squaring algorithm, it is easy to derive a multiplication algorithm. Note that

$$ab = \frac{1}{2}((a+b)^2 - a^2 - b^2), \tag{5.16}$$

and

$$ab = \frac{1}{4}((a+b)^2 - (a-b)^2). \tag{5.17}$$

Equation 5.16 costs 3 squarings, 1 addition, 2 subtractions, and a division by 2. Equation 5.17 costs 2 squarings, 1 addition, 2 subtractions, and a division by 4. In both cases the division is just a bit shift, so both divisions will be equally expensive. But the subtractions are not equally expensive. When using Equation 5.16 the subtractions always subtract a small number from a larger number, which is the easiest case for a subtraction. When Equation 5.17 is used, the subtraction $(a - b)$ may be such that $b > a$. This expression may be replaced by $(b - a)$ in the equation, but this requires comparing a and b, costing 64 memory accesses if side channel information is not to be revealed. Computing $(a-b)$ with $b > a$ requires a reduction modulo p, which also costs 64 memory accesses.

In the Cosmo 5.4, squaring with the RSA function takes time equivalent to 2.4 squarings in software, see Section 6.1. This includes a reduction modulo p. 2.4**S** is very fast, considering that actually a 1024-bit exponentiation is computed. The reason is that RSA uses the coprocessor, which is optimised for this operation, and has access to specifically designed cryptographic hardware. For 256-bit multiplications and squarings, the RSA function does not save time. For 512-bit multiplications and squarings, the RSA function is expected to save time.

5.2.3 Inversion with the RSA function

Section 5.1.7 shows that inversion is usually a very costly operation. Therefore the RSA function provides a great speedup for inversion. The RSA function is used in the same way as in Section 5.2.1, but now with exponent $e = p - 2$ and modulus $N = pq$. Here, q is a 768-bit prime number, so that N has 1024 bits. For an input x, the RSA function computes $x^{p-2} \pmod N$. Reducing this number modulo p yields

$$(x^{p-2} \pmod N) \pmod p \equiv (x^{p-2} \pmod{pq}) \pmod p \equiv x^{-1} \pmod p. \tag{5.18}$$

5.2.4 Efficiency of using the RSA function

While the Cosmo 5.4 RSA implementation is quite fast, the overhead of using it may be too much of a burden. Section 5.2.2 describes overhead that is usually necessary when a squaring function is used to compute multiplications. In the reference implementation the following problems also occurred:

- The RSA function does not support $e = 2$, so the slower $e = \phi(N) + 2$ was used.
- The RSA function returns numbers reduced modulo N, instead of modulo p. N is four times long as p.
- Java Card always uses signed number representations. However, the RSA function uses an unsigned representation. Therefore, a conversion is needed for every byte that is used in the RSA function.

5.3 Elliptic curve arithmetic

The elliptic curve arithmetic is different for signing and verification. The implementation of signing is given in Section 5.3.1. The implementation of verification is given in Section 5.3.2. For a fast implementation, a Twisted Edwards curve over $\mathbb{F}_p$ was used with $p = 2^{255} - 31$, $a = -1$, and $d = 92$. This curve has $4n$ points, where $n = 2^{253} + 7397257049794027223001250076940419149 3$ is prime. The choice $a = -1$ makes sure the fast addition and doubling formulas from Section 2.9 can be used. The choice $p = 2^{255} - 31$ was made from several possible candidates of the form $2^{255} + \gamma$ or $2^{256} + \gamma$. The number γ was chosen to fit in a `signed byte`, so $-128 \leq \gamma \leq 127$. From these candidates the curves with cofactor 4 were selected, and from these curves the one with the smallest $k = 2d$ was used. No further restrictions were imposed on the bit length of n, as long as the curve had $4n$ points with n a prime number.

5.3.1 Signing

The elliptic curve operations use extended coordinates $(X : Y : Z : T)$. This notation is explained in Section 2.9. A point-scalar multiplication on an elliptic curve is computed with a fixed window method. The sequence of two point doublings and one point addition is repeated. The points G, $2G$, $3G$, up to $16G$ are precomputed and randomised. A point is randomised by multiplying its four coordinates with the same random number. The neutral element of the point addition $0G = \mathcal{O} = (0 : 1 : 1 : 0)$ is avoided, because the zero coordinates do not change when randomised. These zero coordinates may be the target of side channel analysis. This idea was introduced by Goubin in [33] and generalised by Akishita and Takagi in [11].

The scalar r consists 64 randomly generated values in the range $\{1, 2, \ldots, 16\}$. These values $r_0, r_1, r_2, \ldots, r_{63}$ represent the number $r_0 + r_1 \cdot 16 + r_2 \cdot 16^2 + \cdots + r_{63} \cdot 16^{63}$. The possible range for r is $\lceil \frac{16}{15} 2^{252} \rceil \leq r \leq \lceil \frac{256}{15} 2^{252} \rceil$. Only the values $n + 1 \leq r \leq 8n - 1$ are used. If $r \equiv 0 \pmod n$ the generation of r is redone. This ensures that $r \pmod n$ is uniformly random in the range $\{1, \ldots, n - 1\}$ as required. The idea of replacing the coefficient range $\{0, 1, \ldots, 2^w - 1\}$ by

$\{1, 2, \ldots, 2^w\}$ was described by Möller for signed representations in [50]. Möller's technique is not entirely secure, because the expansion depends on the value of r. In the reference implementation this is avoided by generating $r_0, \ldots, r_{63}$ at random. The combination of avoiding 0 as a coefficient by generating the coefficients at random in $\{1, 2, \ldots, 2^w\}$ was not found in the literature. Note that the neutral element $\mathcal{O} = 0G = nG$ cannot occur during the computation. The computation starts with $r_{63}G \neq 0G$. This point is doubled 4 times, $r_{62}G$ is added, and so on. So $\mathcal{O}$ can only occur as lnG, where $l \in \{1, 2, \ldots\}$. Since $n > \lceil \frac{16}{15} 2^{252} \rceil$ this means that $\mathcal{O}$ can only occur at the end of the computation. The intermediate value $(r - r_0)G$ is computed by doubling $(\frac{r-r_0}{16})G$ four times. The number $\frac{r-r_0}{16}$ is smaller than n, so $\frac{r-r_0}{16} \not\equiv 0 \pmod{n}$. Since n is prime, the numbers $\frac{r-r_0}{8}$, $\frac{r-r_0}{4}$, $\frac{r-r_0}{2}$, and $r - r_0$ are not divisible by n either. Finally, $rG \neq \mathcal{O}$ because $r \not\equiv 0 \pmod{n}$. So $\mathcal{O}$ does not occur during the computation of rG. When $r \pmod{n}$ has been computed, the values $r_0, \ldots, r_{63}$ are converted as follows:

$$r'_j = 0000000000000001 \ll (j - 1),\ j = 1, 2, \ldots, 63, \tag{5.19}$$

where "$\ll l$" is a left shift over l positions. The use for this conversion is explained below.

The precomputed points G, $2G$, up to $16G$ are stored in EEPROM. When a precomputed point is loaded, the power trace or an electromagnetic radiation analysis may reveal which of these points is loaded. Therefore, all points are loaded to RAM, and a masking technique is used. A common masking technique uses manipulation of pointers, but unfortunately Java Card does not support pointers. An obvious alternative manipulates array indices, but Appendix B shows that the reading time from an array is dependent on the index. Hence, a power trace would still reveal which point is loaded. Therefore, an algebraic masking method is used. Suppose the point $P = 2G$ is needed for a computation. Then the indicator i_2 is set to 1, and the other indicators $i_1, i_3, i_4, \ldots, i_{16}$ are set to 0. P is computed as

$$P = (0 \cdot G) \boxplus (1 \cdot (2G)) \boxplus (0 \cdot (3G)) \boxplus \cdots \boxplus (0 \cdot (16G)), \tag{5.20}$$

where $b \cdot P = (bP_X : bP_Y : bP_Z : bP_T)$, and $P \boxplus Q = (P_X + Q_X : P_Y + Q_Y : P_Z + Q_Z : P_T + Q_T)$. The indicators $i_1, \ldots, i_{16}$ in each step j are computed from the r'_j of Equation 5.19:

$$i_k = (r'_j \gg (k - 1)) \ \&\ 0000000000000001,\ k = 1, 2, \ldots, 16, \tag{5.21}$$

where "$\ll l$" is a left shift over l positions, and "&" is a bitwise AND.

The point-scalar multiplication rG is computed as follows. Start by setting $P = r_{63}G$. Then double P 4 times, and add $r_{62}G$ to P. Repeat this procedure until r_0 is reached. The point r_jG for $j = 63, 62, \ldots, 0$ is always loaded from the precomputed points. Adding and doubling points is done with the formulas from Section 2.9. However, some operations are saved. The point doubling and addition formulas are recalled here as Algorithm 9 and 10.

Algorithm 9 does not need the T-coordinate as an input. Therefore, the computation of T is skipped for point addition, and for three of the four point doublings. This saves 1**M** for each of these operations. Algorithm 10 has two points as input, one of which is precomputed. The precomputed point P' is not represented as $(X : Y : Z : T)$, but as $(Y - X : Y + X : 2Z : kT)$. This saves 2 **add**, 1$*$**2**, and 1**k** in Algorithm 10, where "**add**" is a field addition, "$*$**2**" is a multiplication with 2, and "**k**" is a multiplication with $k = 2d$. The idea of precomputing $(Y - X : Y + X : 2Z : kT)$ seems to be new, since it does not appear in the Explicit-Formulas Database (EFD) [21]. This idea not only saves computation time, but may also prevent side channel attacks, as explained in Section 5.1.2 below 'Avoiding a conditional jump'. One step in the point-scalar multiplication looks like:

ADD $(X : Y : Z : T), (Y - X : Y + X : 2Z : kT) \rightarrow (X : Y : Z)$
DBL $(X : Y : Z) \rightarrow (X : Y : Z)$
DBL $(X : Y : Z) \rightarrow (X : Y : Z)$
DBL $(X : Y : Z) \rightarrow (X : Y : Z)$
DBL $(X : Y : Z) \rightarrow (X : Y : Z : T)$

Algorithm 9 Point doubling on a twisted Edwards curve with $a = -1$:

Input: A point $P = (X1 : Y1 : Z1)$ on the curve
Output: The point $2P = (X3 : Y3 : Z3 : T3)$

$A \leftarrow X1^2$
$B \leftarrow Y1^2$
$C \leftarrow 2 * Z1^2$
$D \leftarrow a * A$
$E \leftarrow (X1 + Y1)^2 - A - B$
$G \leftarrow D + B$
$F \leftarrow G - C$
$H \leftarrow D - B$
$X3 \leftarrow E * F$
$Y3 \leftarrow G * H$
$T3 \leftarrow E * H$
$Z3 \leftarrow F * G$

Algorithm 10 Point addition on a twisted Edwards curve with $a = -1$, where $k = 2d$:

Input: Two points $P1 = (X1 : Y1 : Z1 : T1)$, and $P2 = (X2 : Y2 : Z2 : T2)$ on the curve
Output: The point $P1 + P2 = P3 = (X3 : Y3 : Z3 : T3)$

$A \leftarrow (Y1 - X1) * (Y2 - X2)$
$B \leftarrow (Y1 + X1) * (Y2 + X2)$
$C \leftarrow T1 * k * T2$
$D \leftarrow Z1 * 2 * Z2$
$E \leftarrow B - A$
$F \leftarrow D - C$
$G \leftarrow D + C$
$H \leftarrow B + A$
$X3 \leftarrow E * F$
$Y3 \leftarrow G * H$
$T3 \leftarrow E * H$
$Z3 \leftarrow F * G$

The point addition costs $7\mathbf{M} + 6\text{add}$, the doubling DBL $(X : Y : Z) \to (X : Y : Z)$ costs $3\mathbf{M} + 4\mathbf{S} + 6\text{add} + 1\mathbf{a} + 1{*}\mathbf{2}$, and the doubling DBL $(X : Y : Z) \to (X : Y : Z : T)$ costs $4\mathbf{M} + 4\mathbf{S} + 6\text{add} + 1\mathbf{a} + 1{*}\mathbf{2}$. The point-scalar multiplication costs 64 ADD, 192 DBL $(X : Y : Z) \to (X : Y : Z)$, and 64 DBL $(X : Y : Z) \to (X : Y : Z : T)$. This amounts to $1280\mathbf{M} + 1024\mathbf{S}$ when only multiplications and squarings are counted.

The computation above ignores the costs for randomising, masking, and precomputing. Precomputing actually needs to be done only once for a base point G, but to save EEPROM the precomputation is done for each signing operation. The costs of precomputing are 7 ADD and 8 DBL, where in both cases the T-coordinates is computed. This costs $88\mathbf{M} + 32\mathbf{S}$. Randomising 16 points costs $64\mathbf{M}$. The costs of masking are mainly the loading from EEPROM. Differential fault attacks are prevented by checking whether the result is a point on the curve, i.e. $Z^2(aX^2 + Y^2) \stackrel{?}{=} Z^4 + dX^2Y^2$. This costs an additional $2\mathbf{M} + 4\mathbf{S}$. Since three coordinates X, Y, and Z are used for this check, a fault attack as described in [29] has a negligible probability of being successful.

5.3.2 Verification

Verifying that R is on $\mathcal{E}$ costs $2\mathbf{M} + 4\mathbf{S}$. If the T-coordinate was not transmitted, R is computed as $(XZ : YZ : Z^2 : XY)$ as suggested in [36], costing $3\mathbf{M} + 1\mathbf{S}$. From there, verifying that R is on $\mathcal{E}$ only costs another $1\mathbf{M} + 3\mathbf{S}$, ignoring minor operations.

No secret values are used in verification. Therefore, randomisation and masking are no longer needed. The elliptic curve operations use extended coordinates $(X : Y : Z : T)$, as in Section 5.3.1. The point addition and point doubling algorithms from Section 5.3.1 are used. 12 points are precomputed, which is four less than for signing. So no additional EEPROM is needed. The precomputed points, represented as $(Y - X : Y + X : 2Z : kT)$, are

$$\begin{array}{cccc} & G & & 3G \\ R & R+G & R+2G & R+3G \\ & 2R+G & & 2R+3G \\ 3R & 3R+G & 3R+2G & 3R+3G \end{array}$$

For the verification the double point-scalar multiplication $sG + hR$ is computed. For this computation the binary representations of s and h are used. One step in the point-scalar multiplication consists of 1 addition and 2 doublings, unless s_j and h_j are both 0. In that case a doubling is done, and s and h are shifted by one bit. The following example shows a part of the computation (from right to left), where s and h are represented in binary:

s :	...	11	01	0	10	11	...
h :	...	00	01	0	01	01	...
		ADD $3G$	ADD $G + R$	DBL$(XYZT)$	ADD $2G + R$	ADD $3G + R$	
		DBL(XYZ)	DBL(XYZ)		DBL(XYZ)	DBL(XYZ)	
		DBL$(\dots)$	DBL$(XYZT)$		DBL(XYZ)	DBL$(XYZT)$	

When an operation is followed by a doubling, the coordinate T is not computed. This requires the algorithm to look one bit ahead.

To compute the expected verification time, two states of the algorithm are identified. The first state S_0 is when s_j and h_j are both 0. This state occurs on average $\frac{1}{4}$ of the time. The second state S_1 is when s_j and h_j are not both 0, which occurs on average $\frac{3}{4}$ of the time. Figure 5.3 shows these states with their transition probabilities. Table 5.2 gives the costs of each transition. So the expected costs of a 256-bit double point-scalar multiplication are

$$\begin{aligned} &256\big[\ \tfrac{1}{4}\tfrac{1}{4}(3\mathbf{M} + 4\mathbf{S}) + \tfrac{1}{4}\tfrac{3}{4}(4\mathbf{M} + 4\mathbf{S}) + \tfrac{3}{4}\tfrac{1}{4}\left(\tfrac{1}{2}(13\mathbf{M} + 8\mathbf{S})\right) + \tfrac{3}{4}\tfrac{3}{4}\left(\tfrac{1}{2}(14\mathbf{M} + 8\mathbf{S})\right)\ \big] \\ &= 1560\mathbf{M} + 1024\mathbf{S}. \end{aligned}$$

Precomputing costs 2 point doublings and 10 point additions, which amounts to $88\mathbf{M} + 8\mathbf{S}$. Using precomputed points speeds up the verification method. Since there is room for 16 precomputed points instead of 12, some speed may be gained by using signed fractional windows [50].

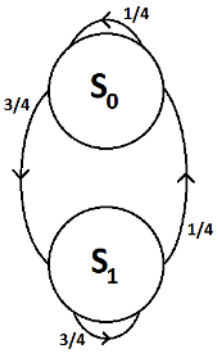


Figure 5.3: Transitions between S_0 (both exponents are 0) and S_1 (at least one exponent is 1).

Transition	Operations	Costs per exponent bit
$S_0 \to S_0$	DBL(XYZ)	$3\mathbf{M} + 4\mathbf{S}$
$S_0 \to S_1$	DBL$(XYZT)$	$4\mathbf{M} + 4\mathbf{S}$
$S_1 \to S_0$	ADD(XYZ) + 2 DBL(XYZ)	$\frac{1}{2}(13\mathbf{M} + 8\mathbf{S})$
$S_1 \to S_1$	ADD(XYZ) + DBL(XYZ) + DBL$(XYZT)$	$\frac{1}{2}(14\mathbf{M} + 8\mathbf{S})$

Table 5.2: Costs of the transitions of Figure 5.3.

This increases the code complexity, and was not tested in the reference implementation.

6 Performance results

The signature scheme from Chapter 3 was implemented in Java Card on the Cosmo 5.4 smart card. The implementation is described in Chapter 5. This chapter lists the performance of the implementation. For most of the performance improvements suggested in Chapter 5, the performance with and without the improvement was measured.

The execution time was selected as the most important benchmark. The implementation attempts to minimize both the time for signing, and the time for verification. Since smart cards are quite restricted in memory, the amount of EEPROM used by the applet was selected as a second benchmark. Where a time/memory tradeoff was made, the used EEPROM was measured as the size of the compressed Java Card applet (a .cap file), plus the EEPROM used for storing values, such as precomputed points. RAM use was not restricted further than it already is by the capacity of the smart card.

The execution time was measured with a stopwatch. This timing method is described in Appendix A.

6.1 Finite field arithmetic

Operation	Time (ms)
Modular addition	37 ± 1
Modular subtraction	37 ± 1
Modular multiplication (basic)	1195 ± 10
Modular multiplication (local and hybrid)	$\mathbf{666} \pm 1$
Modular multiplication (coprocessor)	1578 ± 10
Modular squaring	1053 ± 10
Modular squaring (local and hybrid)	$\mathbf{422} \pm 1$
Modular squaring (coprocessor)	1022 ± 10
Multiplication with a `byte` value	25 ± 1
Convert unsigned to signed	27 ± 1
Convert signed to unsigned	27 ± 1
Inversion (Euclid, blinded)	28000 ± 500
Inversion (Fermat)	113900 ± 100
Inversion (coprocessor)	1022 ± 10

Table 6.1: Time required for finite field arithmetic.

The hybrid multiplication and squaring use local variables to reduce the number of memory loads, which are slow. A window width of $d = 4$ was used. A larger window width makes the multiplication and squaring faster, but the code size increases quadratically in d. Hybrid multiplication with $d = 4$ costs 638 bytes more than basic multiplication. For squaring the additional costs are 1093 bytes.

Inversion with the blinded Binary Extended Euclidean Algorithm, is roughly 4 times as fast as the exponentiation method that uses Fermat's Little Theorem. The time to compute an inversion is equivalent to 42**M** for Euclid's inversion, and 171**M** for Fermat's inversion. The large variance in Euclid's inversion is due tue the nature of the algorithm. For Fermat's inversion, the large variance is because only two measurements were done. The coprocessor makes inversion very fast, taking less time than 2**M**.

6.2 Point addition and doubling

The time required for point addition and point doubling was measured. The results are presented in Table 6.2. The point additions assume that one of the points is represented as $(Y-X : Y+X : 2Z : kT)$.

Operation	Time (ms)
ADD $(X : Y : Z : T) \rightarrow (X : Y : Z)$	4980 ± 20
ADD $(X : Y : Z : T) \rightarrow (X : Y : Z : T)$	5640 ± 20
DBL $(X : Y : Z) \rightarrow (X : Y : Z)$	3910 ± 20
DBL $(X : Y : Z) \rightarrow (X : Y : Z : T)$	4550 ± 20

Table 6.2: Time required for point additions and doublings on a twisted Edwards curve.

6.3 Other operations

This section lists the timing results of the operations needed for the signature scheme of Chapter 3, except operations discussed in Section 6.1 and Section 6.2. These are operations like hashing, or randomisation. SHA-256 is a built-in algorithm that runs on the cryptographic coprocessor.

Operation	Time (ms)
SHA-256 with 128 bytes input	22 ± 1
Precompute $G, 2G, \ldots, 16G$	80000 ± 300
Randomise 16 points $(X : Y : Z : T)$	68300 ± 300
Masking with 16 precomputed points	3080 ± 20

Table 6.3: Time required for hashing, precomputing, randomising, and masking.

6.4 Point-scalar multiplication

No time was spent on testing the point-scalar multiplication. Table 6.4 shows the estimated time for different point-scalar multiplications. Testing these operations would have taken several hours, and was therefore considered not worth the time.

Operation	Time (s)
Point-scalar multiplication	1420 ± 1
Point-scalar multiplication (protected)	1709 ± 1
Double point-scalar multiplication	1656 ± 1
Quadruple point-scalar multiplication	2437 ± 1

Table 6.4: Estimated time required for point-scalar multiplication on a twisted Edwards curve. Note that the time is in seconds. The time was computed from the results in the previous sections. No measurements were done.

The double point-scalar multiplication is more efficient than a protected single point-scalar multiplication. The reason for this is that the verification algorithm needs no protection against side channel attacks. Therefore, no masking is needed, which saves 197 seconds. Furthermore, some additions may be skipped, saving another 126 seconds. Because randomisation of the precomputed points is not needed, an additional 80 seconds are saved.

The methods above use 12 to 16 precomputed points. This costs 2048 bytes of EEPROM. For the protected point-scalar multiplication 16 precomputed points is optimal. If a window width of 5 were used with $2^5 = 32$ precomputed points, the costs of masking would double. The masking costs for window width 4 are listed in Table 6.3. Because of the masking costs, a window width of 5 would result in a slower protected point-scalar multiplication.

For the sliding window verification method from Section 5.3.2, 12 precomputed points is optimal. A window width of w requires $\frac{3}{4}2^{2w}$ precomputed points. The precomputation costs $(2^w - 2)$ DBL and $(\frac{3}{4}2^{2w} - 2)$ ADD. The double point-scalar multiplication costs $\frac{1536}{w}\mathbf{M} + 816\mathbf{M} + 1024\mathbf{S}$, counting in the same way as Section 5.3.2. So the total costs are $(3 \cdot 2^{2w+1} + 2^{w+2} + \frac{1536}{w})\mathbf{M} + 2^{w+2}\mathbf{S} + 792\mathbf{M} + 1016\mathbf{S}$. These costs are minimized for $w = 2$, regardless of the $\mathbf{S}/\mathbf{M}$ ratio.

The quadruple point-scalar multiplication uses 15 precomputed points, which are all possible combinations of 4 points, except for the neutral element $\mathcal{O}$. Increasing the window width from 1 to 2 would require precomputing 255 points, which is not profitable.

6.5 Signing and verification

Table 6.5 shows the estimated time for signing, verification, and batch verification of 16 signatures. No measurements were done because these would take very long.

Operation	Time (s)
Sign	1715 ± 1
Verify	1662 ± 1
Batch verify 16 signatures	11238 ± 10

Table 6.5: Estimated time required for signing and verification. Note that the time is in seconds. The time was computed from the results in the previous sections. No measurements were done.

All signing and verifying operations assume the message is at most 32 bytes. There is a negligible increase in computation time for longer messages. Longer messages were inconvenient to test, because they would require a longer array to call the SHA-2 function on the coprocessor. For such an array not enough memory was available, so longer messages would require redesigning the source code. The verification does not include checking the validity of Q_A. It is assumed that Q_A is in a list of known public keys, and that the checking takes negligible time.

For batch verification the 16-tuple point-scalar multiplication was split into 4 quadruple point-scalar multiplications. The Bos-Coster algorithm described in [56] is more efficient for a large number of signatures. We do not expect a smart card to handle large batches of signatures, so therefore a quadruple point-scalar multiplication was conisdered sufficient.

The batch verification with 4 quadruple point-scalar multiplications, is more than twice as fast as 16 separate verifications. In this case one batch verification is done. Two verifications would cost 22477 seconds, which is still 15% more efficient than 16 separate verifications.

7 Conclusions

This report presents a digital signature scheme for twisted Edwards curves in Chapter 3. The scheme is more efficient than ECDSA, and slightly more efficient than EC-KCDSA. It is completely inversion-free, which reduces the size of the source code. The scheme allows efficient batch verification of multiple signatures, which is not possible with ECDSA nor EC-KCDSA.

A 256-bit version of the signature scheme was implemented on smart card; the Cosmo ID One Lite v 5.4. The implementation was programmed in Java Card 2.1.1. The implementation is designed to be efficient. It uses twisted Edwards curves, which provide the fastest elliptic curve arithmetic presently known. The most important speedups are achieved through the clever selection of a finite field and elliptic curve parameters, using local variables in Java Card, fast squaring, and hybrid multiplication. For signing, an additional speedup is achieved through precomputations. For verification, an additional speedup is achieved through combined point-scalar multiplication.

Side channel attacks are countered by four different measures. First of all, the signing algorithm is regular. The point-scalar multiplication rG always consists of exactly the same operations, independent of r. This counters timing analysis and simple power analysis. Second, a blinding technique is used. All points used in the computation are randomised by multiplying each projective coordinate with a random constant. The point $\mathcal{O}$ is avoided, because it has 0 coordinates that cannot be randomised. The blinding technique counters differential power analysis. Third, a masking technique is used. This prevents that precomputed points can be distinguished when they are loaded from EEPROM. Fourth, it is checked that the result of the point-scalar multiplication rG is a point on the elliptic curve $\mathcal{E}$. This prevents differential fault attacks. The check uses three coordinates X, Y, and Z, which ensures that the odds of a successful fault attack are negligible.

From Chapter 6 it is clear that Java Card is not a suitable platform for implementing custom cryptographic algorithms. This chapter also shows that a smart card per se is not an unsuitable platform for secure cryptography. A 1024-bit modular exponentiation takes about 1 second on the Cosmo 5.4, which means that a 256-bit point-scalar multiplication on a twisted Edwards curve should take between 0.1 and 0.2 seconds. To make Java Card a competitive environment for developing fast smart card cryptography, its API must be extended to include modular multiplication, addition, and subtraction on the cryptographic coprocessor.

A Timing of smart card operations

The timing of smart card operations was done by hand with a stopwatch. All measurement were carried out by the author of this thesis. The mean measured reaction time is $0.20s$, so this time is subtracted from each measurement. The timing method causes additional variance in the results. In road traffic, reaction times usually have a skewed probability distribution with a heavier tail for long reaction times than for short reaction times [37]. However, the reaction time for handling a stopwatch does not include any complex neural activity such as a traffic situation analysis. It turns out that the reaction times fit a normal distribution. This is concluded from Figure A.1 which shows the probability density of human reaction time, and Figure A.2 which compares the probability distribution to a normal distribution. Figure A.2 shows that the reaction time deviates from a normal distribution near the tails. The left tail is heavier than normal, while the right tail is lighter. This is opposite to the distribution of road traffic reaction times. Lacking a better model, the normal distribution was chosen as the most suitable.

The distribution of the human reaction time has a sample mean of $0.20s$ and a sample standard deviation of $0.035s$.

For the benchmarks in this report, one measurement typically consists of 32767 repetitions of the evaluated operation. $32767 = 2^{15} - 1$ is the largest value that fits in a signed SHORT value. Hence, this is the largest number of iterations that can be done on a Java Card by using a simple loop with a check against zero. The duration of a single measurement is between several seconds and one minute. For operations that take longer than 2 ms, fewer than 32767 repetitions were done.

Measurements were carried out with different numbers of repetitions. For example, modular multiplication was done 0 times, 1 time, 2 times, 4 times, and 8 times. The time for a modular multiplication was determined with the linear fit shown in Figure A.3. For all measurement series the linear fit has an R^2 of at least 0.99. This results in accurate timings for the measured operations.

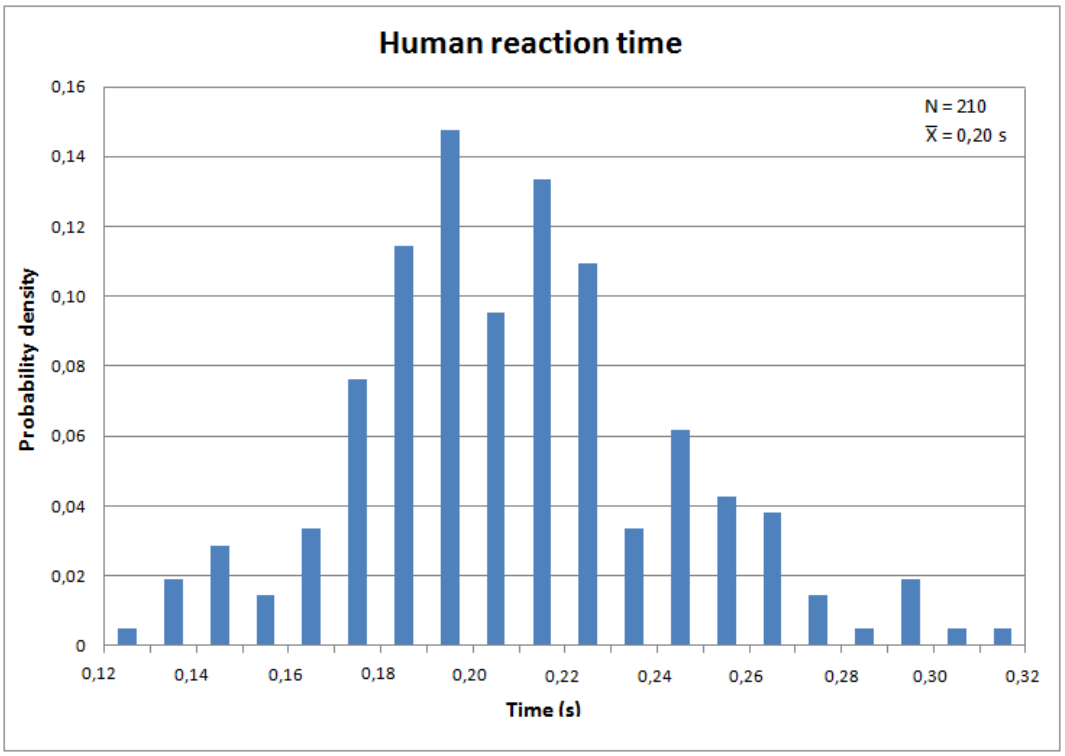


Figure A.1: The probability density of the human reaction time for pressing a stopwatch after output appears on a screen. The data consists of 210 measurements taken by a single person.

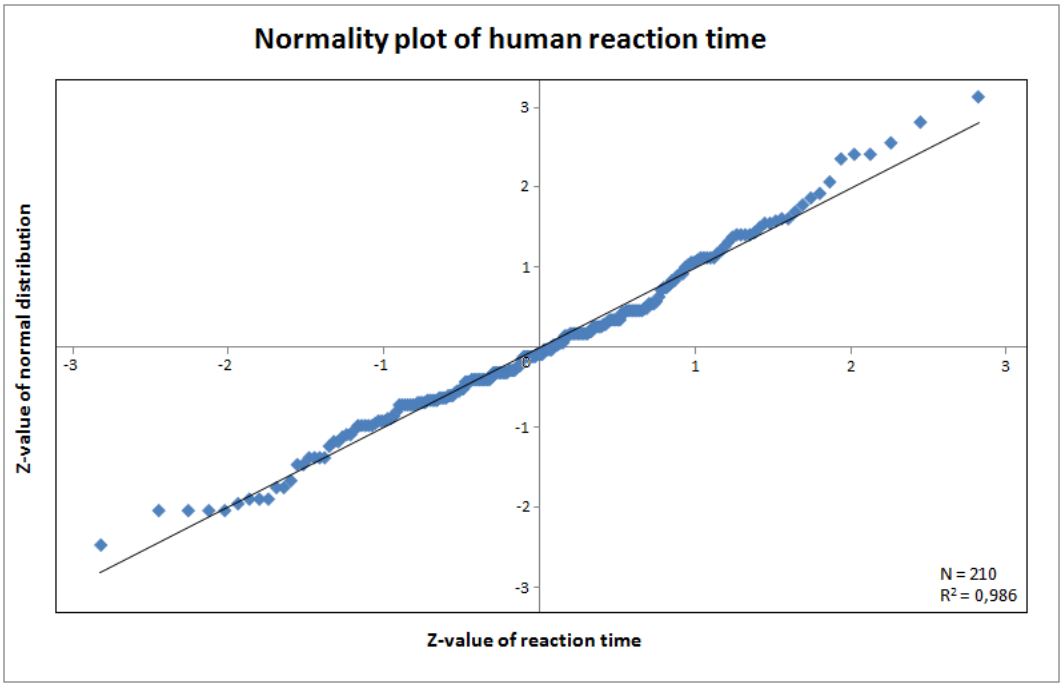


Figure A.2: A normality plot of the human reaction time for pressing a stopwatch after output appears on a screen. The data consists of 210 measurements taken by a single person. Z-values were computed by subtracting the sample mean, then dividing by the sample standard deviation.

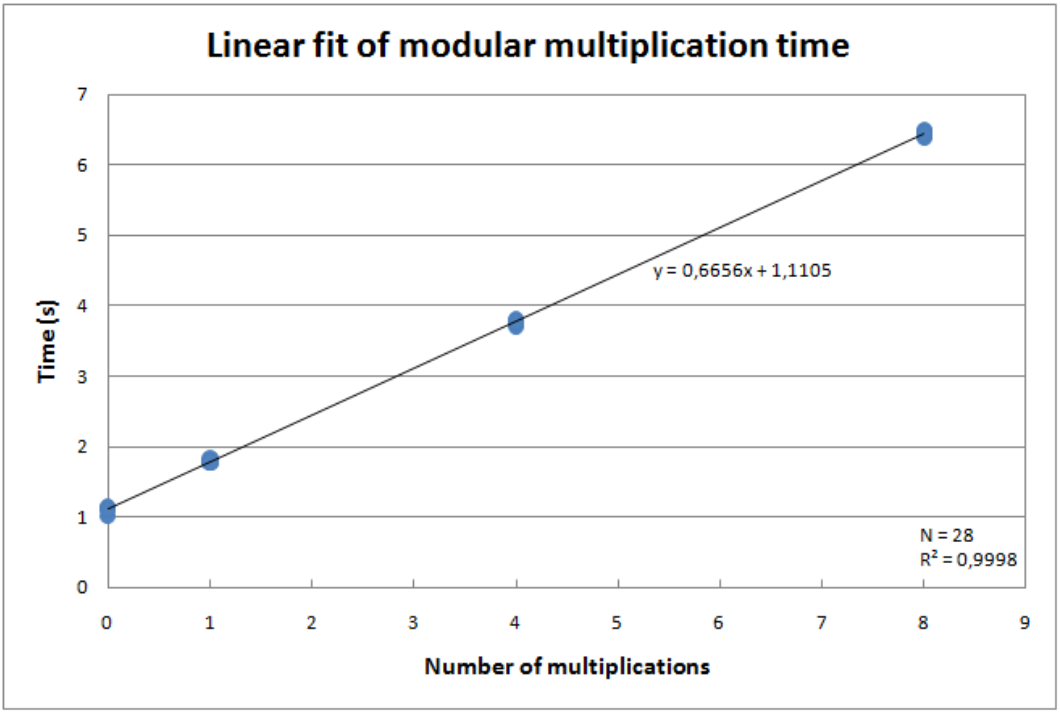


Figure A.3: Linear fit of the time required for a multiplication modulo $2^{255} - 31$ on the Cosmo 5.4. Each cloud of points consists of 7 measurements. The time for one multiplication is 0.6656 s.

B Unexpected RAM behaviour

In the Cosmo 5.4 reading data from RAM results in unexpected behaviour. The time it takes to read a byte from an array depends on the index at which this byte is stored. This phenomenon occurs both in CLEAR_ON_DESELECT and CLEAR_ON_RESET RAM. Figure B.1 shows the mean time required for reading a byte from an array in CLEAR_ON_RESET RAM as a function of the array index. For index 6 and 8 more accurate measurements were taken. These reveal that the array access time for these indices are different, which means that there are at least three different array access times.

When multiple arrays are declared, this behaviour is identical for all arrays. This means that while the array indices may be identified by a simple power analysis, there is no difference in timing between different arrays. This is provided that the arrays all use the same memory, for example CLEAR_ON_RESET RAM.

Distinguishing array indices is no security concern; array elements are processed consecutively in a loop, so the indices may simply be obtained from the order of execution. However, it is important that arrays cannot be distinguished from one another. If this were possible, an attacker could figure out which points on the elliptic curve are added or doubled and thereby obtain the secret key.

There are several possible explanations for this RAM behaviour. One explanation is the hardware layout of the smart card. Some RAM might be directly addressable, while other RAM requires a longer address or an indirect call. Since the total RAM is 1364 bytes, an address wil not always fit in 8 bits, which is the word length of the processor. Another explanation lies in the array index checking by Java Card. Checking whether an address is in the permissible range, may be done from the most to the least significant bit of the address. The number of bits that needs to be checked depends on the address and on the bounds of the permissible range. Both effects may play a role.

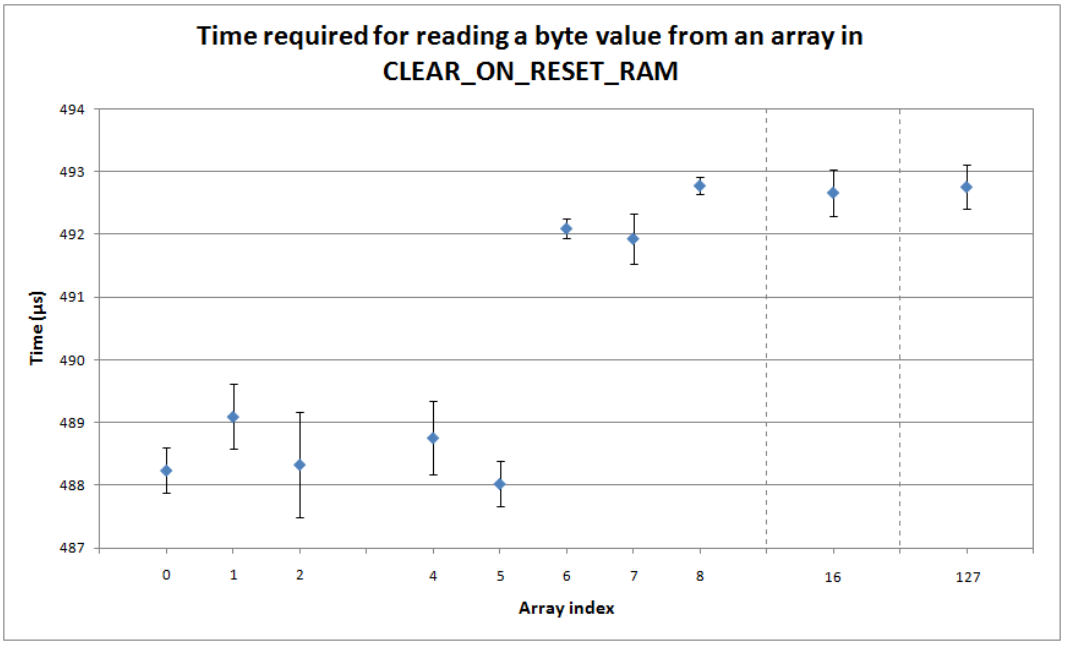


Figure B.1: The mean time required for reading a byte from an array in CLEAR_ON_RESET RAM as a function of the array index. The intervals are 95 % confidence intervals.

Bibliography

[1] —, *19-jarige hacker aangehouden* (in Dutch), Openbaar Ministerie, August 2008, `www.om.nl/actueel/nieuws-_en/@148582/19-jarige_hacker`

[2] —, *Cosmo 64 RSA T V5.4 Technical Brief*, Manual reference 064471 01 UDD AC, Oberthur Card Systems, 2007

[3] —, *Federal Information Processing Standards Publication (FIPS PUB 186-3)*, National Institute of Standards and Technology, U.S. Department of Commerce, June 2009

[4] —, *Het Nieuwe Pinnen* (in Dutch), `http://www.hetnieuwepinnen.nl/index.php?option=com_content&view=article&id=58&Itemid=63`

[5] —, *ID-One Cosmo 64 v5.4 D FIPS 140-2 Level 3 Security Policy*, version 1.0, Oberthur Card Systems, May 2007

[6] —, *Java Card 2.1.1 Runtime Environment (JCRE) Specification*, Sun Microsystems, May 2000

[7] —, *Java Card 2.2.2 Development Kit (JDK) User's Guide*, Sun Microsystems, March 2006

[8] —, *kmGNFS - a free implementation of the General Number Field Sieve*, 2008, `http://kmgnfs.cti.gr/kmGNFS/Home.html`

[9] —, *Public Key Cryptography for the Financial Services Industry*, ANSI X9.62, The Elliptic Curve Digital Signature Algorithm (ECDSA), 2005

[10] —, *Standards for Efficient Cryptography (SEC)*, Certicom Research, September 2000

[11] T. Akishita and T. Takagi, *Zero-value point attacks on elliptic curve cryptosystem*, ISC 2003, LNCS 2851, Springer, pp. 218–233

[12] R. Anderson and M. Kuhn, *Tamper resistance - a cautionary note*, Proceedings of the Second Usenix Workshop on Electronic Commerce, 1996, pp. 1-11

[13] K. Aoki, J. Guo, K. Matusiewicz, Y. Sasaki, and L. Wang, *Preimages for step-reduced SHA-2*, ASIACRYPT 2009, LNCS 5912, pp. 578–597

[14] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren, *Handbook of elliptic and hyperelliptic curve cryptography*, CRC Press, 2005, ISBN: 1584885181

[15] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, *The Sorcerers Apprentice Guide to Fault Attacks*, Proceedings of the IEEE 94 no. 2, February 2006

[16] L. Batina and C. Jansen, *Secret exponent information leakage for timing analyses*, Proceedings of the 23rd Symposium on Information Theory in the Benelux, 2002, pp. 225–232

[17] L. Batina and C. Jansen, *Side-Channel entropy for modular exponentiation algorithms*, Proceedings of the 24th Symposium on Information Theory in the Benelux, 2003

[18] D. J. Bernstein, *Efficient arithmetic in finite fields*, Workshop on Cryptographic Algorithms and Protocols, Mendes Convention Center, Santos, August 2006, http://cr.yp.to/talks/2006.08.30-1/slides.pdf

[19] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters, *Twisted Edwards Curves*, AFRICACRYPT 2008, LNCS 5023, pp. 389–405

[20] D. J. Bernstein and T. Lange, *Analysis and optimization of elliptic-curve single-scalar multiplication*, Contemporary Mathematics 461, AMS 2008, pp. 1–19, http://cr.yp.to/antiforgery/efd-20071204.pdf

[21] D. J. Bernstein and T. Lange, *Explicit-Formulas Database*, http://hyperelliptic.org/EFD/

[22] D. J. Bernstein and T. Lange, *Faster Addition and Doubling on Elliptic Curves*, ASIACRYPT 2007, LNCS 4833, pp. 29–50

[23] R. Bröker and P. Stevenhagen, *Constructing elliptic curves of prime order*, Comtemporary mathematics 463, AMS 2008, pp. 17–28

[24] D. R. L. Brown, *Generic groups, collision resistance, and ECDSA*, Des. Codes Cryptography 35 no. 1, 2005, pp. 119–152

[25] J. W. S. Cassels, *Lectures on Elliptic Curves*, LMSST 24, Cambridge University Press, November 1991

[26] L. Clozel, M. Harris, R. Taylor, *Automorphy groups for some l-adic lifts of automorphic mod l Galois representations*, Publ. Math. Inst. Hautes Études Sci. 108, 2008, pp. 1–181

[27] J. S. Coron, Y. Desmedt, D. Naccache, A. Odlyzko, and J. P. Stern - *Index calculation attacks on RSA signature and encryption*, Designs, Codes, and Cryptography 38 no. 1, 2006, pp. 41–53, www.jscoron.fr/publications.html#indexcal

[28] P. Flajolet and A. M. Odlyzko, *Random mapping statistics*, EUROCRYPT 1989, LNCS 434, Springer, pp. 329–354

[29] P. A. Fouque, R. Lercier, D. Réal, and F. Valette, *Fault Attack on Elliptic Curve with Montgomery Ladder Implementation*, FDTC 2008, pp. 92–98

[30] S. Friedl, *An elementary proof of the group law for elliptic curves*, January 2004, http://math.rice.edu/~friedl/papers/AAELLIPTIC.PDF

[31] M. Fürer, *Faster integer multiplication*, Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, June 2007

[32] L. C. C. García, *Can Schönhage multiplication speed up the RSA encryption or decryption?*, Darmstadt University of Technology, 2005, http://www.cdc.informatik.tu-darmstadt.de/~coronado/Vortrag/MoraviaCrypt-talk-s.pdf

[33] L. Goubin, *A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems*, PKC 2003, LNCS 2567, Springer, pp. 199–211

[34] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, *Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs*, CHES 2004, LNCS 3156, pp. 119–132

[35] M. Harris, *The Sato-Tate conjecture: introduction to the proof*, October 2007, www.institut.math.jussieu.fr/~harris/SatoTate/notes/Introduction.pdf

[36] H. Hisil, K. K. H. Wong, G. Carter, and E. Dawson, *Twisted Edwards Curves Revisited*, ASIACRYPT 2008, LNCS 5350, Springer, pp. 326–343, http://eprint.iacr.org/2008/522.pdf

[37] W. Hugemann, *Driver reaction times in road traffic*, EVU 2002, SLOVENIJA, Portorož, September 2002

[38] E. Jochemsz and A. May, *A Polynomial Time Attack on RSA with Private CRT-Exponents Smaller Than $N^{0.073}$*, CRYPTO 2007, LNCS 4622, pp. 395-411

[39] A. A. Karatsuba and Y. Ofman, *Multiplication of multi-digit numbers on automata*, Soviet Physics Doklady 7, 1963, pp. 595–596

[40] KCDSA Task Force Team, *The Korean certificate-based digital signature algorithm*, ASIACRYPT 1998, August 1998

[41] H. S. Kim, T. H. Kim, J. C. Yoon, and S. Hong, *Practical Second-Order Correlation Power Analysis on the Message Blinding Method and Its Novel Countermeasure for RSA*, ETRI Journal 32 no. 1, February 2010, pp. 102–111

[42] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann, *Factorization of a 768-bit RSA modulus*, CRYPTO 2010, pp. 333–350, `http://eprint.iacr.org/2010/006`

[43] D. Knuth, *The Art of Computer Programming*, Volume 2: Seminumerical Algorithms, Section 4.5.2: The Greatest Common Divisor, Third Edition, Addison-Wesley, 1997, pp. 333-356

[44] G. J. Lay and H. G. Zimmer, *Constructing elliptic curves with given group order over large finite fields*, ANTS 1994, LNCS 877, pp. 250–263

[45] H. W. Lenstra Jr., *Factoring integers with elliptic curves*, Annals of Mathematics (2) 126, 1987, pp. 649–673

[46] H. Mamiya, A. Miyaji, and H. Morimoto, *Efficient Countermeasures against RPA, DPA, and SPA*, CHES 2004, LNCS 3156, pp. 343–356

[47] S. Manuel, *Classification and generation of disturbance vectors for collision attacks against SHA-1*, Cryptology ePrint Archive, Report 2008/469, `http://eprint.iacr.org/2008/469.pdf`

[48] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, Chapter 5: Pseudorandom Bits and Sequences, CRC Press, 1996, `http://www.cacr.math.uwaterloo.ca/hac/about/chap5.pdf`

[49] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, Chapter 11: Digital Signatures, CRC Press, 1996, `http://www.cacr.math.uwaterloo.ca/hac/about/chap5.pdf`

[50] B. Möller, *Improved techniques for fast exponentiation*, ICISC 2002, LNCS 2587, pp. 298-312

[51] P. L. Montgomery, *Modular multiplication without trial division*, Mathematics of computation 44 no. 170, April 1985 pp. 519–521

[52] P. Paillier and D. Vergnaud, *Discrete-log-based signatures may not be equivalent to discrete log*, ASIACRYPT 2005, LNCS 3788, Springer, 2005, pp. 1–20

[53] J. M. Pollard, *Monte Carlo methods for index computation mod p*, Mathematics of Computation 32 no. 143, 1978, pp. 918–924

[54] J. M. Pollard, *Theorems of factorisation and primality testing*, Proceedings of Cambridge Philosophical Society 76, 1974, pp. 521–528

[55] R. L. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM 21 (2), 1978, pp. 120–126

[56] P. de Rooij, *Efficient exponentiation using precomputation and vector addition chains*, EUROCRYPT 1994, LNCS 950, Springer, 1995, pp. 389-399

[57] S. K. Sanadhya and P. Sarkar, *New collision attacks against up to 24-step SHA-2*, INDOCRYPT 2008, LNCS 5365, pp. 91–103

[58] Y. Sasaki, L. Wang, and K. Aoki, *Preimage attacks on 41-step SHA-256 and 46-step SHA-512*, Cryptology ePrint Archive, Report 2009/479, `http://eprint.iacr.org/2009/479.pdf`

[59] P. Sestoft, *Java performance: Reducing time and space consumption*, Royal Veterinary and Agricultural University of Copenhagen and IT University of Copenhagen, version 2, April 2005

[60] A. Schönhage and V. Strassen, *Schnelle Multiplikation großer Zahlen*, Computing 7, 1971, pp. 281-292

[61] V. Shoup, *OAEP reconsidered*, CRYPTO 2001, LNCS 2139, pp. 239–259, `www.shoup.net/papers/oaep.pdf`

[62] S. Skorobogatov and R. J. Anderson, *Optical fault induction attacks*, CHES 2002, LNCS 2523, pp. 2–12

[63] J. Stern, D. Pointcheval, J. Malone-Lee, and N. P. Smart, *Flaws in Applying Proof Methodologies to Signature Schemes*, CRYPTO 2002, LNCS 2442, Springer, pp. 93–110, `http://dx.doi.org/10.1007/3-540-45708-9_7`

[64] J. Swierczewski and W. Stein, *Connections Between the Riemann Hypothesis and the Sato-Tate Conjecture*, June 2008, `http://modular.math.washington.edu/projects/swierczewski.pdf`

[65] R. Taylor, *Automorphy groups for some l-adic lifts of automorphic mod l Galois representations II*, Publ. Math. Inst. Hautes Études Sci. 108, 2008, pp. 183–239

[66] A. L. Toom, *The complexity of a scheme of functional elements realizing the multiplication of integers*, Soviet Mathematics Doklady 3, 1963, pp. 714–716

[67] X. Wang and H. Yu, *How to break MD5 and other hash functions*, EUROCRYPT 2005, LNCS 3494, 2005, pp. 19–35

[68] X. Wang, H. Yu, and Y. L. Yin, *Efficient collision search attacks on SHA-0*, CRYPTO 2005, LNCS 3621, 2005, pp. 1–16

[69] E. W. Weisstein, *Fermat's Little Theorem*, MathWorld - A Wolfram Web Resource, `http://mathworld.wolfram.com/FermatsLittleTheorem.html`

[70] M. J. Wiener and R. J. Zuccherato, *Faster Attacks on Elliptic Curve Cryptosystems*, April 1998, Entrust Technologies

[71] H.C. Williams, *A $p+1$ method of factoring*, Mathematics of Computation 39 no. 157, July 1982, pp. 225–234

[72] D. Zuras, *More On Squaring and Multiplying Large Integers*, IEEE Transactions on Computers 43 no. 8, August 1994, pp. 899–908